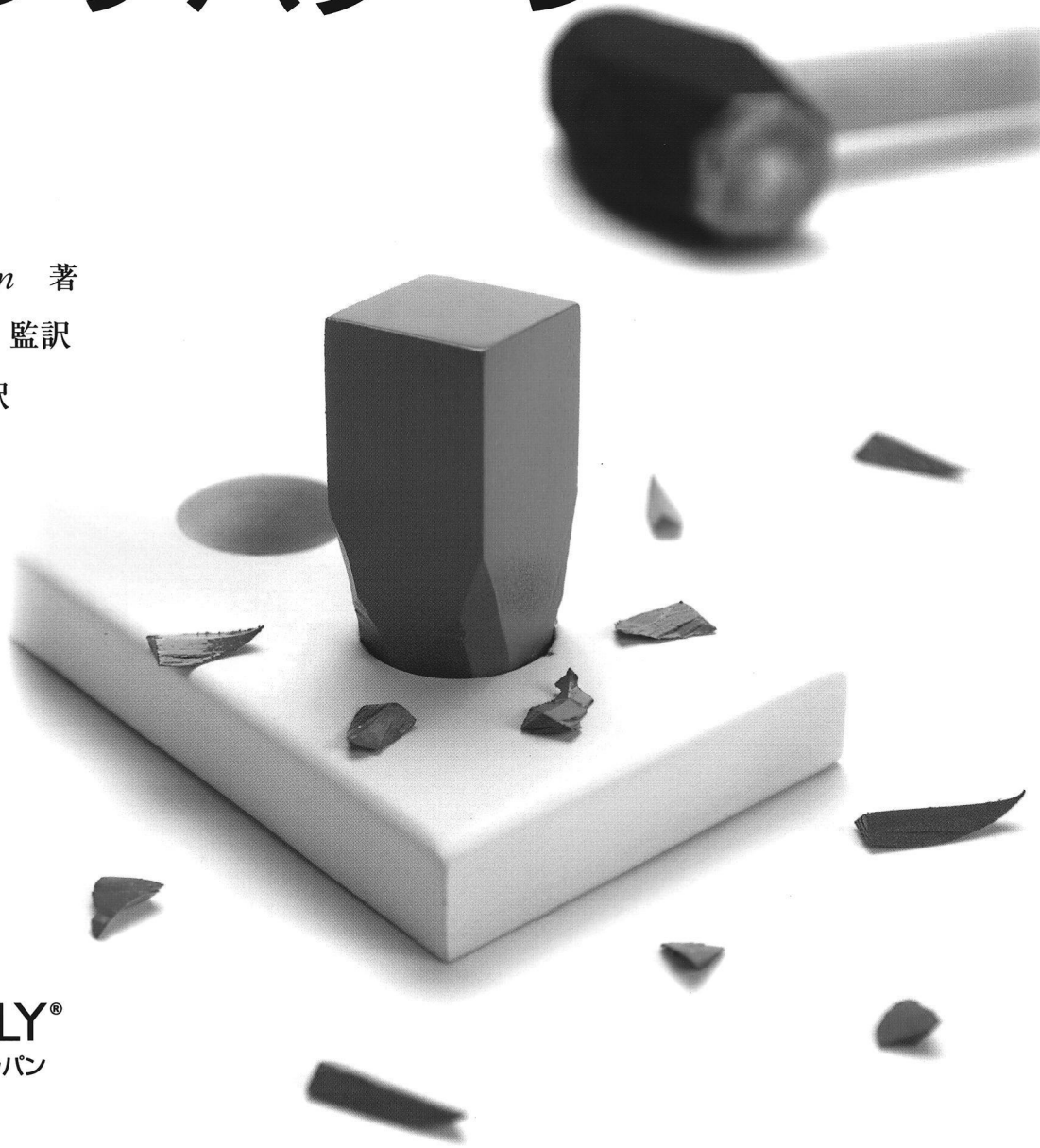
Avoiding the Pitfalls of Database Programming

SQL アンチパターン



Bill Karwin 著
和田 卓人
和田 省二 監訳
児島 修 訳

O'REILLY®
オライリー・ジャパン

Avoiding the Pitfalls of Database Programming

SQL アンチパターン



Bill Karwin 著
和田 卓人 監訳
和田 省二
児島 修 訳

O'REILLY®
オライリー・ジャパン

SQLアンチパターン

Bill Karwin 著
和田 卓人、和田 省二 監訳
児島 修 訳

O'REILLY®
オライリー・ジャパン

本書で使用するシステム名、製品名は、それぞれ各社の商標、または登録商標です。
なお、本文中では、™、®、© マークは省略しています。

SQL Antipatterns

Avoiding the Pitfalls of Database Programming

Bill Karwin

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

© 2010 Bill Karwin. Title of English-language original: SQL Antipatterns, ISBN978-1-93435-655-5. © 2013 O'Reilly Japan, Inc. Authorized translation of the English edition © 2010 The Pragmatic Programmers LLC. This translation is published and sold by permission of The Pragmatic Programmers LLC., the owner of all rights to publish and sell the same.

本書は、株式会社オライリー・ジャパンがThe Pragmatic Programmers LLC. の許諾に基づき翻訳したものです。日本語版についての権利は、株式会社オライリー・ジャパンが保有します。

日本語版の内容について、株式会社オライリー・ジャパンは、最大限の努力をもって正確を期していますが、本書の内容に基づく運用結果についての責任を負いかねますので、ご了承ください。

本書への称賛の声

SQLの強力さ故にその深みにはまる技術者にとって、この本は救いの一冊となるはずです。

大中浩行
devtesing-ja 主宰

業務システムの開発を生業としているからには、RDBMSと無関係ではいられません。本書で述べられているアンチパターンの数々は、過去の苦い経験を思い起こさせる教訓として、また、これから体験するであろう災難を未然に防ぐ知恵として、チームの皆に是非読んでおいてもらいたい内容だと思います。

岡澤裕二
ソフトウェア開発者

アンチパターン形式で適度な長さにまとめられたそれぞれのトピックがいちいち面白くタメになります。中にはBillの考え方に必ずしも賛同しきれないトピックもありますが、著者の考えを押し売りすることなく多様な観点が提示されており、十分フェアな記述になっています。初心者にも有益なことは間違いないですが、これまでデータベース的負債を積み上げてきた中級以降の開発者であればより楽しめることでしょう。

中野靖治
『プログラミングGROOVY』（技術評論社）共著者

本書では、リレーショナルデータベースの基礎が学べるのと共に、これまで自分自身が重ねてきた失敗、また、これからおかしてしまうかもしれない失敗を、先人の貴重な経験を通して発見し、その対処法を身に付ける事ができます。初学者、経験者問わず、リレーショナルデータベースを扱うエンジニアにとっては、一読の価値があります。本書はパターン本ですので、全編順に通して読む必要は無く、カタログ的に関心のある章からつまんで読む、といった使い勝手の良さもあります。日常のソフトウェア開発において、手元に置いておきたい一冊です。

廣田俊浩
ソフトウェアエンジニア

本書では、理論だけでは学ぶことが難しい「RDB 設計の落とし穴」が数多く紹介されています。過去に自分がした設計を思い返すと、本書を読んでいたら避けていたであろう設計がいくつも頭に浮かびました。本書によって、これらの落とし穴を的確に認識し、適切な対処が出来るようになるはずです。

都元ダイスケ
ソフトウェアエンジニア

適切なデータベース設計を行うためには、様々な観点で考慮が必要であり、経験がものをいう部分がありました。本書を読むことで、データベース設計に関わる他者の経験を自分の学びの糧にできます。データベース設計に関わる実務者にとってきわめて有益な書籍であるため、是非一読をお勧めします。

渡部 亮太
株式会社コーソル プリンシパル・エンジニア
Japan Oracle User Group 共同設立者、ボードメンバー

私はベストプラクティスの価値を高く認めている。他者の失敗から学ぶことが好きなのだ。この本には、そのような他者によるあらゆる失敗が網羅されている。驚いたのは、私自身が犯してきたいくつかのミスも含まれていたことだ。もっと早くこの本と出会いたかった。

Marcus Adams
シニアソフトウェアエンジニア

Bill は、興味深く、実用的で、重要で、ユニークな本を書いた。ソフトウェア開発者は、本書に書かれたアンチパターンと解決策から大きなメリットを得られるだろう。私もさっそく本書のテクニックを用いてアプリケーションを改善した。本当に素晴らしい本だ。

Frederic Daoud
『Stripes: ...And Java Web Development Is Fun Again』『Getting Started with Apache Click』著者

『SQL アンチパターン』は、本書で記述されたデータベース設計における選択に頻繁に直面するソフトウェア開発者にとって必読の一冊だ。開発チームは、データベース設計がもたらす結果を深く理解し、要件、期待、測定値、現実に基づいた最善の決定を行えるようになるだろう。

Darby Felton
DevBots Software Development 社　共同創設者

Billの本書へのアプローチがとても気に入った。彼のユニークなスタイルとユーモアのセンスがよく表れている。それは、下手をしたら無味乾燥なものになりかねないトピックを取り上げるうえでとても重要なものだ。本書はとてもわかりやすい形式で記述されているので、開発者はその内容にアクセスしやすく、リファレンスとしての有用性も高い。つまり、pragmatic bookshelfに新たな価値ある一冊が加わったということだ。

Arjen Lentz
Open Query（http://openquery.com） エグゼクティブ・ディレクター
『実践ハイパフォーマンス MySQL 第2版』[SZTZ08] 共著者

本書は、SQLデータベースの長年の実践的な経験の所産である。すべてのトピックの内容は濃く、細部への細かな配慮は私の期待を大きく上回るものだった。初心者向けではないが、ある程度のSQLの経験のあるすべての開発者にとっては、必ずと言っていいほど新たな発見のある、有用なリファレンスになるだろう。

Mike Naberezny
Maintainable Software社 パートナー
『Rails for PHP Developers』共著者

SQLの基礎は知っているものの、SQLデータベースの設計でそれ以上の知識が求められているソフトウェアエンジニアにとって、極めて価値のある一冊だ。

Liz Neely
シニア・データベース・プログラマー

Karwinの本には、良質で実践的なアドバイスが満載だ。出版された時期もいい。多くの人々が新しい、華やかなテーマに注目しているなか、プロフェッショナルにとって、本書はSQLの知識を磨くための最高の一冊だ。

Maik Schmidt
『Enterprise Recipes with Ruby and Rails』『Enterprise Integration with Ruby』著者

Billは、本書で取り上げられている数多くの失敗の本質を鋭く捉えることに成功している。これらの失敗は、おそらく私たちすべてが過去のいずれかの時点で、落とし穴に陥っていると気づかないままに陥ったことがあるものだ。本書で登場するアンチパターンには、「こんな失敗をしていたなんて信じられない（しかも繰り返し）！」という発見から、最善の解決策が、SQLを学んでいたときには正しいとされていたものとは反することもあるというトリッキーなシナリオに至るまで、実に様々である。SQLの初心者から専門家に至る、あらゆるレベルの人にとって有益な書物だ。

Danny Thorpe
Microsoft社 プリンシパル・エンジニア
『Delphi Component Design』著者

監訳者まえがき

和田卓人によるまえがき

諸君は自らの経験からいくらか学ぶことができるという、全く愚かな考えであろうが、
余はむしろ他人の失敗を学ぶことで、自分の失敗を回避することを好む。
——オットー・フォン・ビスマルク

失敗から学ぶ

「愚者は経験に学び、賢者は歴史に学ぶ」。ドイツの宰相ビスマルクが語ったと言われる格言ですが、実はこれが造句であるという説はまだあまり知られていません。実際は「愚者は自分が痛い目を見ることでしか学ばず、賢者は他者の失敗経験からも学ぶ」という趣旨の発言であったという説が濃厚なのだそうです。本書を監訳するにあたり、このビスマルクの言葉は何度も私の胸に去来しました。

私たちは失敗について、どれくらい知っているでしょうか。この業界の常として、成功事例は多く喧伝され、失敗事例には口がつぐまれます。しかし、成功事例よりも、むしろ失敗事例の方が一般性を持っているものです。失敗事例の一因である「技術に対する理解の浅さと研鑽不足」は、その結果を身をもって体験しないと何が問題であったかも分からないという構造的な難しさを抱えています。私たちは、痛い目を見てから学ぶのではなく、もっと積極的に他者の失敗から学ばなければなりません。

アンチパターンとは「べからず集」

この本の章立ては全て同じ形式になっていることを、本書の目次を見てお気づきになった方もいらっしゃるでしょう。この形式は「パターン」形式と呼ばれます。オブジェクト指向の世界では広く知られている「パターン」は、単なる設計カタログではありません。パターンは名前と形式（フォーム）を重んじ、先人の叡智を次代に伝える役割を持ちます。パターンを学んだ開発者は、例えば「Strategy」という名前を聞いただけで、設計の背景と一般的な実装を共有することができます。しかも、「アンチパターン」はただのパターンではなく、パターンとしての形式が整った「べからず集」であり、「転ばぬ先の杖」の役割を持っています。

著者 Bill Karwin について

システム開発の本は数あれど、著者の得意分野からの視点で語った「べし集」は多いものの、「べからず集」は多くはありませんでした。

リレーショナルデータベースを中心に据えたシステム開発には、設計段階にも、SQLのコーディングにも、アプリケーションコードの開発にも落とし穴が口を開けています。

私たち技術者は、自分の専門分野に関連した落とし穴については積極的に知ろうとします。ただし、専門性を高めるあまり、自分の慣れ親しんだ方法を他の分野にも当てはめようとしてしまいがちです。しかし、システム構築には、設計知識と実装知識、リレーショナル理論とオブジェクト指向、セキュリティやパフォーマンス、バージョン管理やテスト、どれも欠くべからざる知識なのです。

本書の原著者 Bill Karwin 氏の素晴らしい点は、まさに設計も実装も手がけてきた人間ならではの視野の広さがあることです。Bill は設計もコーディングも、リレーショナル理論もオブジェクト指向も、よく知っています（オブジェクト指向を知る人でなければ「パターン」という名前は付けなかったでしょう）。

Bill の軸足は基本的にはリレーショナル理論と SQL なので、リレーショナルデータベースの意義と美点を強く意識した本書のメッセージは、ときには過激に聞こえることもあるでしょう（例えば賛否両論ありそうな 3 章「ID リクワイアド（とりあえず ID）」が挙げられます）。しかし、データの寿命はアプリケーションの寿命やシステムの寿命よりも長いものです。データの設計、そしてデータを格納するデータベースの設計は、情報の価値がさらに高まるこれからのシステム開発において、より重要性が高まります。

リレーショナル理論への理解不足から、データベースを単なるファイルの入れ物として扱うようでは、価値あるデータベースを育むことはできません。本書にアンチパターンとして記された他者の失敗から学ぶことで、無駄なく漏れなくデータを配置し、トランザクションやデータ整合性を重んずるリレーショナルデータベースの真価を学ぶことができるでしょう。

本書について

手元の記録を見てみると、私が原書『SQL Antipatterns』と出会ったのは 2010 年 1 月 8 日。"The Pragmatic Bookshelf"（『達人プログラマー』の著者二人が始めた出版社。Ruby 関係の書籍で有名）の近刊をチェックしていたところ、この出版社のラインナップとしては珍しい SQL 本の beta book（書籍の完成前から購入して読み始められる電子書籍）を発見しました。その内容に興味が湧き、すぐに購入したことを覚えています。当時現在進行形で書き進められていた『SQL Antipatterns』は、荒削りでありながらも、技術者の陥りやすい過ちが体系的に整理されており、有用性の高い本になる予感がしました。

Bill Karwin という（当時の私にとって）馴染みのない著者が書き始めた『SQL Antipatterns』は、「ありそうでなかった本」でした。それまでの RDBMS/SQL の本には「こうすべし」という内容が書かれた本はありました。しかし、「この方法は一見良い方法に見えるが、このようなデメリットがあり、より良い方法にはこういったものがある」というスタンスで書かれている本は珍しかったのです。本書を読み始めた頃、まさに本業で EAV 設計への設計変更を開始していた私は、5 章「EAV（エンティティ・アトリビュート・バリュー）」を読んで設計変更を引き返し、結果として多くのデータ不整合の発生を未然に防ぐことができたのでした。それは私が Bill の（あるいは EAV の章のモデルとなった誰かの）失敗経験から学んだ瞬間でした。

EAV 以外にも多くの失敗とその解法を『SQL Antipatterns』から学んだ私は、この本が日本の読者に広く読まれるべきであるという思いを抱き、『プログラマが知るべき 97 のこと』を共に手がけたオライリーの高さんに「次は『SQL アンチパターン』をやりたい」と声を掛けたのでした。

本書の邦訳には苦労が伴いました。本書の SQL は MySQL と PostgreSQL で動作を確認しつつ監訳を行いました。加えて PHP と Ruby のサンプルコードも動作確認を行い最新の内容を反映したため、結果として少なくない分量を加筆修正することになりました。それだけに、思い入れの強い本でもあります。

事実を格納するためのリレーショナルデータベースの設計、そこから情報を引き出すための SQL は、これからも重要な技術であり続けるでしょう。本書がリレーショナルデータベースや SQL の力を知るきっかけになり、よくある落とし穴を避けるための「転ばぬ先の杖」になることができたなら、監訳者としてこの上もない喜びです。

謝辞

この本の出版に関わった全ての方に感謝いたします。

まずはこの素晴らしい書籍を世に生み出した Bill Karwin 氏。突然のお願いにもかかわらずご快諾頂き、書き下ろしのスペシャルチャプター（25 章「砂の城」）を寄稿してくださった奥野幹也さん。査読者として沢山の誤りを指摘し、訳質を上げるべく議論してくださった大中浩行さん、岡澤裕二さん、中野靖治さん、廣田俊浩さん、都元ダイスケさん、渡部亮太さん（50 音順）。驚くほどのスピードで翻訳してくださった児島さん、前著に続き私の極端な遅筆に対して辛抱強く付き合ってくださった編集者の高さん、高度に自動化された組版プロセスを提供してくださったトップスタジオ様。若造のオブジェクト指向の立場からの論戦に飽きずに立ち向かい、私に RDBMS と SQL の奥深さを教えてくれた父（そう、もうひとりの監訳者、和田省二です）。そして、何度もくじけそうになる私を支え、ときに叱咤し、それだけでなく、邦訳にあたって多くのヒントをくれた妻。

皆さんのおかげでようやくこの本を世に出せます。ありがとうございます。本当に、感謝しています。

和田省二によるまえがき

本書は、RDBの論理設計からSQLプログラミング・開発手法に至るまでの正しい方法について書かれています。その類の本は世の中にかなり出回っていますが、それらの本には「正しい方法はこれだ」式の「あるべき論」を展開するスタイルが多いのが事実です。そこから果実を得るには多くの努力が必要です。

それに引き換え本書では、ありがちな良くない具体例をまず先にあげます。この部分が読み物風に実に軽妙に描かれています。いかにもありそうな生々しいシーンが多く、その場に居合わせているかのようです。思わずニヤリとしてしまいます。その上でそれを基に、「なぜ良くないのか」(かなり具体的、実証的で力が入っています)、「ではどうすれば良いのか」という展開で、無理なく正しい方法を伝えることに成功しています。中でも、1章「ジェイウォーク(信号無視)」、4章「キーレスエントリ(外部キー嫌い)」、5章「EAV(エンティティ・アトリビュート・バリュー)」は必読です。

私は、本書に出会う前から、本書に出てくる良くない事例に頻繁に遭遇してきました。

- 参照制約を一切排したプロジェクト
- ER図を作らない(というよりも、エンティティを正しく切り出していないためER図が作れない)プロジェクト
- ひとつのテーブルからSELECTするSQLのみを許し、後は手続言語により自前でゴリゴリ組むといった、SQLのおいしい部分を捨ててしまったプロジェクト

など、枚挙にいとまがありません。

ここまでひどくなくとも、RDB理論が集合論を基にしていること、集合を扱うためのSQL言語であることが正確には知られていないのが現状です。また「SQLを知らない(書いて動かしたことがない)RDB設計者」、「SQL言語が集合をデータとした宣言型言語であるのに手続型言語のように利用しようとするプログラマー」も後を絶ちません。これは憂慮すべき状況と言わざるを得ません。

さて、皆さんは「情報」と「データ」の違いをご存知でしょうか。この二者の関係は「多くの情報を秘めた貴重なデータ」という表現で言い尽くせます。いつでも我々が欲しいのは、意味のある(目的を持った)正しい情報なのです。一方、データは単なる各種の事実の値(何らかの、名称とか日付とか金額とか)であってそれ自体に目的はありません。事実は唯一無二なので同じデータを複数持つのは良くありません。そして目的を持った情報は、無目的な事実を集積したデータを種々加工して得られます。データは唯一無二の事実値ですから、それから作り出される情報はどれもが正しく、互いに整合性がとれています。

もうお分かりと思いますが、「データ」を永続化して蓄えるのに適しているのがRDBであり、それから「情報」を生成するのに適しているのがSQLなのです。

RDBに蓄えられたデータは、長い間にわたり多くの人々が更新を重ね情報取得のためにアクセスする共有財産です。その品質は保証されていなければなりません。不具合が発覚した時に「私のミスではありません!」(4章)と叫んでも何の役にも立ちません。データ保全のためにデータベースに任せられる機能は積

極的に利用すべきです。また、データベースの設計・維持関係者は、品質保証のために、23章「ディプロマティック・イミュニティ（外交特権）」を読むべきでしょう。

本書を通して、RDBの理論と意義を知って正当（正統）なRDB設計ができる人材、SQLらしいSQLを組めるプログラマーが増えることを願ってやみません。

最後に、もうひとりの監訳者、卓人へ。「違和感のない日本語になっている」、「原著がかなり実践的に書かれているためSQL文が多いが、それらSQL文を実際に検証している」、「原著後に状況が変わっている部分についてはその動きをフォローしている」、「日本人読者にとって分かり難い箇所に適切な注釈がつけられている」といった点に労を惜しまなかったことに感謝します。

はじめに

専門家とは、極めて狭い領域で、起こりうるすべてのミスを経験した人である。
ニールス・ボーア（デンマークの理論物理学者）

私は最初に頂いたSQL案件を、お断りしました。

カリフォルニア大学コンピューターサイエンス専攻を卒業して間もない頃です。在学中の学生活動でお世話になっていた同大学職員の方から仕事の打診を受けました。彼は大学での仕事のほかに、自らソフトウェア開発会社を立ち上げたばかりで、シェルスクリプトやawkといった関連ツールを用いて各種UNIXプラットホーム上で実行できるデータベースマネジメントシステムを開発していました（当時、動的プログラミング言語は、RubyやPython、PHPはおろかPerlでさえも現在のようには普及していませんでした）。彼は、SQL言語のうち、ある機能だけを解釈し実行できるコードを書ければいいと考えて私に依頼してきたのです。

「SQLの完全サポートはしなくていいと思っているんだ。それじゃ大変だからね。必要なSQLステートメントはたった1つ。SELECTだけだ」彼は言いました。

学校でSQLを教わったことはありません。データベースは現在ほど広く使用されておらず、MySQLやPostgreSQLなどのオープンソース・ソフトウェアもまだありませんでした。しかし、私にはシェルスクリプトでアプリケーションをいくつか開発した経験がありました。またパーサーについてもある程度の知識があり、コンパイラ設計や計算機言語学などの授業ではいくつかのプロジェクトにも携わりました。それでこの依頼を受けてみようと考えたわけです。SQLのような特殊言語のステートメント1つを解析するのが、どれぐらい難しいことかも知らずに……。

SQL言語リファレンスを参照したところ、これまで学んだことのあるif文、while文、変数の代入や式、そして自作関数などをサポートしている言語とSQLとは別物であると気づきました。SQLにおいてSELECTを1つのステートメントだと言ってしまうことは、エンジンが車を構成する一部品にすぎないと言っているようなものなのです。表現自体に誤りはありませんが、エンジンを真に理解するためには、その背後にある極めて複雑なメカニズムを理解しなければなりません。同じように、1つのSQLステートメントの実行をサポートするには、結局のところすべての機能を満たしたRDBMS（リレーショナルデータベースマネジメントシステム）とクエリエンジンを開発しなければならず、そのためのコードをすべて記述しなければ

ならないのではないか——私はそう考えました。

シェルスクリプトでSQLパーサーとRDBMSエンジンを開発できるよい機会ではありましたが、この仕事は断ることにしました。依頼者の彼は、プロジェクトの開発範囲を過小に見積もっていました。おそらく、RDBMSの仕組みをよく理解していなかったのでしょう。

その後、若き日の私は他の多くのソフトウェア開発者と同じように、手づる式にSQLを学んでいきました。開発者の多くは、他のプログラミング言語のように然るべきトレーニングを通じてではなく、担当しているプロジェクトで必要に迫られて、SQLを独学しています。大学でコンピューターサイエンスを専攻した人も例外ではありません。趣味でプログラミングをしている人であれ、職業プログラマーであれ、博士号を持っているコンピューターサイエンスの研究者であれ、SQLは訓練を受けることなく習得すべきソフトウェアスキルの1つと見なされているようです。

SQLについて詳しく学び始めるとすぐに、それがCやPascal、シェルスクリプトなどの手続き型プログラミング言語とも、C++、Java、Ruby、Pythonなどのオブジェクト指向言語とも大きく異なることを知り驚きました。SQLは、LISP、Haskell、XSLTなどと同じ「宣言型プログラミング言語」だったのです。オブジェクト指向言語ではオブジェクトを基本的なデータ構造として使用しますが、SQLでは、集合を基本的なデータ構造として使用します。従来のソフトウェア教育を受けた開発者は、この**インピーダンスミスマッチ（オブジェクトモデルとリレーショナルモデルのデータ構造の違い）**によりSQLへの学習意欲を削がれ、効果的なSQLの使用方法ではなく、オブジェクト指向のライブラリを学ぼうとするのです。

私は1992年以来、様々な形でSQLに携わってきました。アプリケーション開発でもSQLを使いましたし、PerlやPHPを用いたSQLプログラミング向けのライブラリも開発しました。InterBase（RDBMS製品）のテクニカルサポートや研修も担当しましたし、製品マニュアルの作成を行ったこともあります。インターネットのメーリングリストやニュースグループでは、数え切れないほど多くのSQL関連の質問に答えてきました。私はこれらの経験の中で、同じような「よくある質問」が何度も繰り返されていることに気づきました。そこには、ソフトウェア開発者が犯しがちなミスのパターンがあったのです。

対象読者

本書『SQL アンチパターン』の目的は、SQLを使う必要があるソフトウェア開発者が、SQLで効率的にソフトウェアを開発できるようにすることです。初心者から経験豊富なエキスパートまで、あらゆる段階の方に役立つ内容をお届けします。

本書の読者には、SQLの構文リファレンスを読んだという方もいるでしょう。それを見ればSELECTステートメントのすべての句が掲載されていますから、とりあえずの作業はできます。既存アプリケーションのソースコードや関連記事を読んだりすればSQLスキルは上がっていくでしょう。しかし良い例と悪い例を見分けるのは困難です。玉石混交の中からハズレを引かず、確実にベストプラクティスを学ぶにはどうすればよいのでしょうか。

本書で紹介されるトピックには、よく知られたものも含まれています。もしすでに解決策が分かると思っていたとしても、本書を読めば新たな気づきが得られるでしょう。プログラマーの間で広まっている思い違いを見直すことで、自分の方法が正しいかどうかを確認する良い機会となります。また、初めて知るトピックもあると思います。本書がSQLプログラマーとしての良い習慣づけに役立てば幸いです。

経験豊富なデータベース管理者なら、この本で記述されているSQLの落とし穴を避ける最善策をすでに知っているかもしれません。しかし本書を読むことで、データベース管理者はソフトウェア開発者がどのような考えでミスを犯してしまうのかがわかります。データベース管理者（DBA）と開発者が犬猿の仲であることは珍しくはありません。本書は、双方が互いに尊重し合い、チームワークを高めてより効果的に仕事を進めるために役立つはずです。本書を用いて、同僚のソフトウェア開発者にアンチパターンの適切な回避策を示し、それらがもたらす結果を伝えてあげてください。

対象範囲

アンチパターンとは何でしょうか。それは、問題の解決を意図しながらも、しばしば他の問題を生じさせてしまうような技法を指します。アンチパターンは実に様々な形で現れますが、これらには共通点があります。アンチパターンにつながる発想の源も、開発者が自分で思いついたもの、同僚からのアドバイス、書籍や雑誌・ウェブの記事など多岐にわたります。オブジェクト指向ソフトウェア設計のパターンや、プロジェクト管理で見られる数々のアンチパターンが掲載されている文書には、Portland Pattern Repository（http://c2.com/cgi-bin/wiki?Antipattern）や、1998年刊行の書籍『アンチパターン—ソフトウェア危篤患者の救出』[BMMM98]などがあります。

本書『SQLアンチパターン』には、SQLを使用するプログラマーが最も頻繁に犯しがちなミスを記述しています。これらのミスは、テクニカルサポートや研修、ソフトウェア開発、インターネットフォーラムでのやりとりを通じて私が実際に見てきたものです。もちろん、そのうちの多くは、私自身がしでかしてしまったものでもあります。最高の教師とは、自分が発生させてしまったエラーに対する、夜遅くまでの修正作業なのです。

本書の構成

本書は、4部構成になっています。アンチパターンを以下の4つのカテゴリーに分け、各パートでそれぞれを解説していきます。

データベース論理設計のアンチパターン

コードを書き始める前に、データベースに格納する情報を見極め、データの構成や関連付けを最善の方法で実現するための設計を行う必要があります。この作業には、データベースのテーブルや列、関連（リレーションシップ）の設計などがあります。

データベース物理設計のアンチパターン

格納するデータを決定した後には、使用するRDBMSの機能を最大限に活かして、データ管理の実装を行います。この作業には、テーブルやインデックスの定義、データ型の決定などがあります。定義にはSQLのCREATE TABLEステートメントのような**データ定義言語**（DDL: Data Definition Language）を使用します。

クエリのアンチパターン

データベースでは、データの追加や検索などを行います。SQLのクエリは、SELECT、UPDATE、DELETEステートメントなどの**データ操作言語**（DML: Data Manipulation Language）で構成されます。

アプリケーション開発のアンチパターン

SQLはたいてい、他の言語（C++、Java、PHP、Python、Rubyなど）で開発されたアプリケーションと共に使用されます。アプリケーションでSQLを用いる方法には、適切なものとそうでないものがあります。このパートでは、よくある失敗例を紹介します。

世の中のアンチパターンの多くには、「黄金のハンマー（Golden Hammer）」「車輪の再発明（Reinventing the Wheel）」「委員会による設計（Design by Committee）」などの、ユーモラスで刺激的なタイトルがついています[BMMM98]。優れたデザインパターンとアンチパターンには、比喩的で内容を連想しやすい名前を与えることが慣例となっています。

付録には、リレーショナルデータベース理論に関する実用的な情報を記載しています。本書で取り上げる多くのアンチパターンは、データベース理論を正しく理解していないことから生じるものです。

各章の構成——アンチパターンの「解剖」

各章では、以下の構成でアンチパターンを解説していきます。

目的

解決すべきタスクです。アンチパターンとは、タスクの解決のために用いられたものの、逆に多くの問題を生じさせてしまうものなのです。

アンチパターン

アンチパターンそのものを解説します。タスクの解決策としてよく用いられる方法が、その性質によって予期せぬ結果を招き、アンチパターンになってしまう経緯を説明します。

アンチパターンの見つけ方

プロジェクトでアンチパターンが使われているとき、それを察知するための手がかりがあります。直面している問題の種類や、メンバー間の会話での何気ない言葉が、そこにアンチパターンがあるかもしれないことに気づくヒントになります。

アンチパターンを用いてもよい場合

ルールには例外がつきものです。通常はアンチパターンであると思われるアプローチでも、状況によってはそれが適切である場合や、打てる手のなかでは一番ましなもの、という場合があります。

解決策

アンチパターンに陥いらないようにして、冒頭のタスクを解決するための望ましい方法を紹介します。

本書の対象範囲ではないもの

本書にはSQLの構文や用語に関する説明はありません。SQLの基本を学びたい方は、書籍やインターネットのリファレンスなどを豊富に利用できます。本書では、読者が基本的なSQL構文を学習済みであり、SQLを用いてある程度の仕事を行えることを前提とします。

パフォーマンス、スケーラビリティ、最適化は、データベース駆動のアプリケーション、特にウェブア

プリケーションの開発者にとって重要なテーマですが、これらは本書の対象外とします。パフォーマンスをテーマにしたデータベースプログラミング関連の書籍がいくつか書かれています。私のお薦めは『SQL Performance Tuning』[GP03]と『実践ハイパフォーマンス MySQL 第 2 版』[SZTZ08]です。本書でもトピックの一部にはパフォーマンスに触れていますが、これは本書の主題ではありません[†1]。

本書は、様々なベンダーが提供するデータベース製品やオープンソースデータベースのすべてに当てはまる問題と、その解決策を提供することを目指しています。SQL は、ANSI と ISO で標準化が行われており、各データベースはその標準に準拠しています。本書では、できる限り特定のベンダーに依存しない、中立な SQL の使用方法の提供につとめ、特定のベンダーに固有の SQL について記述するときは、それを明示するようにしています。

データアクセスフレームワークやオブジェクトリレーショナルマッピング（ORM）ライブラリは便利なツールですが、これらも本書の主題ではありません。サンプルコードのほとんどは、PHP で記述しています。読者が様々なプログラミング言語に置き換えながら読み進められるよう、サンプルコードはできる限り平易なものにすることを意識しました。

データベース管理や運用のタスク（サーバーのサイジング、インストールと設定、モニタリング、バックアップ、ログ解析、セキュリティ）はとても重要なテーマであり、詳しく説明しようとすれば、もう一冊本を書かなくてはなりません。本書の対象はデータベース管理者ではなく、SQL 言語を使用するデータベース開発者です。

本書のテーマは SQL とリレーショナルデータベースであり、他の代替技術（オブジェクト指向データベース、Key-Value ストア、カラム指向データベース、ドキュメント指向データベース、階層型データベース、ネットワークデータベース、MapReduce フレームワーク、セマンティックデータストアなど）ではありません。これらの代替ソリューションの長所と短所の比較や、適切な使用方法は興味深いテーマですが、その解説は他の本に譲ることにします。

表記ルール

この本の表記ルールについて説明します。

フォント

等幅フォント（Accounts、BugsProducts、account_name）

SQL のテーブル名には等幅フォントを使います。テーブル名を表す各単語の先頭の文字を大文字にし、間にはスペースを入れません。SQL の列名は小文字で記述し、列名を構成する各単語はアンダースコアでつなげます。

大文字の等幅フォント（SELECT）

SQL キーワードに使用します。

太字

強調や新出用語に使います。

†1 監訳注：日本語で読める書籍としては『データベースパフォーマンスアップの教科書 基本原理編』[Enc06]も良書として知られています。

用語

SQLの正式な発音は「エス・キュー・エル」です。口語的に「シークェル」と発音されることもあり、私もそれに異議を唱えるものではありませんが、本書では前者に従います。データベース関連の用語として用いられる場合、indexは順番に並べられた情報の集合を意味します。この場合のindexの望ましい複数形はindexesです。他の文脈では、indexはindicatorを意味することがあり、この場合の複数形はindicesです。ほとんどの辞書によれば、両方とも正しい語法とされており、書き手の間ではindexの表記に若干の混乱があります。本書では、indexの複数形をindexesと表記します。

SQLでは、**クエリ**と**ステートメント**は「実行可能な完全なSQLコマンド」を指すものとして、同義のものとして用いられることがあります。本書では、より明確な記述をするために、クエリはSELECTステートメントを、ステートメントは他のすべてのステートメント（INSERT、UPDATE、DELETE、およびデータ定義）を指すこととします。

ER図

リレーショナルデータベースを表す図として最も一般的なのは、**ER図（実体関連図）**です。テーブルは四角形、関連（リレーションシップ）は四角形を結びつける線で表されます。線の端にある記号は、関連の多重度を表します。図1に例を示します。

サンプルデータベース

本書のトピックの多くでは、架空のバグ管理アプリケーション用のデータベースを用いて説明を行います。このデータベースのER図を、図2に示します。Bugsテーブルと Accountsテーブルが3つの線でつながれ、3つの外部キーがあることに注目してください。

以下に、サンプルアプリケーションのテーブル定義で用いたデータ定義言語（DDL）のコードを示します。一部の定義は、本書で後述する説明のために行っているため、現実世界のアプリケーションでは一般的ではないものもあります。サンプルアプリケーションでは、各種データベース製品に適用できるよう、極力、標準的なSQLのみを使用することを意図しています。ただし、MySQLのデータ型の一部（SERIALやBIGINTなど）も用いています。

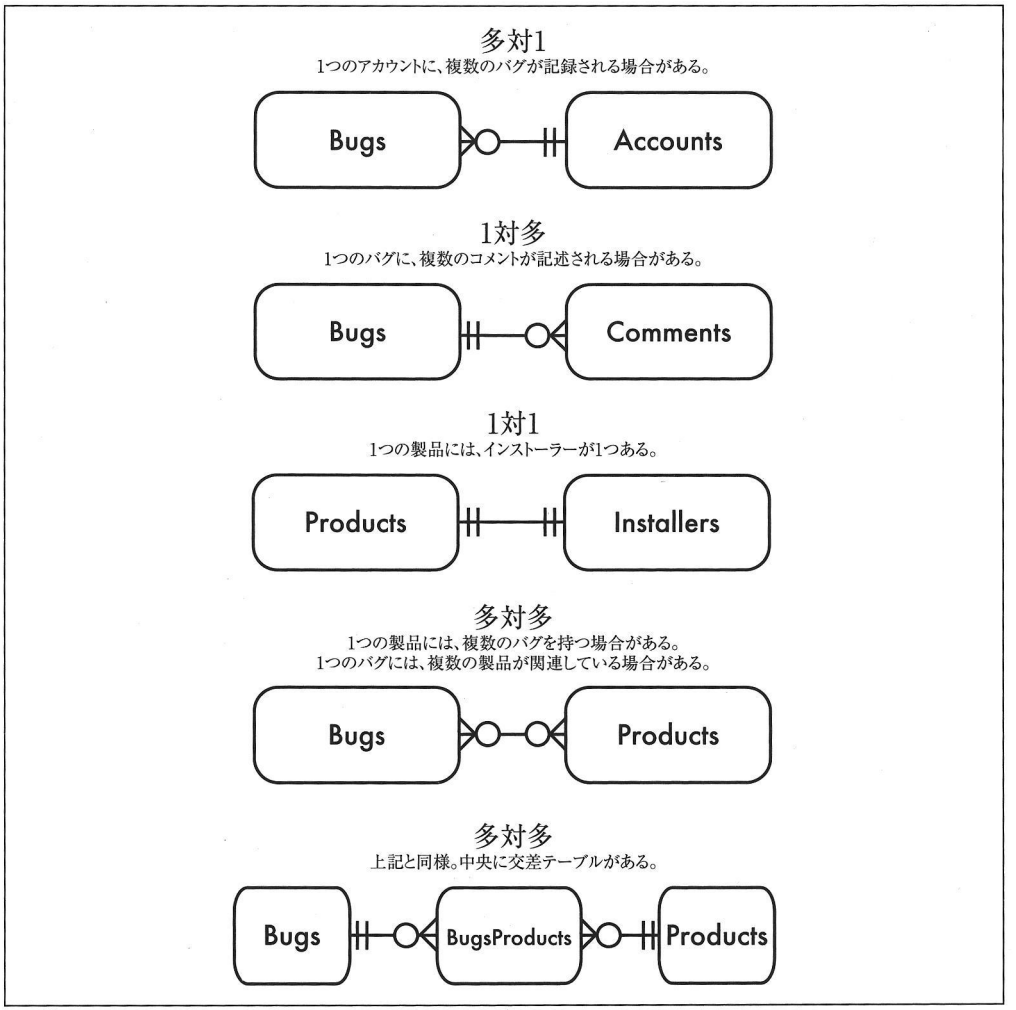


図 1　ER 図の例

Introduction/setup.sql

```
CREATE TABLE Accounts (
  account_id        SERIAL PRIMARY KEY,
  account_name      VARCHAR(20),
  first_name        VARCHAR(20),
  last_name         VARCHAR(20),
  email             VARCHAR(100),
  password_hash     CHAR(64),
  portrait_image    BLOB,
```

```
  hourly_rate       NUMERIC(9,2)
);

CREATE TABLE BugStatus (
  status            VARCHAR(20) PRIMARY KEY
);

CREATE TABLE Bugs (
  bug_id SERIAL     PRIMARY KEY,
  date_reported     DATE NOT NULL,
  summary           VARCHAR(80),
  description       VARCHAR(1000),
  resolution        VARCHAR(1000),
  reported_by       BIGINT UNSIGNED NOT NULL,
  assigned_to       BIGINT UNSIGNED,
  verified_by       BIGINT UNSIGNED,
  status            VARCHAR(20) NOT NULL DEFAULT 'NEW',
  priority          VARCHAR(20),
  hours             NUMERIC(9,2),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);

CREATE TABLE Comments (
  comment_id        SERIAL PRIMARY KEY,
  bug_id            BIGINT UNSIGNED NOT NULL,
  author            BIGINT UNSIGNED NOT NULL,
  comment_date      DATETIME NOT NULL,
  comment           TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          BIGINT UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

```
CREATE TABLE Tags (
  bug_id            BIGINT UNSIGNED NOT NULL,
  tag               VARCHAR(20) NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Products (
  product_id        SERIAL PRIMARY KEY,
  product_name      VARCHAR(50)
);

CREATE TABLE BugsProducts(
  bug_id            BIGINT UNSIGNED NOT NULL,
  product_id        BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

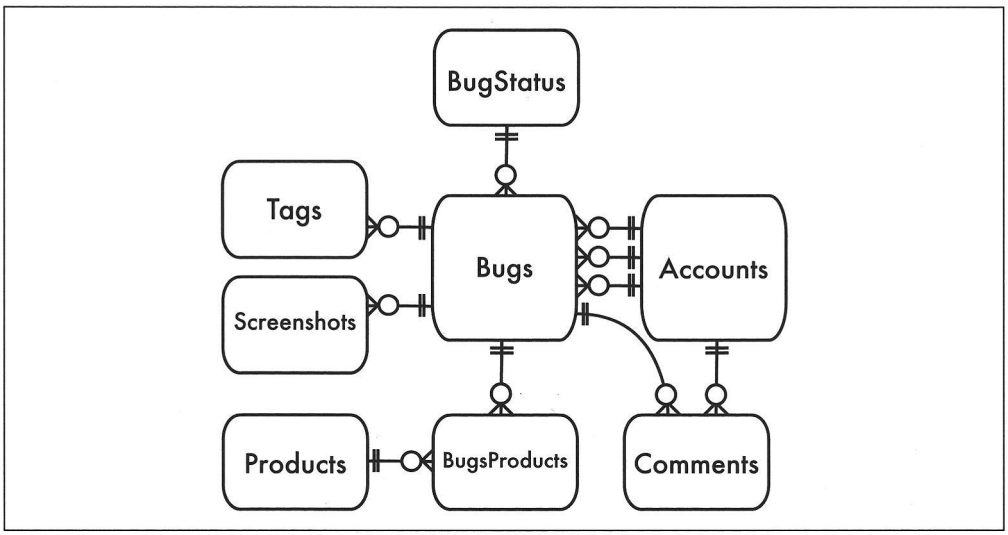


図2　サンプルのバグ管理データベースの図

ただし、いくつかの章、特にI部「データベース論理設計のアンチパターン」内の章では、特定のアンチパターンを示したり、アンチパターン回避のための代替の解決策を示すために、異なるテーブル定義を用いることがあります。

意見と質問

本書の内容については、最大限の努力をもって検証および確認していますが、誤りや不正確な点、誤解や混乱を招くような表現、単純な誤植に気づかれることもあるでしょう。本書を読んで気づいたことは、今後の版で改善できるように知らせていただければ幸いです。将来の改訂に関する提案なども歓迎です。

株式会社オライリー・ジャパン
〒160-0002 東京都新宿区坂町26番地27　インテリジェントプラザビル1F
電話 03-3356-5227
FAX 03-3356-5263
電子メール japan@oreilly.co.jp

本書のWebページには、正誤表、サンプルコードなどに関する追加情報を掲載しています。

http://www.oreilly.co.jp/books/9784873115894
http://pragprog.com/book/bksqla/sql-antipatterns（原著）

本書についての情報をTwitterでも提供しています。公式Twitterハッシュタグは、#sqlapです。
オライリーに関するその他の情報については、次のオライリーのWebサイトを参照してください。

http://www.oreilly.co.jp/
http://oreilly.com/

謝辞

何よりもまず、妻のJanに感謝を捧げます。いつも私を触発し、愛情とサポート（それから、ときどきの尻叩き）を与え続けてくれる彼女がいなければ、本書は執筆できませんでした。

同じく、多くの時間を割いてくれた、レビューアー各位にも感謝を。本書をより良い書籍にするうえで、レビューアーの皆様の意見が大きく役立ちました。Marcus Adams、Jeff Bean、Frederic Daoud、Darby Felton、Arjen Lentz、Andy Lester、Chris Levesque、Mike Naberezny、Liz Nealy、Daev Roehr、Marco Romanini、Maik Schmidt、Gale Straney、Danny Thorpeに感謝いたします。

本書の使命を信じ、執筆の機会を与えてくださった編集者のJacquelyn CarterとPragmatic Bookshelfの皆様にも御礼を申し上げます。

目　次

本書への称賛の声 v
監訳者まえがき ix
はじめに xv

I部　データベース論理設計のアンチパターン 1

1章　ジェイウォーク（信号無視） 3
1.1　目的：複数の値を持つ属性を格納する 4
1.2　アンチパターン：カンマ区切りフォーマットのリストを格納する 4
1.2.1　特定のアカウントに関連する製品の検索 5
1.2.2　特定の製品に関連するアカウントの検索 5
1.2.3　集約クエリの作成 5
1.2.4　特定の製品に関連するアカウントの更新 6
1.2.5　製品 ID の妥当性検証 7
1.2.6　区切り文字の選択 7
1.2.7　リストの長さの制限 7
1.3　アンチパターンの見つけ方 7
1.4　アンチパターンを用いてもよい場合 8
1.5　解決策：交差テーブルを作成する 8
1.5.1　特定のアカウントに関連する製品の検索／特定の製品に関連するアカウントの検索 9
1.5.2　集約クエリの作成 9
1.5.3　製品の連絡先の更新 10
1.5.4　製品 ID の妥当性検証 10
1.5.5　区切り文字の選択 11

1.5.6 リストの長さの制限 11
1.5.7 交差テーブルの他のメリット 11

2章 ナイーブツリー（素朴な木） 13
2.1 目的：階層構造を格納し、クエリを実行する 14
2.2 アンチパターン：常に親のみに依存する 14
2.2.1 隣接リストへのクエリ実行 15
2.2.2 隣接リストのツリーのメンテナンス 17
2.3 アンチパターンの見つけ方 18
2.4 アンチパターンを用いてもよい場合 18
2.5 解決策：代替ツリーモデルを使用する 20
2.5.1 経路列挙（Path Enumeration） 20
2.5.2 入れ子集合（Nested Set） 22
2.5.3 閉包テーブル（Closure Table） 26
2.5.4 どの設計を使うべきか 30

3章 IDリクワイアド（とりあえずID） 33
3.1 目的：主キーの規約を確立する 34
3.2 アンチパターン：すべてのテーブルに「id」列を用いる 36
3.2.1 冗長なキーが作成されてしまう 36
3.2.2 重複行を許可してしまう 37
3.2.3 キーの意味がわかりにくくなる 37
3.2.4 USINGを使用する 38
3.2.5 複合キーは使いにくい 39
3.3 アンチパターンの見つけ方 39
3.4 アンチパターンを用いてもよい場合 40
3.5 解決策：状況に応じて適切に調整する 40
3.5.1 わかりやすい列名にしよう 40
3.5.2 規約に縛られない 41
3.5.3 自然キーと複合キーの活用 41

4章 キーレスエントリ（外部キー嫌い） 43
4.1 目的：データベースのアーキテクチャを単純化する 44
4.2 アンチパターン：外部キー制約を使用しない 44
4.2.1 完ぺきなコードを前提にしている 44
4.2.2 ミスを調べなければならない 45
4.2.3 「私のミスではありません！」 46

4.2.4 「キャッチ=22」なUPDATE……46
4.3 アンチパターンの見つけ方……47
4.4 アンチパターンを用いてもよい場合……47
4.5 解決策：外部キー制約を宣言する……48
4.5.1 複数テーブルの変更をサポートする……48
4.5.2 オーバーヘッド、……にはなりません……49

5章 EAV（エンティティ・アトリビュート・バリュー）……51
5.1 目的：可変属性をサポートする……51
5.2 アンチパターン：汎用的な属性テーブルを使用する……53
5.2.1 属性を取得するにはどうするか……54
5.2.2 データ整合性をどう保つか……55
5.2.3 行を再構築しなければならない……57
5.3 アンチパターンの見つけ方……58
5.4 アンチパターンを用いてもよい場合……58
5.5 解決策：サブタイプのモデリングを行う……60
5.5.1 シングルテーブル継承……60
5.5.2 具象テーブル継承……61
5.5.3 クラステーブル継承……62
5.5.4 半構造化データ……64
5.5.5 後処理……64

6章 ポリモーフィック関連……67
6.1 目的：複数の親テーブルを参照する……68
6.2 アンチパターン：二重目的の外部キーを使用する……68
6.2.1 ポリモーフィック関連を定義する……69
6.2.2 ポリモーフィック関連へのクエリ実行……70
6.2.3 非オブジェクト指向の例……71
6.3 アンチパターンの見つけ方……72
6.4 アンチパターンを用いてもよい場合……73
6.4.1 ポリモーフィック関連を意識的に選択するとき……73
6.5 解決策：関連（リレーションシップ）を単純化する……74
6.5.1 参照を逆にする……74
6.5.2 交差テーブルの作成……74
6.5.3 交差点に交通信号を設置する……75
6.5.4 両方の「道」を見る……76
6.5.5 「道」を合流させる……76

6.5.6 共通の親テーブルの作成 78

7章 マルチカラムアトリビュート（複数列属性） 81
7.1 目的：複数の値を持つ属性を格納する 81
7.2 アンチパターン：複数の列を定義する 82
7.2.1 値の検索 82
7.2.2 値の追加と削除 83
7.2.3 一意性の保証 84
7.2.4 増加する値の処理 85
7.3 アンチパターンの見つけ方 85
7.4 アンチパターンを用いてもよい場合 86
7.5 解決策：従属テーブルを作成する 87

8章 メタデータトリブル（メタデータ大増殖） 89
8.1 目的：スケーラビリティを高める 90
8.2 アンチパターン：テーブルや列をコピーする 90
8.2.1 テーブルの増殖 91
8.2.2 データの整合性を管理する 92
8.2.3 データの同期 92
8.2.4 一意性の保証 93
8.2.5 テーブルをまたいだクエリ実行 93
8.2.6 メタデータの同期 94
8.2.7 参照整合性の管理 94
8.2.8 メタデータトリブル列の特定 95
8.3 アンチパターンの見つけ方 95
8.4 アンチパターンを用いてもよい場合 96
8.5 解決策：パーティショニングと正規化を行う 97
8.5.1 水平パーティショニングの使用 97
8.5.2 垂直パーティショニングの使用 97
8.5.3 従属テーブルの導入 99

Ⅱ部 データベース物理設計のアンチパターン 101

9章 ラウンディングエラー（丸め誤差） 103
9.1 目的：整数の代わりに小数値を使用する 104
9.2 アンチパターン：FLOAT データ型を使用する 104
9.2.1 丸めが避けられない 104

9.2.2 SQLでのFLOATの使用 105
9.3 アンチパターンの見つけ方 107
9.4 アンチパターンを用いてもよい場合 107
9.5 解決策：NUMERICデータ型を使用する 108

10章 サーティワンフレーバー（31のフレーバー） 111
10.1 目的：列を特定の値に限定する 111
10.2 アンチパターン：限定する値を列定義で指定する 112
10.2.1 中身は何だろう 113
10.2.2 新しいフレーバーの追加 114
10.2.3 昔ながらの味は色褪せない 115
10.2.4 移植が困難 115
10.3 アンチパターンの見つけ方 115
10.4 アンチパターンを用いてもよい場合 116
10.5 解決策：限定する値をデータで指定する 116
10.5.1 値セットの取得 117
10.5.2 参照テーブルの値の更新 117
10.5.3 廃止された値のサポート 117
10.5.4 移植が容易 118

11章 ファントムファイル（幻のファイル） 119
11.1 目的：画像をはじめとする大容量メディアファイルを格納する 120
11.2 アンチパターン：物理ファイルの使用を必須と思い込む 120
11.2.1 ファイルの削除時における問題 121
11.2.2 トランザクション分離の問題 121
11.2.3 ロールバック時における問題 122
11.2.4 データベースのバックアップツール使用時における問題 122
11.2.5 SQLアクセス権限使用時における問題 123
11.2.6 ファイルはSQLデータ型ではない 123
11.3 アンチパターンの見つけ方 124
11.4 アンチパターンを用いてもよい場合 124
11.4.1 常に2つの設計を検討する 125
11.5 解決策：必要に応じてBLOB型を採用する 125

12章 インデックスショットガン（闇雲インデックス） 129
12.1 目的：パフォーマンスを最適化する 130
12.2 アンチパターン：闇雲にインデックスを使用する 130

12.2.1 インデックスをまったく定義しない……130
12.2.2 インデックスを多く定義し過ぎる……131
12.2.3 インデックスが役立たないとき……133
12.3 アンチパターンの見つけ方……134
12.4 アンチパターンを用いてもよい場合……136
12.5 解決策：「MENTOR」の原則に基づいて効果的なインデックス管理を行う……136
12.5.1 Measure（測定）……137
12.5.2 Explain（解析）……137
12.5.3 Nominate（指名）……139
12.5.4 Test（テスト）……140
12.5.5 Optimize（最適化）……140
12.5.6 Rebuild（再構築）……140

Ⅲ部 クエリのアンチパターン……143

13章 フィア・オブ・ジ・アンノウン（恐怖の unknown）……145
13.1 目的：欠けている値を区別する……146
13.2 アンチパターン：NULL を一般値として使う、または一般値を NULL として使う……146
13.2.1 式で NULL を扱う……146
13.2.2 NULL を許容する列の検索……147
13.2.3 プリペアドステートメントで NULL を扱う……147
13.2.4 NULL の使用を避ける……148
13.3 アンチパターンの見つけ方……149
13.4 アンチパターンを用いてもよい場合……150
13.5 解決策：NULL を一意な値として使う……151
13.5.1 スカラー式での NULL……151
13.5.2 論理式での NULL……152
13.5.3 NULL の検索……152
13.5.4 列に NOT NULL 制約を宣言する……153
13.5.5 動的なデフォルト……154

14章 アンビギュアスグループ（曖昧なグループ）……155
14.1 目的：グループ内で最大値を持つ行を取得する……156
14.2 アンチパターン：非グループ化列を参照する……156
14.2.1 単一値の原則（Single-Value Rule）……156
14.2.2 SQL がクエリの意図を汲んでくれるとは限らない……157

14.3 アンチパターンの見つけ方 158
14.4 アンチパターンを用いてもよい場合 159
14.5 解決策：曖昧でない列を使用する 160
14.5.1 関数従属性のある列のみにクエリを実行する 160
14.5.2 相関サブクエリを使用する 160
14.5.3 導出テーブルを使用する 161
14.5.4 JOIN を使用する 162
14.5.5 他の列に対しても集約関数を使用する 163
14.5.6 グループごとにすべての値を連結する 163

15章 ランダムセレクション 165
15.1 目的：サンプル行をフェッチする 165
15.2 アンチパターン：データをランダムにソートする 166
15.3 アンチパターンの見つけ方 167
15.4 アンチパターンを用いてもよい場合 167
15.5 解決策：特定の順番に依存しない 168
15.5.1 1と最大値の間のランダムなキー値を選択する 168
15.5.2 欠番の穴の後にあるキー値を選択する 168
15.5.3 すべてのキー値のリストを受けとり、ランダムに1つを選択する 169
15.5.4 オフセットを用いてランダムに行を選択する 170
15.5.5 ベンダー依存の解決策 171

16章 プアマンズ・サーチエンジン（貧者のサーチエンジン） 173
16.1 目的：全文検索を行う 173
16.2 アンチパターン：パターンマッチ述語を使用する 174
16.3 アンチパターンの見つけ方 175
16.4 アンチパターンを用いてもよい場合 175
16.5 解決策：適切なツールを使用する 176
16.5.1 ベンダー拡張 176
16.5.2 MySQL のフルテキストインデックス 176
16.5.3 Oracle でのテキストインデックス 177
16.5.4 Microsoft SQL Server での全文検索 178
16.5.5 PostgreSQL でのテキスト検索 179
16.5.6 SQLite での全文検索（FTS） 180
16.5.7 サードパーティーのサーチエンジン 181

17章 スパゲッティクエリ 187
17.1 目的：SQLクエリの数を減らす 188
17.2 アンチパターン：複雑な問題をワンステップで解決しようとする 188
17.2.1 意図に反した結果 188
17.2.2 さらなる弊害 190
17.3 アンチパターンの見つけ方 190
17.4 アンチパターンを用いてもよい場合 191
17.5 解決策：分割統治を行う 191
17.5.1 ワンステップずつ 191
17.5.2 UNIONを用いる 192
17.5.3 CASE式とSUM関数を組み合わせる 193
17.5.4 上司の問題を解決する 194
17.5.5 SQLを用いたSQLの自動的な記述 195

18章 インプリシットカラム（暗黙の列） 197
18.1 目的：タイプ数を減らす 198
18.2 アンチパターン：ショートカットの罠に陥る 198
18.2.1 リファクタリングにおける問題 199
18.2.2 隠れた代償 200
18.2.3 求めなければ得られない 200
18.3 アンチパターンの見つけ方 200
18.4 アンチパターンを用いてもよい場合 201
18.5 解決策：列名を明示的に指定する 201
18.5.1 誤りの防止 202
18.5.2 それは多分、必要ない（YAGNI：You Ain't Gonna Need It） 202
18.5.3 ワイルドカードを使えない局面はいずれ訪れる 203

Ⅳ部 アプリケーション開発のアンチパターン 205

19章 リーダブルパスワード（読み取り可能パスワード） 207
19.1 目的：パスワードのリカバリーとリセットを行う 207
19.2 アンチパターン：パスワードを平文で格納する 208
19.2.1 パスワードの格納 208
19.2.2 パスワードの認証 209
19.2.3 パスワードを電子メールで送信する 209
19.3 アンチパターンの見つけ方 210
19.4 アンチパターンを用いてもよい場合 210

19.5 解決策：ソルトを付けてパスワードハッシュを格納する 211
19.5.1 ハッシュ関数を理解する 211
19.5.2 SQL でのハッシュの使用 212
19.5.3 ハッシュにソルトを加える 213
19.5.4 SQL からパスワードを隠す 215
19.5.5 パスワードをリカバリーするのではなく、リセットする 216

20章 SQL インジェクション 219
20.1 目的：動的 SQL を記述する 220
20.2 アンチパターン：未検証の入力をコードとして実行する 220
20.2.1 アクシデントは起きる 221
20.2.2 ウェブ最大のセキュリティ脅威 222
20.2.3 対処法の追求 222
20.3 アンチパターンの見つけ方 228
20.4 アンチパターンを用いてもよい場合 228
20.5 解決策：誰も信用してはならない 229
20.5.1 入力のフィルタリング 229
20.5.2 動的値のパラメータ化 230
20.5.3 動的値を引用符で囲む 230
20.5.4 ユーザーの入力をコードから隔離する 232
20.5.5 他の開発者にコードをレビューしてもらう 234

21章 シュードキー・ニートフリーク（疑似キー潔癖症） 235
21.1 目的：欠番を詰める 235
21.2 アンチパターン：隙間を埋める 236
21.2.1 欠番を割り当てる 236
21.2.2 既存行に番号を振り直す 237
21.2.3 データ不一致の元 238
21.3 アンチパターンの見つけ方 238
21.4 アンチパターンを用いてもよい場合 238
21.5 解決策：疑似キーの欠番は埋めない 239
21.5.1 行のナンバリング 239
21.5.2 GUID の使用 240
21.5.3 最も重要な問題 241

22章 シー・ノー・エビル（臭いものに蓋） 243
22.1 目的：簡潔なコードを書く 243

22.2 アンチパターン：肝心な部分を見逃す 244
22.2.1 診断せずに判断する 244
22.2.2 見逃しがちなコード 245
22.3 アンチパターンの見つけ方 246
22.4 アンチパターンを用いてもよい場合 247
22.5 解決策：エラーから優雅に回復する 247
22.5.1 リズムを維持する 247
22.5.2 ステップをたどり直す 248

23章 ディプロマティック・イミュニティ（外交特権） 251
23.1 目的：ベストプラクティスを採用する 252
23.2 アンチパターン：SQLを特別扱いする 252
23.3 アンチパターンの見つけ方 253
23.4 アンチパターンを用いてもよい場合 253
23.5 解決策：包括的に品質問題に取り組む 254
23.5.1 文書化 254
23.5.2 バージョン管理 256
23.5.3 テスティング 258
23.5.4 複数のブランチを扱う 261

24章 マジックビーンズ（魔法の豆） 263
24.1 目的：MVCのM（モデル）を単純化する 264
24.2 アンチパターン：モデルがアクティブレコードそのもの 265
24.2.1 アクティブレコードはモデルをデータベーススキーマに強く依存させてしまう 266
24.2.2 アクティブレコードはCRUD機能を公開してしまう 266
24.2.3 アクティブレコードはドメインモデル貧血症をもたらす 267
24.2.4 マジックビーンズのユニットテストは困難 269
24.3 アンチパターンの見つけ方 271
24.4 アンチパターンを用いてもよい場合 271
24.5 解決策：モデルがアクティブレコードを「持つ」ようにする 272
24.5.1 モデルを理解する 272
24.5.2 ドメインモデルの使用 273
24.5.3 プレーンなオブジェクトのテスト 276
24.5.4 現実的に考える 277

25章 砂の城 279
25.1 目的：サービスの安定稼働 279
25.2 アンチパターン：想定不足 280
25.3 アンチパターンの見つけ方 280
25.4 アンチパターンを用いてもよい場合 281
25.5 解決策 281
25.5.1 ベンチマーク 281
25.5.2 テスト環境の構築 282
25.5.3 例外処理 283
25.5.4 バックアップ 283
25.5.5 高可用性 283
25.5.6 ディザスタリカバリ 284
25.5.7 運用ポリシーの策定 284

V部 付録 287

付録A 正規化のルール 289
A.1 リレーショナルとは何か 289
A.1.1 行に上下の順番がない 290
A.1.2 列に左右の順番がない 290
A.1.3 重複行を許可しない 290
A.1.4 すべての列は1つの型を持ち、各行に1つの値を持つ 290
A.1.5 行に隠されたコンポーネントがない 291
A.2 正規化の神話 291
A.3 正規化とは何か 292
A.3.1 第1正規形 293
A.3.2 第2正規形 293
A.3.3 第3正規形 295
A.3.4 ボイスコッド正規形 296
A.3.5 第4正規形 297
A.3.6 第5正規形 299
A.3.7 他の正規形 301
A.4 正規化は常識的なもの 301

付録B 参考文献 303

索引 306

I部
データベース論理設計のアンチパターン

1章
ジェイウォーク（信号無視）

昔、とある Netscape のエンジニアが、JavaScript にポインターを渡し、それを文字列として格納して、C に渡した。30 人くらいが事故死してもおかしくないほどの大惨事になった。
——ブレイク・ロス（Mozilla Firefox の開発で知られるソフトウェアエンジニア）

あなたは、バグ管理アプリケーションの新機能を開発しています。製品の連絡窓口として、ユーザーを1人指定する機能です。最初の仕様では、各製品の連絡先となるユーザーは1人のみでした。ところが案の定、後になって1つの製品に複数ユーザーを連絡先として登録できるようにして欲しいという要望が出てきました。

初めは、その変更は簡単だと思いました。列にアカウントIDを1件のみ格納するのではなく、カンマ区切りで複数のアカウントIDのリストを格納できるようにすればよいと考え、その通りに変更を行いました。

しかし数日後、上司が困り顔で近づいて来ました。

「うちの開発部門では、プロジェクトに関連会社の社員も追加するようになった。だが、システムには1つのプロジェクトで5人までしか登録できないらしい。それ以上追加しようとするとエラーになるというクレームが来ている。いったい、どうなっているんだ？」

「そうです。プロジェクトに登録できる人数には限りがあります」あなたはうなずきながら、それがさも当然であるかのように答えます。

上司がさらに細かい説明を求めていることを感じ取り、あなたはこう言います。

「登録できるのは、5人から10人くらいです。それは、ユーザーがアカウントをつくった時期によって変わります」

上司は眉をしかめました。あなたは続けます。

「アカウントIDは、カンマ区切りのリストで格納しています。しかしこのリストは定義された長さの文字列に収まる必要があります。つまり、IDの長さが短ければ、リストに加えられる数が増えます。初期にIDを作った人は99以下のIDが割り振られているので、文字数が少なくなるのです」

上司はまた眉をひそめました。今日は帰りが遅くなるだろうという予感がしました。

開発者はよく、「多対多」の関連を表現する**交差テーブル**の作成を避けるために、カンマ区切りのリストを使います。私はこのアンチパターンを**ジェイウォーク（信号無視）**と名づけました。どちらも、

"intersection"（交差点 / 交差テーブル）を避けようとする行為だからです[†1]。

1.1 目的：複数の値を持つ属性を格納する

列に値を1つのみ格納する場合、設計は簡単です。例えば整数、日付、文字列などに対応したSQLデータ型を選択し、値がその型の1つのインスタンスであることを表現します。しかし、関連する値の集合を列に収める場合には、どうすればよいのでしょうか。

サンプルのバグ管理データベースでは、当初はProductsテーブルの整数型の列を用いて、製品に連絡先を関連付けることでしょう。あるアカウントは複数の製品を持つ場合があり、ある製品はただ1つの連絡先を参照しています。つまり、製品とアカウントの間には「多対1」の関連があります[†2]。

Jaywalking/obj/create.sql

```
CREATE TABLE Products (
  product_id   SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id   BIGINT UNSIGNED,
  -- 他の列. . .
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', 12);
```

プロジェクトが進むにつれて、製品が複数の連絡先を持つ場合があることがわかってきました。多対1関連だけでなく、製品からアカウントに対する1対多の関連をサポートする必要があります。Productsテーブルの1つの行が、複数の連絡先を持てるようにしなければなりません。

1.2 アンチパターン：カンマ区切りフォーマットのリストを格納する

データベースの構造に対する変更を最小限に抑えるために、あなたはaccount_id列を VARCHAR型の列として再定義し、複数アカウントのIDをカンマ区切りで連結して格納できるようにしました。

Jaywalking/anti/create.sql

```
CREATE TABLE Products (
  product_id   SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id   VARCHAR(100), -- カンマ区切りのリスト
  -- 他の列. . .
```

†1 訳注：Jaywalkingとは、横断歩道のないところで道路を横断することや、信号無視をすることです。

†2 監訳注：この章の説明のために、「はじめに」のProductsテーブル定義とは若干異なっている（account_id列が加わっている）ことに注意してください。

```
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34');
```

テーブルや列を新たに追加しなくても済んだので、うまくいっているように思えます。変更したのは、1つの列のデータ型のみです。しかし実際には、このテーブル設計の変更が原因で、パフォーマンス上の問題やデータ整合性の問題が生じてしまうのです。

1.2.1 特定のアカウントに関連する製品の検索

すべての外部キーが文字列連結されて1つのフィールドに格納されていると、クエリを作ることが難しくなります。等価性による比較ができなくなるため、何らかの文字列パターンに対する**パターンマッチ**が必要になります。例えばMySQLでは、account_idが12のアカウントが指定されたすべての製品を取得するためには、以下のように記述します。

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products WHERE account_id REGEXP '[[:<:]]12[[:>:]]';
```

パターンマッチは、式の書き方によっては意図しない一致結果が返される可能性があり、**インデックス**を使うメリットも得られません。またパターンマッチ構文は各種データベース製品において異なるため、SQLはベンダー中立のものにはなりません。

1.2.2 特定の製品に関連するアカウントの検索

同じく、カンマ区切りのリストの値を使って参照先のテーブルと**結合**（JOIN）するのは手間がかかり、効率的なものにもなりません[†3]。

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products AS p INNER JOIN Accounts AS a
    ON p.account_id REGEXP '[[:<:]]' || a.account_id || '[[:>:]]'
WHERE p.product_id = 123;
```

このような式で2つのテーブルを結合すると、インデックスを使う機会が失われます。このクエリは、両方のテーブルをすべてスキャンして、クロス積を生成し、すべての行の組み合わせについて正規表現を評価します。

1.2.3 集約クエリの作成

集約クエリでは、COUNT、SUM、AVGなどの関数を使用します。しかし、これらの関数はカンマ区切りリストではなく、複数の行に対して使われるように設計されています。このため、以下のような細工を施さなく

†3 監訳注：MySQLで文字列連結を行うには、文字列連結演算子||の代わりにCONCAT関数を使うか、または「set @@session.sql_mode='PIPES_AS_CONCAT';」を実行する必要があります。

てはならなくなります。

Jaywalking/anti/count.sql

```
SELECT product_id, LENGTH(account_id) - LENGTH(REPLACE(account_id, ',', '')) + 1
    AS contacts_per_product
FROM Products;
```

このようなトリックは巧妙なものかもしれませんが、決して単純明快とは言えません。開発に時間がかかり、デバッグも難しくなります。また一部の集約クエリには、こうした方法では実現できないものがあります。

1.2.4 特定の製品に関連するアカウントの更新

文字列連結によって、リストの末尾に新しいIDを追加できます。しかしこの方法では、リストのソート順を維持できなくなる場合があります。

Jaywalking/anti/update.sql

```
UPDATE Products
SET account_id = account_id || ',' || 56
WHERE product_id = 123;
```

IDをリストから削除するには、SQLを2つ発行しなければなりません。古いリストの取得と、更新したリストの保存です。以下に例を示します。

Jaywalking/anti/remove.php

```
<?php
$stmt = $pdo->query(
  "SELECT account_id FROM Products WHERE product_id = 123");
$row = $stmt->fetch();
$contact_list = $row['account_id'];

// PHPコードでのlistの変更
$value_to_remove = "34";
$contact_list = split(",", $contact_list);
$key_to_remove = array_search($value_to_remove, $contact_list);
unset($contact_list[$key_to_remove]);
$contact_list = join(",", $contact_list);

$stmt = $pdo->prepare(
  "UPDATE Products SET account_id = ?
   WHERE product_id = 123");
$stmt->execute(array($contact_list));
```

リストから要素を1つ削除するために、これだけ長いコードが必要になってしまうのです。

1.2.5 製品IDの妥当性検証

ユーザーが"banana"のような無効な入力を行ってしまうことを、どうやったら防げるでしょうか。

Jaywalking/anti/banana.sql

```
INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34,banana');
```

ユーザーは様々な値を入力できるので、データベースはゴミだらけになります。必ずしもデータベースエラーにはなりませんが、データは意味不明なものになってしまうでしょう。

1.2.6 区切り文字の選択

整数値のリストではなく文字列値のリストを格納する場合、リストの個々の入力値にも区切り文字（カンマなど）が含まれることがあります。このため、カンマを区切り文字に指定している場合には、入力された文字列の一部としてカンマが使われているのか、各要素を区別するための区切り文字として使われているのかが不明確になる可能性があります。他の区切り文字を指定することもできますが、その区切り文字が入力文字列として使用されないという保証はできません。

1.2.7 リストの長さの制限

データ型VARCHAR(30)の列に格納できるエントリの数は、各エントリの長さによって変わります。エントリの長さが2文字の場合、10個のエントリをリストに格納できます（カンマを含めて1エントリ3文字）。しかしエントリの長さが6文字の場合、リストに格納できるのは4個のみです。

Jaywalking/anti/length.sql

```
UPDATE Products SET account_id = '10,14,18,22,26,30,34,38,42,46'
WHERE product_id = 123;

UPDATE Products SET account_id = '101418,222630,343842,467790'
WHERE product_id = 123;
```

また、将来必要となるかもしれないリストの長さにVARCHAR(30)が対応できるかどうかを、どうやって判断すればよいのでしょうか。上司や顧客に対し、どのようにその根拠を説明できるでしょうか。

1.3 アンチパターンの見つけ方

チーム内で以下のような会話が交わされていたら、注意してください。「**ジェイウォーク（信号無視）**」アンチパターンが発生している可能性があります。

「このリストでサポートしなくてはならない最大のエントリ数は？」

これは、VARCHAR列の最大長を決定しようとしているときに生じる疑問です。

「SQL で単語境界を一致させる方法を知ってる？」

正規表現で文字列の一部を取得しようとしているならば、それらを別個に格納した方が良いという予兆かもしれません。

「リストのエントリに絶対使われない文字って何だっけ？」

エントリの文字列には絶対に使われない、確実に区別できる区切り文字を探しているときに出てくる言葉です。しかし、どのような文字でも、いつかは入力される可能性があることを肝に銘じておくべきです。

1.4 アンチパターンを用いてもよい場合

ある種のクエリのパフォーマンス向上のために、データベース構造に**非正規化**（denormalization）を適用する場合があります。リストをカンマ区切りの文字列として格納することも、非正規化の一例です。

アプリケーションによっては、カンマ区切りフォーマットのデータが必要で、かつリスト内の各要素への個別アクセスが不要な場合があります。カンマ区切りを用いていれば、他の場所からカンマ区切り形式のデータを受け取った場合に、そのリストをそのまま列に格納でき、その後もそのまま取得できます。値をバラバラにする必要はありません。

ただし、非正規化は、十分に検討してから採用しなければなりません。まずは正規化されたデータベース構造の採用を検討しましょう。正規化された構造のほうが、アプリケーションコードの柔軟性を高め、データの整合性を保つ助けにもなるからです。

1.5 解決策：交差テーブルを作成する

account_idを Productsテーブルに格納するのではなく、新たに作成したテーブルの各行に account_idを１つずつ格納します。新たに作成したこの Contactsテーブルによって、 Productsと Accountsの間には「多対多」の関連が生じます。

Jaywalking/soln/create.sql

```
CREATE TABLE Contacts (
  product_id  BIGINT UNSIGNED NOT NULL,
  account_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (product_id, account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Contacts (product_id, account_id)
VALUES (123, 12), (123, 34), (345, 23), (567, 12), (567, 34);
```

テーブルが2つのテーブルを参照する外部キーを持つとき、そのテーブルは**交差（インターセクション：intersection）テーブル**と呼ばれます[†4]。交差テーブルは、2つの参照先のテーブルに多対多の関連をもたらします。つまり、製品は複数のアカウントに、アカウントは複数の製品に結びつけられます。図1-1に、ER図を示します。

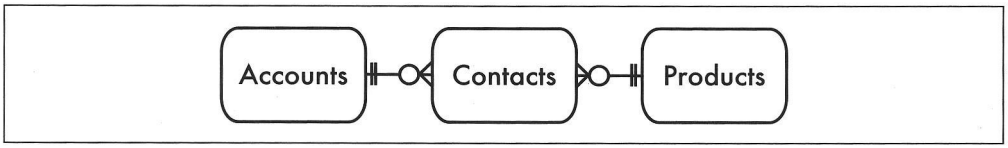


図1-1 交差テーブルのER図

では、交差テーブルを使うことで、「アンチパターン」の節で見た問題をどう解決できるかを見ていきましょう。

1.5.1 特定のアカウントに関連する製品の検索／特定の製品に関連するアカウントの検索

特定のアカウントに関連するすべての製品の検索は、Productsテーブルと Contactsテーブルを結合（JOIN）することで、より簡単に行うことができます。

Jaywalking/soln/join.sql

```
SELECT p.*
FROM Products AS p INNER JOIN Contacts AS c ON p.product_id = c.product_id
WHERE c.account_id = 34;
```

クエリのパフォーマンス低下を懸念して、結合を使うことに抵抗を示す人もいます。しかし、このクエリでは「アンチパターン」の節で示した解決策よりも、はるかに効果的にインデックスを使えます。

また、アカウントの詳細の検索クエリもわかりやすく、最適化も容易になります。このクエリは結合の際にインデックスを効果的に使うので、複雑な正規表現はもう必要ありません。

Jaywalking/soln/join.sql

```
SELECT a.*
FROM Accounts AS a INNER JOIN Contacts AS c ON a.account_id = c.account_id
WHERE c.product_id = 123;
```

1.5.2 集約クエリの作成

次の例は、製品ごとのアカウントの数を返します。

†4 交差テーブルは、関連テーブル、結合テーブル、多対多テーブル、マッピングテーブルなどと呼ばれることもありますが、これらが指しているコンセプトは同じです。

Jaywalking/soln/group.sql

```
SELECT product_id, COUNT(*) AS accounts_per_product
FROM Contacts
GROUP BY product_id;
```

アカウントごとの製品数も、同じぐらい単純です。

Jaywalking/soln/group.sql

```
SELECT account_id, COUNT(*) AS products_per_account
FROM Contacts
GROUP BY account_id;
```

他にも、より洗練された方法が使えます。例えば、最も関連アカウント数の多い製品などを取得できます。

Jaywalking/soln/group.sql

```
SELECT c.product_id, c.accounts_per_product
FROM (
  SELECT product_id, COUNT(*) AS accounts_per_product
  FROM Contacts
  GROUP BY product_id
) AS c
HAVING c.accounts_per_product = MAX(c.accounts_per_product)
```

1.5.3 製品の連絡先の更新

リストへの要素の追加や削除は、交差テーブルへの行の挿入や削除によって行えます。各製品への参照はContactsテーブルの個別の行に格納されているため、製品ごとに個別に追加・削除が行えるのです。

Jaywalking/soln/remove.sql

```
INSERT INTO Contacts (product_id, account_id) VALUES (456, 34);
DELETE FROM Contacts WHERE product_id = 456 AND account_id = 34;
```

1.5.4 製品IDの妥当性検証

外部キーを用いて別テーブルの値を参照することで、入力の妥当性検証を行えます。Contacts.account_idからAccounts.account_idの参照を宣言して、データベースに**参照整合性**を保証させることができます。**外部キー制約**によって、交差テーブルにはAccountsテーブルに存在するアカウントIDのみが格納されるようになります。

また、SQLのデータ型によって入力内容を制限できます。例えば、リストの各要素の有効値を整数値や日付にしたい場合、列に対してINTEGERや DATEといったデータ型を宣言することで、すべてのエントリが

これらのデータ型の値であることを保証できます。これで、bananaのようなナンセンスなエントリを受け付けないようにできます。

1.5.5 区切り文字の選択

各エントリを個別の行に格納するので、区切り文字は不要です。もう入力内容にカンマ区切りと同じ文字が含まれているかどうかを心配する必要はありません。

1.5.6 リストの長さの制限

各エントリは交差テーブルの個別の行に格納されているので、残っている制限があるとすれば、1つのテーブルに物理的に格納できる行数のみです。関連するエントリ数の上限を設けたい場合には、リストの長さではなく、エントリ数のCOUNTを制限の基準にするというポリシーで、アプリケーション側で制御するようにしましょう。

1.5.7 交差テーブルの他のメリット

Contacts.account_id上のインデックスを用いることで、カンマ区切りリストの部分文字列とマッチさせる方法よりも、パフォーマンスが向上します。多くのデータベース製品では、列を外部キーとして宣言することで、その列に暗黙のうちにインデックスが作成されます（ただし、念のため各データベース製品のドキュメントを確認してください）。

交差テーブルにさらに列を加えることによって、各エントリに属性を追加できます。例えば、製品に連絡先が追加された日付を記録したり、連絡先の優先順位を示す属性をつけたりできます。これは、カンマ区切りリストでは実現できないことです。

ひとつひとつの値は個別の行と列に格納しましょう

2章
ナイーブツリー（素朴な木）

森林保全？ 木なんてどれも同じじゃないか―― 1本あれば十分だ
――ロナルド・レーガン（第40代アメリカ合衆国大統領）

あなたは、科学技術ニュースを掲載する、ある人気サイト向けにソフトウェアを開発しています。

それは今風のウェブサイトで、各記事には読者がコメントを書き込めるほか、読者同士でのディスカッションも可能です。コメントはスレッド形式で表示され、スレッドはディスカッションのトピックに応じて枝分かれします。開発者のあなたは、枝分かれしたコメントスレッドを扱わなければなりません。そのための方法としてまず浮かんだのは、各コメントにリプライの対象となっている親コメントを参照させるというシンプルな案でした。

Trees/intro/parent.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id)
);
```

しかし、すぐに問題にぶつかりました。リプライの連鎖が長い場合、単一のSQLクエリで取得を行うことが難しくなるのです。可能なのは、直近の子か、固定した深さまでの子孫の取得のみです。しかし、スレッドの深さには制限がありません。あるスレッドのすべてのコメントを取得するには、大量のSQLクエリが必要になってしまいます。

そこで、あなたはもう1つ別の案をひねり出しました。すべてのコメントを取得し、学校で教わった伝統的なツリーアルゴリズムを用いて、それらをアプリケーションメモリ上でツリー型のデータ構造として構築するというものです。しかしこのニュースサイトでは、毎日いくつもの記事がリリースされており、各記事には多ければ数百ものコメントが書き込まれます。誰かがウェブサイトを見るたびに毎回大量のコメントをソートするのは、現実的ではありません。

コメントのスレッドすべてを格納し、シンプルかつ効果的に特定のスレッド全体を取得する良い方法はないのでしょうか。

2.1 目的：階層構造を格納し、クエリを実行する

再帰的な関連を持つデータは珍しくはありません。そのようなデータは、ツリー状の構造や階層的な構造で組織化されます。ツリー構造では、各エントリは**ノード**と呼ばれます。ノードは1つ以上の子と、1つの親を持てます。親を持たない最上位のノードは**根（ルート）**、子を持たない最下部のノードは**葉（リーフ）**、中間のノードは**非葉ノード**と呼ばれます。

前述の階層型データでは、個別のエントリ、部分集合、集合全体を取得するクエリが必要になると考えられます。ツリー指向のデータ構造の代表例として、以下の2つが挙げられます。

組織図

従業員の上下関係を表す組織図はツリー構造の代表例で、SQL関連の書籍や記事にも頻繁に登場します。組織図では、上司と部下の関係が、ツリー構造での親と子の関係に相当します。ある部下にとっての上司も、そのまた上司から見れば部下になります。

スレッド形式のコメント欄（掲示板）

この章の冒頭で見たように、あるコメントに対してリプライコメントがぶらさがっていく形式のデータを表すために、ツリー構造が用いられることがあります。ツリー構造では、ノード（コメント）の子はそのリプライです。

この章では、スレッド形式のコメント欄の例を用いて、アンチパターンとその解決策を見ていくことにします。

2.2 アンチパターン：常に親のみに依存する

前述の問題に対する解決策として多くの書籍や記事で推奨されているのは、parent_id列を加えるという単純な方法です。しかし、この方法は思慮が浅い、素朴（ナイーブ）な解決策であるとも言えるでしょう。parent_id列は、同じテーブルにある別の行（このサンプルでは comment_id）を参照します。この関連付けは、外部キー制約によって強制できます。このテーブルを定義するDDLを以下に示し、図2-1にER図を示します。

Trees/anti/adjacency-list.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id),
```

```
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

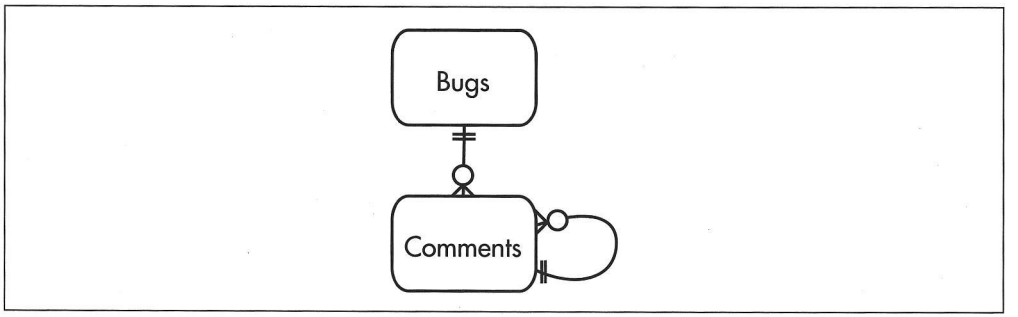


図 2-1　隣接リストの ER 図

このような設計は、**隣接リスト**（Adjacency List）と呼ばれます。おそらく、階層的なデータの格納に用いられる最も一般的な設計でしょう。表 2-1は、コメントを階層構造で表すためのサンプルデータです。図 2-2でツリーを図解します。

表 2-1　隣接リストのデータ

comment_id	parent_id	発言者	コメント
1	NULL	Fran	このバグの原因は何かな？
2	1	Ollie	ヌルポインターのせいじゃないかな？
3	2	Fran	そうじゃないよ。それは確認済みだ。
4	1	Kukla	無効な入力を調べてみたら？
5	4	Ollie	そうか、バグの原因はそれだな。
6	4	Fran	よし、じゃあチェック機能を追加してもらえるかな？
7	6	Kukla	了解。修正したよ。

2.2.1　隣接リストへのクエリ実行

多くの開発者にとってデフォルトの選択肢であるにもかかわらず、隣接リストは、ツリーに関する最も一般的な処理を行うときにアンチパターンになる可能性があります。それは、すべての子孫を取得するクエリです。

コメントとその直近の子は、比較的単純なクエリで取得できます。

Trees/anti/parent.sql

```
SELECT c1.*, c2.*
FROM Comments c1 LEFT OUTER JOIN Comments c2
  ON c2.parent_id = c1.comment_id;
```

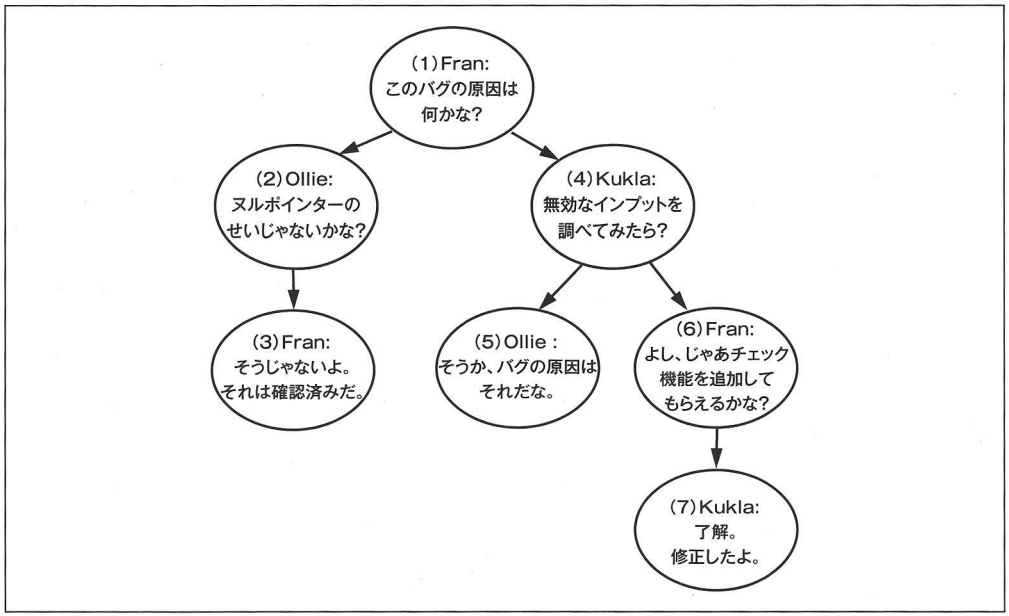


図2-2 ツリー構造で表したスレッド形式のコメント

しかし、このクエリが対象にできるのは、ツリー中の2つの階層のみです。ツリーはその性質上、深さに制限が無い場合が多いため、階層の深さにかかわらず子孫を取得するクエリを実行できなくてはなりません。例えば、ノードの数（例：スレッドのコメント数）をCOUNTで数えたり、ノードの値（例：製造業の各部品のコスト）をSUMで合計する場合などには、すべての子孫を対象にする必要があります。

隣接リストでは、この種のクエリをうまく使えません。隣接リストではツリーの1つの階層が1つのJOINに対応していますが、SQLを書く際にはJOINの数を固定しなくてはならないからです。以下のクエリでは、4階層までのコメントしか取得できません。

Trees/anti/ancestors.sql

```
SELECT c1.*, c2.*, c3.*, c4.*
FROM Comments c1 -- 1階層目
  LEFT OUTER JOIN Comments c2
    ON c2.parent_id = c1.comment_id -- 2階層目
  LEFT OUTER JOIN Comments c3
    ON c3.parent_id = c2.comment_id -- 3階層目
  LEFT OUTER JOIN Comments c4
    ON c4.parent_id = c3.comment_id; -- 4階層目
```

このクエリがさらに不格好なのは、より深い階層の子孫を含めれば含めるほど、結果に列も加わってしま

うことです。これによりCOUNTのような集約関数の扱いが難しくなります。

隣接リストを使っているツリー構造に対してクエリを実行するには、集合のすべての行を取得し、アプリケーションの側で階層構造を再構築してから、ツリーのように使用する方法もあります。

Trees/anti/all-comments.sql

```
SELECT * FROM Comments WHERE bug_id = 1234;
```

しかし、操作のたびにデータベースからアプリケーションへ大量のデータをコピーするのは、極めて非効率です。必要なのはツリー全体ではなくサブツリーのみの場合もありますし、コメント数のCOUNTのようなデータの集約情報のみである場合もあるからです。

2.2.2 隣接リストのツリーのメンテナンス

隣接リストでは、葉ノードの追加のように、比較的簡単に行える操作もあります。

Trees/anti/insert.sql

```
INSERT INTO Comments (bug_id, parent_id, author, comment_date, comment)
  VALUES (1234, 7, 23, CURRENT_TIMESTAMP, 'ありがとう！');
```

ノードやサブツリーの移動も簡単です。

Trees/anti/update.sql

```
UPDATE Comments SET parent_id = 3 WHERE comment_id = 6;
```

ただし、ノードの削除は簡単ではありません。サブツリー全体を削除したい場合、まず、すべての子孫を特定するために複数回クエリを実行し、次に、外部キーの整合性制約を満たすために、最下層から順番に子孫を削除する必要があります。

Trees/anti/delete-subtree.sql

```
SELECT comment_id FROM Comments WHERE parent_id = 4; -- 5と6を返す
SELECT comment_id FROM Comments WHERE parent_id = 5; -- 何も返さない
SELECT comment_id FROM Comments WHERE parent_id = 6; -- 7を返す
SELECT comment_id FROM Comments WHERE parent_id = 7; -- 何も返さない
DELETE FROM Comments WHERE comment_id IN (7);
DELETE FROM Comments WHERE comment_id IN (5, 6);
DELETE FROM Comments WHERE comment_id = 4;
```

この操作は、外部キー定義時にON DELETE CASCADE修飾子を付けることによって自動化できます。ただしその自動化が有効なのは子孫を削除する場合のみです。ノードの昇格や移動は自動化できません。

非葉ノードを削除した際に、その子を昇格させたりツリー内の別の場所に移動する場合には、対象のノードの削除前に子の parent_id を変更しなければなりません。

Trees/anti/delete-non-leaf.sql

```
SELECT parent_id FROM Comments WHERE comment_id = 6; -- 4を返す
UPDATE Comments SET parent_id = 4 WHERE parent_id = 6;
DELETE FROM Comments WHERE comment_id = 6;
```

上に挙げたのは、隣接リスト設計を用いているときに複数のステップが必要になる操作の一部に過ぎません。隣接リストでは、データベースでは本来シンプルかつ効率的に行えるべきことでも、多くのコードが必要になってしまうのです。

2.3 アンチパターンの見つけ方

以下のような言葉を耳にしたら、「**ナイーブツリー（素朴な木）**」アンチパターンが使われている可能性があります。

「このツリーでは、深さを何階層までサポートすればいい？」

ノードのすべての子孫や先祖の取得に再帰的なクエリを使わずに苦心していると考えられます。深さに制限を設けるという妥協案もありますが、そうなると今度は深さをいくつに設定すればよいのかという頭の痛くなる問題に直面します。

「ツリー型のデータ構造を扱うコードなんて二度と書きたくないな」

階層構造を扱うための洗練された解決策の1つを採用したものの、目下の作業がうまく処理できていないことが考えられます。ある技術を採用すれば、楽になるタスクはあります。しかしそれは、他のタスクを難しくするという犠牲の上に成り立つ場合が多いのです。現在の解決策は目の前のタスクには不向きだと認識し、別の選択肢を探るべきです。

「ツリーの中で孤児になった行をきれいにするために、定期的にスクリプトを実行しなければ」

非葉ノードを削除した際に、ツリーから切り離されたノードが発生してしまっています。複雑なデータ構造をデータベースに格納する場合には、変更のたびに構造を整合性の取れた状態に保たなくてはいけません。この章で後述する解決策をトリガー及びカスケード外部キー制約と併用することで、壊れにくく回復力のある方法でデータ構造を格納できるようになります。

2.4 アンチパターンを用いてもよい場合

隣接リスト設計は、アプリケーションで求められているタスクに適している限りは有効です。隣接リストの長所は、ノードの直近の親と子を簡単に取得できることです。また、列の挿入も容易です。階層構造を持つデータに対して必要な操作がこれらのみである場合、隣接リストは効果的に機能します。

また、隣接リストに格納された階層構造をサポートするSQL拡張機能を備えているデータベース製品もあります。SQL-99 標準では、WITH キーワードの後に**共通テーブル式**（Common Table Expression：

CTE）を指定する形式の**再帰クエリ**構文を定義しています。

Trees/legit/cte.sql

```
WITH CommentTree
    (comment_id, bug_id, parent_id, author, comment, depth)
AS (
    SELECT *, 0 AS depth FROM Comments
    WHERE parent_id IS NULL
  UNION ALL
    SELECT c.*, ct.depth+1 AS depth FROM CommentTree ct
    JOIN Comments c ON ct.comment_id = c.parent_id
)
SELECT * FROM CommentTree WHERE bug_id = 1234;
```

共通テーブル式を用いた再帰クエリは、Microsoft SQL Server 2005、Oracle Database 11g、IBM DB2、PostgreSQL8.4 でサポートされています。

ただし MySQL、SQLite、Informix などでは、この構文はまだサポートされていません。現在でも多く使用されている Oracle Database 10g も同様です。将来的には、主な RDBMS のほぼすべてで再帰クエリ構文がサポートされるようになるかもしれません。そうなれば、隣接リスト設計も一般的に使える選択肢になるでしょう。

Oracle9i Database と Oracle Database 10g は、WITH 句をサポートしています。ただし、再帰クエリをサポートしておらず、代わりに独自の構文があります。START WITH と CONNECT BY PRIOR です。この構文を以下のように用いて、再帰クエリを実行できます。

Trees/legit/connect-by.sql

```
SELECT * FROM Comments
START WITH comment_id = 9876
CONNECT BY PRIOR parent_id = comment_id;
```

オーバーエンジニアリング（やりすぎ）に注意しよう

以前、データセンター向けの設備管理アプリケーションを開発したことがあります。一部の設備は、コンピューターの内部に設置されていました。例えば、キャッシング・ディスク・コントローラがラック型マウントサーバーに、追加メモリモジュールがディスク・コントローラに設置されていました。

求められていたのは、階層的な集合として表されるこれら装置の使用状況を、SQL を用いて簡単に管理する方法でした。同時に、会計報告向けに、個々の装置の使用状況、償却コスト、投資収益率も管理する必要がありました。

データセンターの管理者からは、集合には下位集合が含まれる場合があり、理論上、ツリーの深さには制限がないと伝えられました。ツリーのデータベースへの格納、ユーザーインタフェース、管理、レポートなどの各機能を完成させるまでに、何週間もの期間を要しました。

しかし実際には、この設備管理アプリケーションで、1階層よりも深い階層で設備の親子関係をグループ化する必要はありませんでした。もし、データセンターの管理者が設備管理アプリケーションの要件を適切に理解していれば、かなりの労力を節約できたはずでした。

2.5 解決策：代替ツリーモデルを使用する

階層型データを格納するための設計として隣接リストモデルの代替となるものには、**経路列挙モデル**（Path Enumeration）、**入れ子集合モデル**（Nested Set）、**閉包テーブルモデル**（Closure Table）などがあります。続く3つの節では、「アンチパターン」の節で見てきた、ツリー構造を持つコメントの集合の格納とクエリ実行での失敗を、これらのモデルによってどのように解決できるかを見ていきます。

これらの解決策は、慣れるまでに少し時間を要します。始めのうちは、隣接リストよりも複雑に思えるかもしれません。しかし馴染むにつれ、隣接リストでは極めて非効率で手間のかかる複雑なツリー操作を、シンプルに実行できることに気づくでしょう。こうした複雑な操作が必要なアプリケーションでは、代替ツリーモデルの選択が効果的なのです。

2.5.1 経路列挙（Path Enumeration）

隣接リストには、先祖ノード取得の効率が悪いという弱点があります。**経路列挙**（Path Enumeration）は、先祖の系譜を表す文字列を各ノードの属性として格納することで、この問題を解決できます[†1]。

経路列挙の一例を、ディレクトリ構造に見ることができます。UNIXのパス /usr/local/lib/は、ファイルシステムの経路列挙です。usrが localの親で、localは libの親です。

Commentsテーブルで、parent_id列の代わりに path列を大きめの VARCHARとして定義します。この列に格納するのは、ツリーの上から下の順番で現在の行まで並べられたUNIXパスのような文字列です。区切り文字にはスラッシュ「/」も使えます。

Trees/soln/path-enum/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  path         VARCHAR(1000),
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

†1 監訳注：Path Enumeration は Materialized Path とも呼ばれます。

表 2-2 経路列挙のデータ

comment_id	path	発言者	コメント
1	1/	Fran	このバグの原因は何かな？
2	1/2/	Ollie	ヌルポインターのせいじゃないかな？
3	1/2/3/	Fran	そうじゃないよ。それは確認済みだ。
4	1/4/	Kukla	無効なインプットを調べてみたら？
5	1/4/5/	Ollie	そうか、バグの原因はそれだな。
6	1/4/6/	Fran	よし、じゃあチェック機能を追加してもらえるかな？
7	1/4/6/7/	Kukla	了解。修正したよ。

パスに対してパターン比較を行うことによって先祖を取得できます。例えば、パスが「1/4/6/7/」であるコメント7の先祖を取得するには、以下のように記述します[†2]。

Trees/soln/path-enum/ancestors.sql

```
SELECT *
FROM Comments AS c
WHERE '1/4/6/7/' LIKE c.path || '%' ;
```

このクエリは、先祖のパスが「1/4/6/%」、「1/4/%」、「1/%」のパターンとマッチします。

LIKE述語の引数を逆にすることで、子孫を取得できます。パスが「1/4/」のコメント4の子孫を特定するには、以下のように記述します。

Trees/soln/path-enum/descendants.sql

```
SELECT *
FROM Comments AS c
WHERE c.path LIKE '1/4/' || '%' ;
```

パターン「1/4/%」は、子孫「1/4/5/」、「1/4/6/」、「1/4/6/7/」とマッチします。

ツリーのサブセットや、ツリーの頂点に至るまでの先祖を簡単に選択できるようになると、サブツリーのノードの値（例：コスト）のSUMや、ノード数の計算など、他の多くのクエリを容易に実行できます。例えば、コメント4を頂点とするサブツリーで、発言者別のコメント数を計算するには、以下のように記述します。

Trees/soln/path-enum/count.sql

```
SELECT c.author, COUNT(*)
FROM Comments AS c
WHERE c.path LIKE '1/4/' || '%'
GROUP BY c.author;
```

ノードの挿入は、隣接リストの場合と似ています。他の行を修正せずに、非葉ノードを挿入できます。新

†2 監訳注：MySQLで文字列連結を行うには、文字列連結演算子 ||の代わりに CONCAT関数を使うか、または「set @@session.sql_mode='PIPES_AS_CONCAT';」を実行する必要があります。

しいノードの親からパスをコピーし、この文字列に新規ノードのIDを付加します。挿入時に主キーが自動的に値を生成する場合、いったん行を挿入した後に、新しい行のID値を用いてパスを更新する必要があるかもしれません。例えばMySQLの組込み関数LAST_INSERT_IDは、現在のセッションでの行挿入によって生成された最新のIDを返します。残りのパスは、親のノードから取得します[†3]。

Trees/soln/path-enum/insert.sql

```
INSERT INTO Comments (bug_id, author, comment_date, comment)
  VALUES (1234, 23, CURRENT_TIMESTAMP, 'ありがとう!');

UPDATE Comments
  SET path =
    (SELECT x.path FROM (
      SELECT path FROM Comments WHERE comment_id = 7
    ) AS x) || LAST_INSERT_ID() || '/'
WHERE comment_id = LAST_INSERT_ID();
```

経路列挙には、1章「ジェイウォーク（信号無視）」に示したものと同様の弱点があります。つまり、データベースはパスの正確な形成や、パス値の既存ノードへの対応を保証できないのです。パス文字列のメンテナンスはアプリケーションコードに依存し、その検証には手間がかかります。また、VARCHAR列の長さをどれだけ大きく定義するにしろ、つまるところ長さの制限があります。このため、厳密には無制限の深さのツリーをサポートできません。

また経路列挙には、セパレーター間の要素が一貫した長さである限り、階層構造による行のソートを容易に行えるという利点もあります[†4]。

2.5.2 入れ子集合（Nested Set）

入れ子集合（Nested Set）は、直近の親ではなく、子孫の集合に関する情報を各ノードに格納します。この情報は、各ノードのnsleftおよびnsrightと呼ばれる数値で表されます。

Trees/soln/nested-sets/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  nsleft       INTEGER NOT NULL,
  nsright      INTEGER NOT NULL,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
```

†3 監訳注：MySQLは更新対象のテーブルをUPDATE文のサブクエリに書けないという制限があるので、SELECT path FROM Comments WHERE comment_id = 7と書くところをSELECT x.path FROM (SELECT path FROM Comments WHERE comment_id = 7) AS xと書いています。この章の他のサンプルにも同様の修正を加えています。

†4 しかし、これはジェイウォーク（信号無視）アンチパターンに陥る危険を大いにはらんでいます。

```
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs (bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

各ノードにはルールに基づいてnsleftと nsrightの値が割り当てられます。具体的には、nsleft値にはそのノードより下の階層にあるすべてのノードが持つ値より小さな値が、nsright値にはそのノードより下の階層にあるすべてのノードが持つ値より大きな値が与えられます。これらの値は、comment_idの値とは関係ありません。

nsleftと nsrightの値は、深さ優先探索を用いることで、簡単に割り当てられます。すなわち、根から下に向かって降りていきながらノードに順番にnsleft値を割当て、葉から上に向かって nsright値を割り当てます。

表2-3と図2-3を見れば、このパターンをイメージしやすくなるでしょう。

表2-3 入れ子集合のデータ

comment_id	nsleft	nsright	発言者	コメント
1	1	14	Fran	このバグの原因は何かな？
2	2	5	Ollie	ヌルポインターのせいじゃないかな？
3	3	4	Fran	そうじゃないよ。それは確認済みだ。
4	6	13	Kukla	無効なインプットを調べてみたら？
5	7	8	Ollie	そうか、バグの原因はそれだな。
6	9	12	Fran	よし、じゃあチェック機能を追加してもらえるかな？
7	10	11	Kukla	了解。修正したよ。

これらの値を各ノードに割り当てることで、ノードの先祖や子孫を特定できるようになります。例えば、コメント4とその子孫は、コメント 4のノードの nsleftと nsrightの間に nsleftが含まれるノードをすべて検索することによって取得できます。

Trees/soln/nested-sets/descendants.sql

```
SELECT c2.*
FROM Comments AS c1
  INNER JOIN Comments as c2
    ON c2.nsleft BETWEEN c1.nsleft AND c1.nsright
WHERE c1.comment_id = 4;
```

同様に、コメント6とその先祖は、コメント 6のノードの nsleftを、nsleftと nsrightの間に含むノードをすべて検索することによって取得できます。

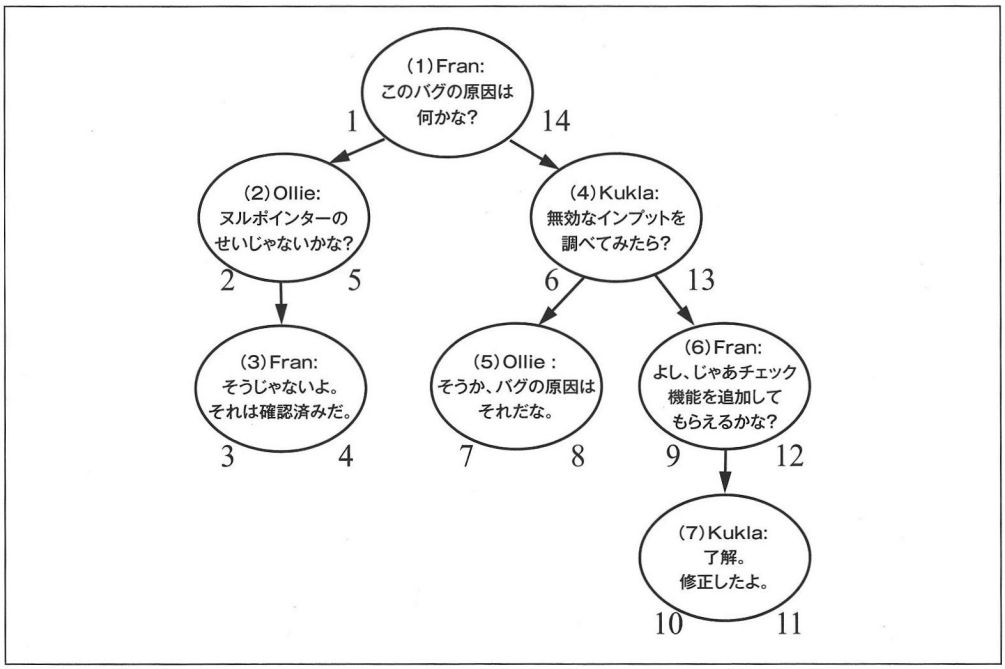


図2-3 入れ子集合

Trees/soln/nested-sets/ancestors.sql

```
SELECT c2.*
FROM Comments AS c1
  INNER JOIN Comments AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 6;
```

入れ子構造設計の大きな長所は、非葉ノードを削除すると、削除されたノードの子孫は、削除されたノードの親の直接の子であると自動的に見なされるという点です。また上の図で示したように、ノードのnsleftと nsrightの値は連続しており、隣接する兄弟や親との差は常に 1ですが、差が 1であることは、入れ子集合で階層構造を維持するための必要条件ではありません。このため、ノードの削除によって隣接するノードの値に 2以上の差が生じても、ツリー構造には影響がありません。

例えば、あるノードの深さを取得し、その親を削除してから再びノードの深さを取得すると、深さが1つ減少したことがわかります。

Trees/soln/nested-sets/depth.sql

```
-- この時点では depth は 4
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
```

```
FROM Comments AS c1
  INNER JOIN Comments AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;

DELETE FROM Comments WHERE comment_id = 6;

-- depth は 3 になる
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comments AS c1
  INNER JOIN Comments AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;
```

しかし、入れ子集合では、直近の親の取得などの、隣接リストでは簡単に実行できるクエリの一部が複雑になってしまいます。ノードc1の親はそのノードの先祖でもありますが、その間には他のノードは存在できません。このため、c1の先祖であり、かつ、c1の親の子孫でもあるノードを検索する**外部結合**を追加することで、直近の親を特定することができます。他にこの条件を満たすノードが見つからない場合（すなわち、外部結合の結果がNULLの場合）に、その先祖がc1の直近の親であるということになります。

例えば、コメント6の直近の親を取得するには、次のように記述します。

Trees/soln/nested-sets/parent.sql

```
SELECT parent.*
FROM Comments AS c
  INNER JOIN Comments AS parent
    ON parent.nsleft < c.nsleft AND c.nsleft < parent.nsright
  LEFT OUTER JOIN Comments AS in_between
    ON in_between.nsleft < c.nsleft AND c.nsleft < in_between.nsright
    AND parent.nsleft < in_between.nsleft AND in_between.nsleft < parent.nsright
WHERE c.comment_id = 6
  AND in_between.comment_id IS NULL;
```

入れ子集合では、ノードの挿入や移動などのツリーの操作が他のモデルよりも複雑になります。ノードの挿入では、新しいノードの左（nsleft）の値より大きいすべての左右のノードの値を計算し直す必要があります。

再計算が必要になるのは、新しいノードの右側の兄弟、新しいノードの先祖、そして新しいノードの先祖の右側の兄弟とその子孫たちです。なお、非葉ノードを新しく挿入する場合には、その新しいノードの子孫も再計算の対象になります。葉ノードの追加で必要な再計算をすべて行うためには、以下のように記述します。

Trees/soln/nested-sets/insert.sql

```
-- NS値8と9を入れるためにスペースを空ける
UPDATE Comments
  SET nsleft = CASE WHEN nsleft > 7 THEN nsleft+2 ELSE nsleft END,
      nsright = nsright+2
WHERE nsright >= 7;

-- コメント5の新しい子供を作成し、NS値に8と9を割り当てる
INSERT INTO Comments (nsleft, nsright, bug_id, author, comment_date, comment)
  VALUES (8, 9, 1234, 12, CURRENT_TIMESTAMP, '私もそう思います!');
```

入れ子集合が適しているのは、個々のノードの操作ではなく、サブツリーに対する迅速かつ容易なクエリ実行が重要な場合です。ノードの挿入や移動は、関連するノードの左右値の再計算が必要になるために複雑になります。ノードの挿入が頻繁に求められるツリーでは、入れ子集合の採用は最適とは言えません[†5]。

2.5.3 閉包テーブル（Closure Table）

閉包テーブル（Closure Table）は、階層構造のシンプルかつエレガントな格納方法です。閉包テーブルは、直接の親子関係だけではなく、ツリー全体のパスを格納します。

シンプルなCommentsテーブルに加えて、TreePathsテーブルを新たに定義します。TreePathsテーブルは、それぞれがCommentsテーブルの外部キーである2つの列を持ちます。

Trees/soln/closure-table/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE TreePaths (
  ancestor    BIGINT UNSIGNED NOT NULL,
  descendant  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (ancestor, descendant),
  FOREIGN KEY (ancestor) REFERENCES Comments(comment_id),
  FOREIGN KEY (descendant) REFERENCES Comments(comment_id)
);
```

†5 監訳注：『達人に学ぶDB設計徹底指南書』[Mic12]には、再計算量が多いという入れ子集合モデルの弱点を克服する「入れ子区間モデル」が紹介されています。

閉包テーブルでは、Commentsテーブルの代わりに、TreePathsテーブルを用いてツリー構造の情報を格納します。

このテーブルの各行には、先祖／子孫関係を共有するノードの組み合わせを格納します。これは、ツリー上の離れた位置にあるノードも含めた、すべてのノードが対象になります。また、各行のノードには、自分自身を参照する行も追加します。ノードの組み合わせを、表2-4と図2-4に示します。

表2-4 閉包テーブルのデータ

先祖	子孫	先祖	子孫	先祖	子孫
1	1	1	7	4	6
1	2	2	2	4	7
1	3	2	3	5	5
1	4	3	3	6	6
1	5	4	4	6	7
1	6	4	5	7	7

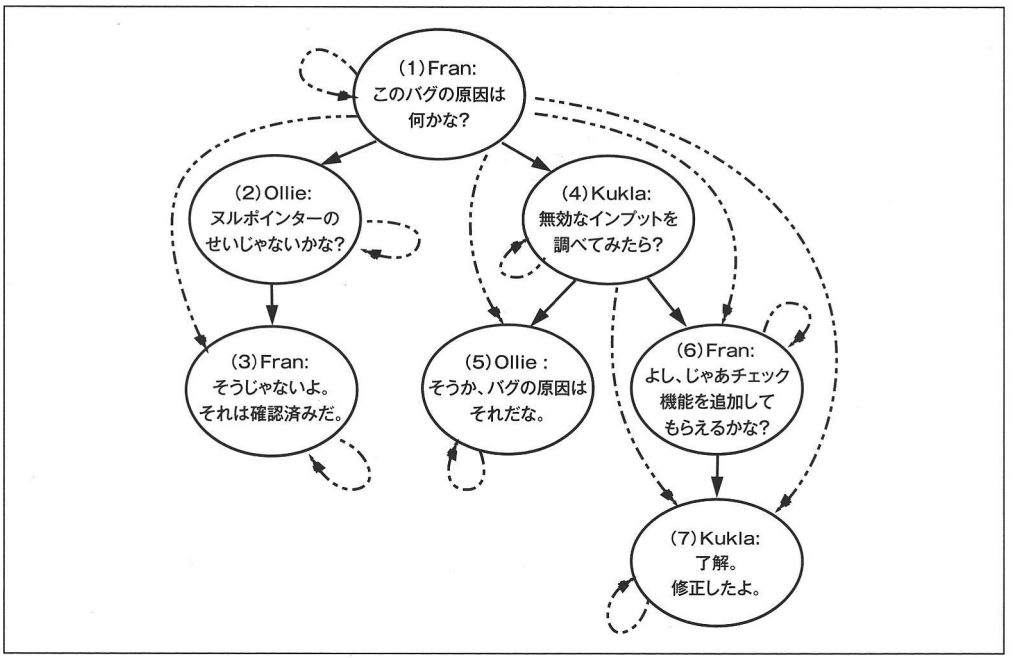


図2-4 閉包テーブル

このテーブルを用いて先祖と子孫を取得するクエリは、入れ子集合よりもさらに簡単です。コメント4の子孫を取得するには、TreePathsで先祖が4の行を探します。

Trees/soln/closure-table/descendants.sql

```
SELECT c.*
FROM Comments AS c
  INNER JOIN TreePaths AS t ON c.comment_id = t.descendant
WHERE t.ancestor = 4;
```

コメント6の先祖を取得するには、子孫が6の行を探します。

Trees/soln/closure-table/ancestors.sql

```
SELECT c.*
FROM Comments AS c
  INNER JOIN TreePaths AS t ON c.comment_id = t.ancestor
WHERE t.descendant = 6;
```

新たに葉ノード、例えばコメント5の子を挿入するには、最初に自己参照の行を挿入します。次に、コメント5を子孫として参照する行の集合（コメント5が自己参照する行を含む）を取得し、子孫を新しいコメントの番号で置き換えて新たに挿入します。

Trees/soln/closure-table/insert.sql

```
INSERT INTO Comments (comment_id, bug_id, author, comment_date, comment)
  VALUES (8, 1234, 23, CURRENT_TIMESTAMP, ' 確認お願いします ');

INSERT INTO TreePaths (ancestor, descendant)
  SELECT t.ancestor, 8
  FROM TreePaths AS t
  WHERE t.descendant = 5
 UNION ALL
  SELECT 8, 8;
```

葉ノード、例えばコメント7を削除するには、子孫としてコメント7を参照するすべての行を削除します。

Trees/soln/closure-table/delete-leaf.sql

```
DELETE FROM TreePaths WHERE descendant = 7;
```

サブツリー全体、例えばと4とその子孫をすべて削除するには、子孫としてコメント4を参照するすべての行と、コメント4の子孫を子孫として参照するすべての行を削除します。

Trees/soln/closure-table/delete-subtree.sql

```
DELETE FROM TreePaths
```

```
WHERE descendant IN (SELECT x.id FROM
                       (SELECT descendant AS id
                        FROM TreePaths
                        WHERE ancestor = 4) AS x);
```

TreePathsで行を削除しても、コメントそのものは削除されないことに注意してください。ここで紹介したCommentsテーブルの例では奇妙に思えるかもしれませんが、製品カタログのカテゴリーや組織図の従業員など、他の種類の階層構造を扱っている場合にはよくわかると思います。ノードの関連付けを変更するとき、必ずしもノードを削除するとは限りません。閉包テーブル設計では、パスを別のテーブルに格納することで、柔軟にノードの関連付けを変更できるようになります。

サブツリーを移動するには、まず、サブツリーのトップのノードとそのノードの子孫の先祖を参照する行を削除して、先祖からサブツリーを外します。例えば、コメント6を、コメント 4の子からコメント 3の子の位置に動かすには、まずは削除から始めます。コメント6の自己参照を削除しないように気をつけましょう。

Trees/soln/closure-table/move-subtree.sql

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT x.id FROM (SELECT descendant AS id
                     FROM TreePaths
                     WHERE ancestor = 6) AS x)
  AND ancestor IN (SELECT y.id FROM (SELECT ancestor AS id
                   FROM TreePaths
                   WHERE descendant = 6
                     AND ancestor != descendant) AS y);
```

サンプルのようにWHERE句を指定することで、6のすべての先祖とそれらの子孫を確実に削除できます。つまり、このステートメントはパス（1, 6）、（1, 7）、（4, 6）、（4, 7）を削除し、（6, 6）と（6, 7）は削除しません。

次に、移動先の先祖とサブツリーの子孫の組み合わせを表す行を挿入し、孤児になったサブツリーを追加します。新しい移動先の先祖とサブツリーのすべてのノードの組み合わせ行を生成するために、デカルト積（Cartesian product）を生成するCROSS JOIN構文が使えます。

Trees/soln/closure-table/move-subtree.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT supertree.ancestor, subtree.descendant
  FROM TreePaths AS supertree
    CROSS JOIN TreePaths AS subtree
  WHERE supertree.descendant = 3
    AND subtree.ancestor = 6;
```

このステートメントにより、コメント 3を含む 3の先祖と、コメント 6を含む 6の子孫のパスが新たに作られます。新しいパスは、(1, 6)、(2, 6)、(3, 6)、(1, 7)、(2, 7)、(3, 7) です。この結果、コメント 6から開始するサブツリーは、コメント 3の子の位置に移動します。サブツリーの移動先がツリー上で根に近くなるか葉に近くなるかにかかわらず、クロス結合はすべての必要とされるパスを作成します。

閉包テーブル設計は、入れ子集合よりもはるかにシンプルです。どちらの設計も先祖と子孫への迅速かつ容易なクエリ実行が可能ですが、閉包テーブルでは階層構造のメンテナンスが入れ子集合よりも簡単です。ただしどちらの設計も、隣接リストや経路列挙に比べて、直近の子や親へのクエリがやや複雑になります。

しかし、直近の親や子ノードへのクエリをより容易に書けるように閉包テーブルを改善できます。その方法は、閉包テーブルへの TreePaths.path_length属性の追加です。 path_lengthの値は、ノードの自己参照は 0、直近の子は 1、孫は 2という具合に割り当てられます。こうすることで、例えばコメント 4の子を、以下のように簡単に取得できます。

Trees/soln/closure-table/child.sql

```
SELECT *
FROM TreePaths
WHERE ancestor = 4 AND path_length = 1;
```

2.5.4 どの設計を使うべきか

それぞれの設計には、長所と短所があります。最も重要な操作に沿って、最適な設計を選ぶことが大切です。表 2-5に、各設計で簡単な操作と難しい操作を示しました。長所と短所を含む、各設計の概要を以下に述べます。

表 2-5 階層的なデータ設計の比較

設計	テーブル数	子へのクエリ実行	ツリーへのクエリ実行	挿入	削除	参照整合性維持
隣接リスト	1	簡単	難しい	簡単	簡単	可能
再帰クエリ	1	簡単	簡単	簡単	簡単	可能
経路列挙	1	簡単	簡単	簡単	簡単	不可
入れ子集合	1	難しい	難しい	難しい	難しい	不可
閉包テーブル	2	簡単	簡単	簡単	簡単	可能

隣接リスト

従来最も一般的に用いられてきた設計で、ソフトウェア開発者の多くが知っている方法です。

再帰クエリ

WITHや CONNECT BY PRIOR構文をサポートするデータベース製品では、隣接リスト設計であっても計算効率を高められます。

経路列挙

「パンくず」ユーザーインタフェースとして効果的ですが、参照整合性が保証できず、冗長な情報を格納するために、脆弱な面を持っています。

入れ子集合

巧妙な設計手法ですが、巧妙すぎるとも言えます。経路列挙と同じく、参照整合性を保証できません。ツリーの修正よりも、ツリーの検索の必要が多い場合に適しています。

閉包テーブル

この章で紹介した設計のなかで、最も用途が幅広く、また唯一、ノードが複数のツリーへ所属することができます。関連付けを格納するために、別個のテーブルが必要です。また、階層が深くなると多くの行数が必要になり、計算が楽になる分、スペースが消費されるというトレードオフが生じます。

SQLでの階層的データの格納と操作については、学ぶべきことは他にもたくさんあります。階層に対するクエリ実行を扱う良書として、Joe Celko著『Trees and Hierarchies in SQL for Smarties』[Cel04]を挙げられます。また、ツリー構造だけでなく**グラフ構造**も対象にした書籍にVadim Tropashko著『SQL Design Patterns』[Tro06]があります。後者は、より形式的な、アカデミックなスタイルで書かれています[†6]。

階層構造はエントリと関連（リレーションシップ）から成り立ちます。これを念頭に、行う作業に合わせて最適な設計手法を選択しましょう。

†6 監訳注：『Trees and Hierarchies in SQL for Smarties』[Cel04]の邦訳は残念ながらありませんが、階層構造の扱いに関して日本語で読めるセルコの著作には『プログラマのためのSQL第2版』[Cel01]があります。

3章
IDリクワイアド（とりあえずID）

屋外の動物たちは、豚から人間に、また、人間から豚へ目を移し、もう一度、豚から人間に目を移した。
しかし、もう、どちらがどちらか、さっぱり見分けがつかなくなっていたのだった。
——ジョージ・オーウェル『動物農場』（角川文庫／高畠文夫訳）

最近、あるソフトウェア開発者から、「行の重複を避けるにはどうすればよいか」という質問がありました。過去に何度も経験したことのある質問です。最初は、主キーが設定されていないことが原因かと思いましたが、問題はそうではありませんでした。

対象のアプリケーションは、ウェブサイトに掲載する記事を格納するコンテンツ管理データベースです。記事のテーブルとタグのテーブルを多対多で関連付ける交差テーブルが用いられています。

ID-Required/intro/articletags.sql

```
CREATE TABLE ArticleTags (
  id          SERIAL PRIMARY KEY,
  article_id  BIGINT UNSIGNED NOT NULL,
  tag_id      BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (article_id) REFERENCES Articles (id),
  FOREIGN KEY (tag_id) REFERENCES Tags (id)
);
```

あるタグに関連付けられた記事の数を計算しようとすると、クエリから誤った結果が返されるというのです。彼は、economyタグに関連付けられた記事は5件しかないと知っていました。しかし、クエリは7件という結果を返します。

ID-Required/intro/articletags.sql

```
SELECT tag_id, COUNT(*) AS articles_per_tag FROM ArticleTags WHERE tag_id = 327;
```

対象のtag_idと一致するすべての行を検索すると、そのタグが1件の記事と3重に関連付けられている

ことがわかりました（表3-1）。IDの値が異なるにもかかわらず、これらの3行は同じ関連付けを持っていたのです。

表3-1 3重の関連付け

id	tag_id	article_id
22	327	1234
23	327	1234
24	327	1234

このテーブルには主キー（id列）がありますが、主キーだけでは他の2つの列（tag_id, article_id）の組み合わせの重複を防ぐことができていませんでした。考えられる解決策の1つは、tag_id列とarticle_id列への複合UNIQUE制約の追加です。しかし、だとすれば、そもそもなぜid列が必要になるのでしょうか。

3.1 目的：主キーの規約を確立する

目的は、すべてのテーブルが**主キー**（primary key）を持つことです。しかし主キーの特性についての誤解が、アンチパターンを生じさせてしまっています。

データベース設計を学ぶ人は誰しも、テーブルにとって主キーは非常に重要な要素であり、必要不可欠なものであると教わります。たしかに、それは間違いではありません。主キーはデータベースをうまく設計するために欠かせないものです。主キーは、テーブルのすべての行が一意であることを保証するものであり、個々の行を扱い、行の重複を避けるために必要な論理的メカニズムです。また、主キーは**外部キー**（foreign key）から参照されることで、テーブルの関連付けを行います。

しかし難しいのは、どの列を主キーに選ぶかという問題です。テーブル内の列の属性値はたいてい、重複する可能性を秘めています。典型的な例としてよく挙げられるのは名字や名前で、これは明らかに重複してしまう可能性があることがわかります。メールアドレスや、行政上の個人識別番号（アメリカにおける社会保障番号や納税者番号など）ですら、厳密には一意であるとは限りません。

ある対象領域をモデル化したテーブルに、その領域で意味を持たない人工的な値を格納するには、新たな列を追加しなければなりません。この列を主キーとして使用することで、他の属性列には重複を許可しながら、行を一意のものとして扱えるようになります。このタイプの主キー列は、**疑似キー**（pseudo key）や**代理キー**（surrogate key）と呼ばれることもあります。

複数クライアントから同時に行が挿入された時にも疑似キー値が一意であることを保証するために、データベースの多くでは、一意の連続した整数値を、トランザクションの外側のスコープで生成する仕組みを備えています。

疑似キー値を生成する機能は、SQL:2003までは標準化されていなかったため、各種のデータベースでは独自の拡張機能を用いてSQLにこの機能を実装しています。機能を表す用語も、表3-2に示すようにベンダーによって異なります。

表 3-2 各データベース製品における疑似キー

機能名	サポートしているデータベース
AUTO_INCREMENT	MySQL
GENERATOR	Firebird、InterBase
IDENTITY	DB2、Derby、Microsoft SQL Server、Sybase
ROWID	SQLite
SEQUENCE	DB2、Firebird、Informix、Ingres、Oracle、PostgreSQL
SERIAL	MySQL、PostgreSQL

疑似キーは便利な機能です。ただし、それは主キーを宣言するための唯一の解決策ではないのです。

主キーが本当に必要か

「このテーブルには主キーは不要だ」という言葉を、何人かのソフトウェア開発者から聞いたことがあります。

彼らは一意のインデックスでの管理を面倒だと考えていたり、主キーとして使える列がないテーブルを扱っていたりするようです。

主キー制約が重要になるのは、次のような必要があるときです。

- 行の重複を避けたい
- クエリで個別の行を参照したい
- 外部キー参照をサポートしたい

主キー制約を使わないと、例えば以下のように重複行をチェックするという、煩わしい手間が生じます。

```
SELECT bug_id FROM Bugs GROUP BY bug_id HAVING COUNT(*) > 1;
```

主キー制約を設定することを避けたばかりに、この先このようなチェックを何度実行することになるのでしょうか。もし重複してしまったときはどのように対処できるでしょうか。

主キーのないテーブルは、大量の MP3 の音楽ファイルを、曲タイトルなしで管理するようなものです。音楽は聴けますが、聴きたい曲を特定したり、同じ曲が重複しないようにすることはできません。

3.2 アンチパターン：すべてのテーブルに「id」列を用いる

書籍や雑誌記事、フレームワークなどの影響で、すべてのテーブルには以下の特徴を持つ主キー列が存在しなくてはならないという考えが普及しています。

- 列名はid
- データ型は32ビットまたは64ビットの整数
- 一意の値が自動的に生成される

すべてのテーブルにidという名前の列がある状況は極めて一般的であるため、idは主キーの同義語になっています。SQLを学ぶプログラマーは、主キーとはこのように定義された列だと考えるようになりますが、実はそれは誤解なのです。

ID-Required/anti/id-ubiquitous.sql

```
CREATE TABLE Bugs (
  id          SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  -- 他の列 . . .
);
```

すべてのテーブルにid列を加えると、意図に反した影響が生じることがあります。

3.2.1 冗長なキーが作成されてしまう

同じテーブルにある別の列が「**自然な**」**主キー**（**自然キー、ナチュラルキー**）として使えそうであっても、さらにその列にUNIQUE制約を付与できそうであっても、慣習を重視して、主キーとしてのid列を作成してしまうことがあります。例えばBugsテーブルでは、次のbug_id列のようにプロジェクト名や他の識別情報を使ってバグをラベリングすることがあります。

ID-Required/anti/id-redundant.sql

```
CREATE TABLE Bugs (
  id          SERIAL PRIMARY KEY,
  bug_id      VARCHAR(10) UNIQUE,
  description VARCHAR(1000),
  -- 他の列 . . .
);
INSERT INTO Bugs (bug_id, description, ...)
  VALUES ('VIS-078', '保存時にクラッシュが発生', ...);
```

このbug_id列は、各行で一意になっているので、id列と同じように利用することができます。

3.2.2 重複行を許可してしまう

複合キー（compound key）は、複数の列で構成されたキーです。複合キーは、**交差テーブル**（前述のBugsProductsなど）で良く用いられます。BugsProductsの主キーは、bug_idと product_idの値の組み合わせが、テーブル上で一意であることを保証しなければなりません（組み合わせに用いられるbug_idとproduct_idのそれぞれの値は、他の値との組み合わせに何度も登場することができます）。

しかし、単に慣習に従ってid列を主キーとして使用してしまうと、bug_idと product_idの組み合わせが常に一意であることを保証しなくなってしまいます。

ID-Required/anti/superfluous.sql

```
CREATE TABLE BugsProducts (
  id          SERIAL PRIMARY KEY,
  bug_id      BIGINT UNSIGNED NOT NULL,
  product_id  BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 1), (1234, 1); -- 重複を許可してしまう
```

交差テーブル内の重複は、このテーブルを使ってBugsと Productsを一致させるときに、思いがけない結果を招きます。重複を防ぐためには、id以外の２列に UNIQUE制約を宣言することもできます。

ID-Required/anti/superfluous.sql

```
CREATE TABLE BugsProducts (
  id          SERIAL PRIMARY KEY,
  bug_id      BIGINT UNSIGNED NOT NULL,
  product_id  BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

しかし、id以外の２列に UNIQUE制約が必要であるならば、id列はただの無駄です。

3.2.3 キーの意味がわかりにくくなる

コード（code）という単語は、暗号化、すなわち情報を簡潔かつ秘密裏に伝達する方法、という意味を持っています。しかしプログラムを書くときには、意味をより明確にしていこうと「コーディング」しているはずです。

idという列名は極めて一般的であるため、明確な意味を持ちません。これは、idという主キー名を持つ

2つのテーブルを結合する際に留意すべき、重要なポイントです。

ID-Required/anti/ambiguous.sql

```
SELECT b.id, a.id
FROM Bugs b
INNER JOIN Accounts a ON b.assigned_to = a.id
WHERE b.status = 'OPEN';
```

クエリの結果の列を列番号ではなく列名で参照するとき、バグのidとアカウントのidをどう区別すればよいのでしょうか。これは、クエリの結果が連想配列として返されるPHPのような動的言語では特に問題になります。クエリの中で列のエイリアスを指定しなければ、連想配列上で列が上書きされてしまうからです。

「id」という列名は、このクエリの明確化には役立ちません。列名をbug_idとaccount_idにすれば、クエリの結果はずっとわかりやすくなります。主キーは「あるテーブル上の1つの行」を識別するためのものなので、主キーの列名でもそのテーブルの種類が分かるようにしておくべきです。そうすれば「bug_id = 1」という情報だけで、データベース上の1行を特定することができます。

3.2.4 USINGを使用する

予約語ONに続けて、2つのテーブルの結合条件を表す式を書く形式のJOIN構文に、馴染みのある読者が多いのではないでしょうか。

ID-Required/anti/join.sql

```
SELECT * FROM Bugs AS b
  INNER JOIN BugsProducts AS bp
    ON b.bug_id = bp.bug_id;
```

SQLでは、2つのテーブルを結合するための、さらに簡潔な構文もサポートしています。両方のテーブルに同じ名前の列がある場合、上記のクエリをUSINGを使用して以下のように書き直せます。

ID-Required/anti/join.sql

```
SELECT * FROM Bugs INNER JOIN BugsProducts USING (bug_id);
```

しかし、すべてのテーブルで主キーを「id」という名前にしなければならないとすると、従属テーブル側の外部キー列には、参照する主キーと同じ名前は使えません。代わりに、冗長なON構文を常に使わなくてはなりません。

ID-Required/anti/join.sql

```
SELECT * FROM Bugs AS b
  INNER JOIN BugsProducts AS bp
    ON b.id = bp.bug_id;
```

3.2.5 複合キーは使いにくい

「複合キーは使いにくいので、いっさい使わない」という開発者もいます。別の複合主キーとの比較を行う場合に、どのような式でも、すべての複合キー列を比較しなくてはなりません。複合主キーを参照する外部キーは、それ自体も複合外部キーでなくてはいけません。このため、複合キーを使用する場合は、コードが長くなってしまいます。

このような開発者の複合キーへの拒否反応は、数学者が2次元や3次元の座標を使うことを拒否し、物体が1次元に存在するかのようにして計算を行うことに似ています。これによって、幾何や三角法をはるかに単純化できることは事実ですが、対象とすべき現実世界の物体を正確に表すことはできません。

シーケンスの特殊なスコープ

以下のように、現在使用中の最大値をとり、1を加えて新しい行に値を割り当てる方法があります。

```
SELECT MAX(bug_id) + 1 AS next_bug_id FROM Bugs;
```

しかし、複数のクライアントが同時にクエリを実行するときには、この方法の信頼性は低下します。複数のクライアントが、同じ値を取得する可能性があるからです。これは**競合状態**（race condition）と呼ばれます。

競合状態を避けるためには、現在の最大値を読み取って新規行に使用する間に、他のクライアントが行を追加しないようにしなければなりません。このためには行レベルのロックでは不十分であり、テーブル全体をロックする必要があります。ロックされている間は他のクライアントはロック解放待ち行列に加わることになるので、テーブルロックによってボトルネックが生じてしまいます。

シーケンスは、トランザクション外で処理を行うことによって、この問題を解決します。シーケンスは絶対に複数のクライアントに同じ値を割り当てません。そして値を行にコミットするか否かにかかわらず、値の割当てをロールバックしません。したがって、複数のクライアントが同時にアクセスしても、同一値が割り当てられることはなく、一意の値を並行して生成できます。

データベースの多くは、シーケンスが生成した最新の値を返す関数をサポートしています。例えばMySQLでは、LAST_INSERT_ID関数がこの役割を果たします。同じくMicrosoft SQL ServerにはSCOPE_IDENTITY、Oracleにはシーケンス名.CURRVALがあります。

これらの関数は、他のクライアントが同時にシーケンスから値を生成しているような状況でも、自クライアントの現在のセッションで生成された値を返します。このため、競合状態は生じません。

3.3 アンチパターンの見つけ方

このアンチパターンの兆候は簡単に見つかります。テーブルの主キーの列名が「id」であれば、それが「IDリクワイアド（とりあえずID）」アンチパターンの兆候です。記述的な名前の代わりに「id」を列名に使うことを正当化する理由はないと言ってもいいでしょう。

他にも、以下のような言葉を耳にしたら、このアンチパターンに注意すべきです。

「このテーブルでは主キーは不要だと思うな」

この開発者は、主キーと疑似キーを混同しています。すべてのテーブルには、重複行を防ぎ、個々の行を識別するための主キー制約が必要です。主キーには、疑似キーではなく自然キーや複合キーも使えます。

「どうして、多対多の関連が重複してしまったんだろう？」

多対多関連のための交差テーブルでは、外部キー列の組み合わせに対して主キー制約を宣言するか、少なくともUNIQUEキー制約を宣言すべきです。

「あるデータベース理論によると、値を参照テーブルに移動し、IDでそれらを参照すべきらしい。でも、値を取得するたびに結合をしなければならないから、そうしたくはないな」

これは正規化と呼ばれるデータベース設計理論についてのよくある誤解です。正規化は、疑似キーとは何の関連もありません。この点についての詳細は、付録A「正規化のルール」で説明します。

3.4 アンチパターンを用いてもよい場合

一部のオブジェクトリレーショナルマッピング（ORM）フレームワークは、開発をシンプルにするために、**設定より規約**（convention over configuration）の原則に従っています。これらのフレームワークでは、すべてのテーブルで同じように主キーを定義します。すなわち、列名はid、データ型は整数の疑似キー列です。このようなフレームワークでは、規約に従った方がよい場合があります。フレームワークの便利な機能を使いやすくなるからです。

疑似キーの使用や、整数値の自動インクリメント機構による採番に問題があるわけではありません。ただ、すべてのテーブルが疑似キーを必要とするわけではなく、また、すべての疑似キーを「id」と命名する必要もないことに注意しましょう。

また疑似キーは、自然キーがあまりにも長すぎる場合にはよい選択肢になります。例えば、ファイルシステムの上のファイルの属性を記録するテーブルでは、ファイルパスを自然キーとして選択できるかもしれません。しかし、ファイルパスのように長い文字列の列にインデックスをつけるのは非効率です。

3.5 解決策：状況に応じて適切に調整する

主キーは制約であり、データ型ではありません。データ型がインデックスをサポートしている限り、単一の列や複数の列を主キーとして宣言できます。また、自動インクリメントする整数型の列を、主キーではない列としても定義できます。この2つは**独立した概念**です。

良い設計のためには、規約に縛られて柔軟性を欠いてしまわないようにすることが大切です。

3.5.1 わかりやすい列名にしよう

主キーには、**わかりやすい名前**をつけましょう。主キーの名前は、主キーが識別する対象のエンティティを表すものにすべきです。例えば、Bugsテーブルの主キーの名前は bug_idが相応しいでしょう。

可能ならば、外部キーの列名にも同じような命名規則を用いましょう。これはつまり、主キー名はスキーマ内で一意となるべきであることを意味します。片方の主キーがもう片方の主キーを参照する外部キーでは

ない限り、複数のテーブルで主キーに同じ名前を使うべきではありません。しかし例外もあります。参照する主キーとは異なる命名規則の名前を外部キーに与えることが適切な場合があります。例えば、関連の意味を示すような名前です[†1]。

ID-Required/soln/foreignkey-name.sql

```
CREATE TABLE Bugs (
  -- 他の列 . . .
  reported_by  BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
```

メタデータの命名規則は、ISO/IEC 11179 [†2] が業界標準の規格になっています。この規格は、情報技術システムにおける「分類体系の管理」に関するガイドラインです。つまり、テーブルと列に実用的な名前をつけるための基準を示すものです。他のISO規格と同じく、この文書は膨大な量があり、読み通すのは大変ですが、幸い、Joe Celko著『SQL Programming Style』[Cel05]で、この規格についてSQLに焦点を絞った解説が読めます。

3.5.2 規約に縛られない

ORMフレームワークの多くは、「id」という名前の疑似キーが使われることを規約としています。しかし、この規約を上書きして、別の名前を宣言することも可能です。以下に、Ruby on Railsの例を示します[†3]。

ID-Required/soln/custom-primarykey.rb

```
class Bug < ActiveRecord::Base
  self.primary_key = 'bug_id'
end
```

規約から外れた主キー名の明示的な指定が必要なのは、望ましい規約を使えない、**レガシーデータベース**を扱う場合のみであるという考えかたもあるでしょう。しかし実は、列名の明示的な指定は、実用的な主キー名を設定できるという意味で、むしろ新しいプロジェクトでこそ重要なのです。

3.5.3 自然キーと複合キーの活用

一意であることが保証できて、NULLを許容しない、行の識別に使える属性がテーブルに含まれているのであれば、単に慣習に従うという理由のために、疑似キーを追加する義務を感じる必要はありません。

現実的には、テーブルのすべての属性が変更の対象であったり、一意性が無いというようなことは珍しくありません。データベースは、プロジェクト期間中に大きく設計が変化する傾向があります。プロジェクトの意思決定者が、自然キーの一意性を尊重しない決定をすることもあるでしょう。初めは良い自然キーにな

†1 監訳注：関連の意味を示すために、外部キーの名前がaccount_idではなくreported_byになっています。
†2 http://metadata-standards.org/11179/
†3 「Rails」および「Ruby on Rails」は、David Heinemeier Hanssonの登録商標です。

ると思われた列が、途中から重複を許可するようになってしまう場合もあります。これらのケースの唯一の解決策は、疑似キーの使用です。

複合キーは、適切な場合に用いるようにしましょう。例えば交差テーブルBugsProductsのように、行を識別するための最適な方法が複数の属性列の組み合わせである場合に、複合主キーを使うとよいでしょう。

ID-Required/soln/compound.sql

```
CREATE TABLE BugsProducts (
  bug_id      BIGINT UNSIGNED NOT NULL,
  product_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 2), (1234, 3);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1); -- エラー：重複エントリ
```

1つ注意すべき点は、複合主キーを参照する外部キーもまた、列の組み合わせでなくてはならないことです。このため、従属テーブル側では同じ列定義が重複することになり、簡潔さを欠いてしまいます。ただ、利点もあります。自然キーの複合主キーを参照する外部キー列は、疑似キーの場合には結合しないと取得できなかった（自然キーの持っている）情報を既に持っているので、外部キーの情報だけで足りる場合には、結合を行わずにシンプルにクエリを書けるのです。

> 規約は、役立つと思える場合のみ従いましょう。

4章
キーレスエントリ（外部キー嫌い）

勝兵はまず勝ちて而る後に戦いを求め、敗兵はまず戦いて而る後に勝を求む。
——孫子

ある日、研究所の管理者が血相を変えて私のところにやってきました。「ビル、大変だ。研究所の設備を2人のマネージャーが同じ日に予約してしまっているんだ。なぜこんなことになってしまったんだろう。すぐに調べて修正してほしい。どちらのマネージャーも、その日にテスト設備がどうしても必要で、使えなければスケジュールが遅れてしまうとクレームを言ってきているんだ」

そのアプリケーションは、数年前に私がMySQLを用いて設計した設備管理アプリケーションでした。MySQLのデフォルトのストレージエンジンは外部キー制約をサポートしていないMyISAMでした。このため、データベースには多くの論理的な関連（リレーションシップ）があったにもかかわらず、参照整合性を強制できませんでした。

プロジェクトが進むにつれ、アプリケーションでのデータ操作の方法が変わり、問題が生じ始めました。参照整合性を満たせないときに、レポートに不整合が出始め、小計が合わず、スケジュールにも重複が生じるようになったのです。

プロジェクトマネージャーからは、不整合を早めに検出するために定期的に実行するデータ品質管理スクリプトの作成を依頼されました。そこで私は、データベースを調べ、子テーブルで孤児になっている行が存在するなどの不正な状態を検出して、結果を電子メールで報告するスクリプトを何本か書きました。

スクリプトでは、すべてのテーブルの関連チェックが必要でした。データ容量が大きくなり、テーブルの数が増えるにつれ、品質管理用のクエリの数も増え、スクリプトの実行に時間がかかるようになり、電子メールの報告も長くなりました（このような状況が、身に覚えのある読者もいるのではないでしょうか）。

もちろん、このスクリプトによる解決策は機能しました。しかしこの手間のかかる作業は、本来は不要なはずのものでした。本当に必要だったのは、ユーザーが無効なデータを入力した際に、アプリケーションがすぐにエラーを返すことでした。この話を読んで、何のために外部キー制約の役割があるのか、わかるような気がしてきたのではないでしょうか。

4.1 目的：データベースのアーキテクチャを単純化する

リレーショナルデータベースの設計において、各テーブルの設計と同じくらい重要なのが、テーブル間の関連（リレーションシップ）の設計です。**参照整合性**は、適切なデータベース設計と運用において極めて重要です。1つの列、または複数の列に**外部キー制約**を宣言するとき、これらの列の値が、親テーブルの主キー列やユニークキー列に存在しなくてはいけません。ここまでは、ごくシンプルな話です。

しかし、参照整合性制約を使わないように勧める考え方もあります。外部キーを使用しない主な理由としては、以下のものが挙げられます。おそらく、あなたも見聞きしたことがあるのではないでしょうか。

- データの更新が、参照整合性制約と衝突してしまう。
- データベース設計の柔軟性が極めて高いので、参照整合性制約をサポートできない。
- データベースが外部キーのために作成するインデックスが、パフォーマンスに影響すると考えている。
- 外部キーをサポートしないデータベース製品を使っている。
- 外部キーを宣言する構文を調べなければならない。

4.2 アンチパターン：外部キー制約を使用しない

外部キー制約を省略すれば、データベース設計がシンプルになり、柔軟性が高まり、実行速度が速くなると思っている読者もいるかもしれません。しかし、そこには代償があり、代償は別の形で支払わなくてはならないのです。つまり、開発者であるあなたに、参照整合性を保証するためのコードを書く責任が生じるのです。

4.2.1 完ぺきなコードを前提にしている

参照整合性を保証するために多く用いられる解決策は、データの関連付けを常に維持するためのコードを書くことです。行の挿入時には、外部キー列の値が参照先のテーブルの既存の値を参照していることを確認します。行の削除時には、それに合わせてすべての子テーブルも適切に更新されることを確認します。言い換えるなら、多くの開発者が用いる、この最も一般的な解決策の本質とは、「ミスをしないようにする」ということなのです。

外部キー制約を設定しなかった場合、参照整合性が損われることを避けるには、変更を適用する前に追加のSELECTクエリを実行し、変更によって参照が壊れないことを確認しなければなりません。例えば、行を挿入する前に、親の行の存在を確認する必要があります。

Keyless-Entry/anti/insert.sql

```
SELECT account_id FROM Accounts WHERE account_id = 1;
```

ようやく、アカウントを参照するバグレポートを追加できます。

Keyless-Entry/anti/insert.sql

```
INSERT INTO Bugs (reported_by) VALUES (1);
```

行を削除するためには、子の行が存在しないことを確認しなければなりません。

Keyless-Entry/anti/delete.sql

```
SELECT bug_id FROM Bugs WHERE reported_by = 1;
```

これで、アカウントを削除できます。

Keyless-Entry/anti/delete.sql

```
DELETE FROM Accounts WHERE account_id = 1;
```

しかし、もしaccount_idが1のアカウント利用者があなたの気づかないところで作業をしていて、あなたが確認クエリを実行した直後、かつアカウントの削除直前に、新しいバグレポートを登録したらどうなるでしょうか。そんなことは、めったにありそうもないことのように思えます。しかし、Gordon Letwin（DOS 4のアーキテクト）の有名な言葉の通り、「100万分の1の確率で生じることは、次の火曜日に起きる」のです。存在しないアカウントによって報告されたバグ、という不正なレコードがBugsテーブル上にそのまま残ってしまうことになります。

唯一の対処策は、Bugsテーブルを明示的にロックしながらチェックを行い、アカウントの削除後にロックを解除することです。しかし、この種のロックを必要とするアーキテクチャでは、同時接続ユーザーが増え、スケーラビリティが求められるようになるにつれ、様々な問題に直面してしまいます。

4.2.2 ミスを調べなければならない

この章の冒頭の逸話で紹介した「間違った解決策」は、破損データを報告するスクリプトを開発者が書くというものでした。

本書のバグデータベースでは、Bugs.status列は**参照テーブル**BugStatusを参照しています。無効な値を持つバグレポートを特定するには、以下のようなクエリを書きます。

Keyless-Entry/anti/find-orphans.sql

```
SELECT b.bug_id, b.status
FROM Bugs b LEFT OUTER JOIN BugStatus s
  ON b.status = s.status
WHERE s.status IS NULL;
```

似たようなクエリを、すべての参照に対して書かなければならないことが想像できるのではないでしょうか。

壊れた参照をチェックする処理を書いているときには、チェックをどれくらいの頻度で行うかを考えなく

てはなりません。数百ものチェックを毎日、あるいは1日に複数回、実行しなくてはならないこともありえます。これは、相当に煩わしい仕事です。

壊れた参照を検出したときは、何をしなければならないのでしょうか。壊れた参照は修正できるのでしょうか。修正できるものもあるでしょう——ただし、毎回そうだとは限りません。例えば、無効なバグステータス値を、有効なデフォルト値に変更することなどが考えられます[†1]。

Keyless-Entry/anti/set-default.sql

```
UPDATE Bugs SET status = DEFAULT WHERE status = 'BANANA';
```

この種の不具合には必然的に、データ変更による修正が行えない場合があります。例えば、Bugs.reported_by列は対象のバグを報告したユーザーのアカウントを参照すべきですが、この列に無効な値が入ってしまった場合、どのユーザーのアカウントを代わりに使えばよいのでしょうか。

4.2.3 「私のミスではありません！」

データベースに関するすべてのコードが完ぺきであることは、まずありえないと考えていいでしょう。例えば、アプリケーションの様々な箇所に似たような更新処理が散らばっているとして、次にコードを変更するときに、アプリケーション内のすべての箇所で適切な変更を行ったことを確認するのは、簡単ではありません。

ユーザーがSQL実行ツールや自ら書いたスクリプトによって、直接データベースを変更することもあるかもしれません。アドホックなSQLによって、参照は簡単に壊れてしまいます。長い期間アプリケーションを運用していれば、このような事態がいつかは生じることを想定しなければなりません。

データベースには整合性の維持を、言い換えるなら、参照がいつも正しく保たれていることを期待するでしょう。しかし、データベースにアクセスするすべてのアプリケーションとスクリプトが正しく変更を行ったかどうかは、信頼できるものではありません。

4.2.4 「キャッチ=22」なUPDATE

多くの開発者が外部キーを避けるのは、複数のテーブルの関連し合う列を更新する際に、外部キー制約が邪魔になると感じるからです。例えば他の行が参照している行（親）を削除する必要がある場合、外部キー制約違反を避けるためには、まず子側の行から削除しなければなりません。

Keyless-Entry/anti/delete-child.sql

```
DELETE FROM BugStatus WHERE status = 'BOGUS'; -- エラー！
DELETE FROM Bugs WHERE status = 'BOGUS';
DELETE FROM BugStatus WHERE status = 'BOGUS'; -- 今度は成功する
```

このように、それぞれの子テーブルに対してステートメントを明示的に実行しなければいけません。将来の機能拡張でデータベースに別の子テーブルを追加する場合には、新しい子テーブルの行から先に削除する

†1 例に示すように、SQLには予約語DEFAULTがあります。

ように既存のコードを修正しなければなりません。ただ、ここまでは、単にそのように修正すれば解決できる話です。

解決できない問題もあります。子の行が依存する列をUPDATEする場合です。親の行を更新するまで、子の行は更新できません。また、親の更新も、参照する子の値を更新する前には実行できません。両方を同時に変更する必要がありますが、2つの異なる更新処理を同時に実行することは不可能です。これはまさに「キャッチ=22」的なジレンマに陥るシナリオだと言えます[†2]。

Keyless-Entry/anti/update-catch22.sql

```
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS'; -- エラー！

UPDATE Bugs SET status = 'INVALID' WHERE status = 'BOGUS'; -- エラー！
```

このような状況に陥りたくないため、外部キーをまったく使わないことを決意する開発者もいます。しかし後述するように、外部キーを用いてシンプルかつ効率的に複数テーブルの更新と削除を行う方法はあるのです。

4.3 アンチパターンの見つけ方

次のような言葉を耳にしたら、それはおそらく「**キーレスエントリ（外部キー嫌い）**」アンチパターンの兆候があることを示しています。

「あるテーブルには存在するけど、もう1つのテーブルには存在しない値を調べるクエリは、どう書けばいい？」

親が更新または削除され、孤児になった子の行を特定しようとしていると考えられます。

「あるテーブルの値が、別のテーブルへの挿入に使われていることを簡単にチェックする方法はある？」

親の行が存在していることを確認したいと考えられます。外部キーは親の行の確認を自動的に実行します。しかも、親テーブルのインデックスを自動的に使うことで、できる限り効率的に確認を行えます。

「外部キー？ データベースの実行速度を遅くするから、使ってはならないと教わったのだけど」

パフォーマンスは、外部キーの省略を正当化する理由としてよく挙げられます。しかし実際には、外部キーの省略によって、様々な問題が生じることも多いのです。その問題には、パフォーマンスの問題も含まれています。

4.4 アンチパターンを用いてもよい場合

ときには、外部キー制約をサポートしていないデータベース製品（MySQLのMyISAMストレージエンジンや、バージョン3.6.19未満のSQLite）を使わざるを得ない場合があります。そういう場合には、章の冒頭で紹介したデータ品質管理スクリプトのように、代わりの方法を見つけなくてはなりません。

また、外部キーを用いた関連付けを行えない、極端に柔軟なデータベース設計を扱わなければならない場

†2 監訳注：「英語圏では、ジレンマ、パラドキシカルな状況を、「キャッチ=22」「キャッチ=22的状況」と呼ぶことがあり、いくつかの辞書では慣用句のひとつとしている」（http://ja.wikipedia.org/wiki/キャッチ=22より抜粋）

合もあります。従来型の参照整合性制約を使えない場合、他のSQLアンチパターンに陥る可能性が高まります。この点についての詳細は、5章「EAV（エンティティ・アトリビュート・バリュー）」と6章「ポリモーフィック関連」を参照してください。

4.5 解決策：外部キー制約を宣言する

ポカヨケ（poka-yoke）とは、「ミス防止」を意味する、日本生まれの言葉です[†3]。この用語は、欠陥が発見されたときに、その原因となった人為的ミスを防止・是正したり、気付きやすくしたりすることで、製品の欠陥を根本から撲滅していく製造プロセスのことを意味しています。品質が向上し、修正の必要性を低下させるため、ポカヨケは投じた費用以上の効果が得られるとされます。

外部キー制約による参照整合性の強制によって、このポカヨケの原則をデータベース設計にも適用できます。データ不整合を検出してから修正するのではなく、データベースへの登録時点でこれらのミスを阻止できるのです。

Keyless-Entry/soln/foreign-keys.sql

```
CREATE TABLE Bugs (
  -- 他の列...
  reported_by       BIGINT UNSIGNED NOT NULL,
  status            VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

既存のコードも、その場限りのアドホックなクエリも、同じ外部キー制約に従います。このため、うっかり見逃していたコードが、この制約の強制を逃れることは**ありません**。データベースは、誰が変更を行ったかにかかわらず、妥当でないすべての変更を拒否します。

外部キーは、不要なコードを書く手間を省き、データベースを変更した場合でもすべてのコードが同じように適切に動作することを保証します。コードの作成だけでなく、修正や保守の時間も大幅に削減してくれます。ソフトウェアの世界では、平均するとコード1,000行ごとに15から50のバグが生じると言われています。それを基準として考えるならば、コードの行数が少なければ、バグも少なくなると言えるでしょう。

4.5.1 複数テーブルの変更をサポートする

外部キーには、アプリケーションコードには真似のできない機能もあります。それは**カスケード更新**と呼ばれる機能です。

Keyless-Entry/soln/cascade.sql

```
CREATE TABLE Bugs (
  -- 他の列...
```

†3 ポカヨケのコンセプトは、技術者の新郷重夫氏が、トヨタ自動車の製造システムについての自らの研究をベースにして構築したものです。

```
  reported_by       BIGINT UNSIGNED NOT NULL,
  status            VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
    ON UPDATE CASCADE
    ON DELETE RESTRICT,
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
    ON DELETE SET DEFAULT
);
```

カスケード更新を使うと、親の行の更新や削除が可能になり、さらにその行を参照しているあらゆる子の行もデータベースが適切に処理してくれるようになります。例えば、親テーブルのBugStatusと Accountsを更新すると、Bugsの子の行に自動的に反映されます。もう、「キャッチ=22問題」で頭を悩ます必要はありません。

外部キー制約にON UPDATE句や ON DELETE句を宣言することで、カスケード処理の内容を定義できます。このサンプルのように、reported_byの外部キーに RESTRICTを宣言すると、Bugs列が参照しているアカウントを削除できなくなります。外部キー制約が削除を阻止し、エラーを発生させます。一方、BugStatusの status値を削除した場合には、SET DEFAULTの効果によって、その statusを持つすべてのバグは自動的にデフォルトのステータス値にリセットされます。

どちらの場合でも、データベースは両方のテーブルをアトミックに変更します。このため、外部キー参照の整合性は、変更の前後で維持されます。

カスケード処理の振舞いは子テーブル側の外部キー上で定義するので、データベースに新しい子テーブルを加える場合でも、アプリケーションコードの変更は不要です。また、参照している子テーブルの数にかかわらず、親テーブルの定義変更も不要です。

4.5.2 オーバーヘッド、……にはなりません

外部キー制約によって、多少のオーバーヘッドが生じるのは事実です。しかし、以下に挙げるように、他の選択肢と比べると、外部キーの方がより効率的であることがわかります。

- 挿入、更新、削除の前に、チェックのためにSELECTクエリを実行する必要がありません。
- 複数テーブルの変更を防ぐために、テーブルをロックする必要がありません。
- 他の方法のように孤児が生じてしまうことがないので、データ品質管理用スクリプトを定期的に実行する必要がありません。

外部キーは簡単に使え、パフォーマンスも改善します。また、単純か複雑かを問わず、あらゆるデータの変更における参照整合性の維持に役立ちます。

> データベースでのミスの発生を未然に防ぐために、外部キー制約を用いましょう。

5章
EAV（エンティティ・アトリビュート・バリュー）

猫のからだをバラバラにして、その仕組みを調べようとしても、手にするものは動かない猫だけだ。
——ダグラス・アダムズ（イギリスのSF作家・脚本家）

日付ごとの行数を計算するようなタスクは、データベースプログラマーにとって、お馴染みのシンプルなタスクです。答えは、SQLの入門書でも必ずといっていいほど紹介されています。以下のような基本的なSQL構文を使うというものです。

EAV/intro/count.sql

```
SELECT date_reported, COUNT(*)
FROM Bugs
GROUP BY date_reported;
```

しかし、この単純な解決策には、以下の2つの前提条件があります。

- 値が同じ列に格納されていること（例：Bugs.date_reported）
- 値を比較できること（GROUP BYで同じ値の日付を正確にグループ化できるように）

これらの前提条件が成立しない場合があります。日付が行によって違う列に格納されてしまっているとき（date_reported列、report_date列など）や、日付のフォーマットが異なるために、2つの日付を簡単には比較できない場合です。こういうときは、どうすればよいのでしょうか。

これらの問題は、EAV（エンティティ・アトリビュート・バリュー）と呼ばれるアンチパターンを用いているときに遭遇する問題の1つなのです。

5.1 目的：可変属性をサポートする

高い拡張性は、ソフトウェアを作る上での大きな目標です。開発者は、将来の変更に対して、ごくわずかな変更で対応したり、あるいはまったく手を入れずに対応できるような、柔軟なソフトウェアを設計したいと考えています。

拡張性の議論の歴史は長く、**エドガー・F・コッド**が1970年に論文『A Relational Model of Data for Large Shared Data Banks（大規模共有データバンクのデータリレーショナルモデル）』[Cod70]で初めてリレーショナルモデルを提唱して以来、リレーショナルデータベースのメタデータの柔軟性が低いという問題は、ほとんど途切れることなく議論され続けてきました。

従来型のテーブルでは、テーブルのすべての行は同じ属性列を持っています。そして、テーブルの各行が同じ種類のオブジェクトの個別のインスタンスを表します。異なる属性の組み合わせは異なるタイプのオブジェクトを表すため、別のテーブルに属します。

しかし、最近のオブジェクト指向プログラミングモデルでは、例えば、同じ基底型を拡張したそれぞれ異なるオブジェクトタイプとの間にも関連を持つ可能性があります。オブジェクト指向設計では、同じ基底型を拡張したオブジェクトは、同じ基底型のインスタンスであり、かつ各サブタイプのインスタンスでもあると見なされます。開発者は、そういったオブジェクトたちを1つのテーブルに行として格納することで、比較と計算を単純化したいと考えます。しかし、各サブタイプのオブジェクトの属性列も同じように格納する必要があります。サブタイプの属性は、基底型や他のサブタイプには存在しない場合があります。

バグデータベースの例を使って説明しましょう。図5-1では、Bugと FeatureRequest（機能要望）が、基底型であるIssue（問題）で共通の属性を共有しています。各Issueは報告者と関連付けられます。また、Issueは製品とも関連を持っており、作業優先度の属性もあります。しかし、Bugには独自の属性もあります。バグが発生した製品のバージョンや、バグの重大度などです。同じく、FeatureRequestも独自の属性を持つ可能性があります。例えば、機能要望の開発に出資しているスポンサーです。

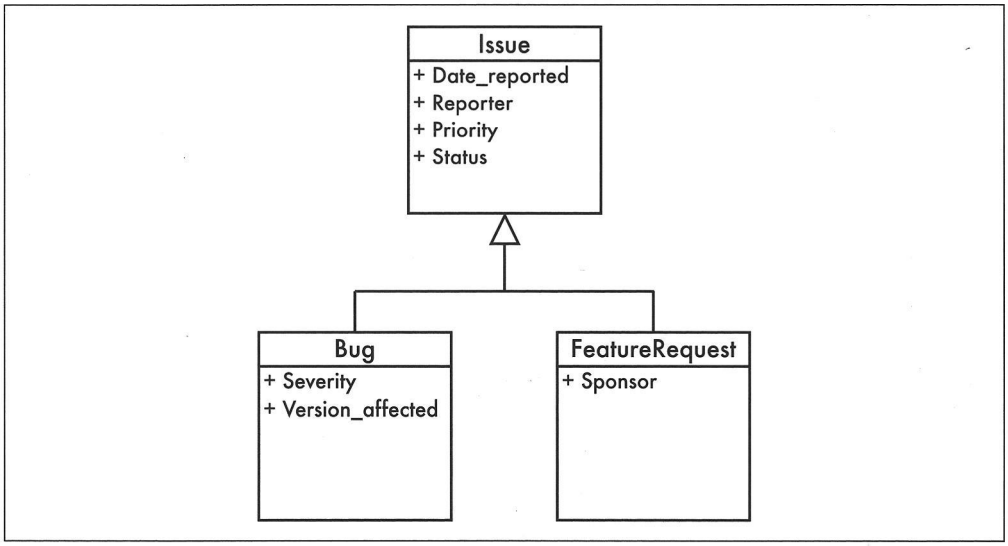


図5-1 Issueに関するクラス図（UML）

5.2 アンチパターン：汎用的な属性テーブルを使用する

可変属性をサポートする必要があるとき、プログラマーにとって魅力的な解決策と思えるのは、もう1つ別のテーブルを作成して、属性を「行」に格納することです。図5-2を見てください。2つのテーブルが示されています。属性テーブル（この例ではIssueAttributes）の各行には、3つの列があります。

エンティティ

通常の場合、親テーブルに対応する外部キーです。親テーブルの方ではエンティティごとに1行が割り当てられています。

属性

従来型のテーブルでは属性は列の名前に相当しますが、この新しい設計では属性名が各行に入っていて、行ごとに識別する必要があります。

値

エンティティの属性の値です。

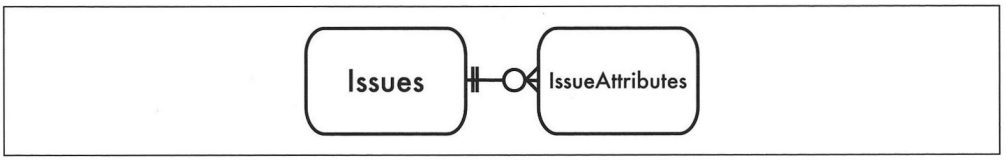


図5-2 EAV設計のER図

EAV/anti/create-eav-table.sql

```
CREATE TABLE Issues (
  issue_id    SERIAL PRIMARY KEY
);

INSERT INTO Issues (issue_id) VALUES (1234);

CREATE TABLE IssueAttributes (
  issue_id    BIGINT UNSIGNED NOT NULL,
  attr_name   VARCHAR(100) NOT NULL,
  attr_value  VARCHAR(100),
  PRIMARY KEY (issue_id, attr_name),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);

INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
  VALUES
    (1234, 'product',          '1'),
    (1234, 'date_reported',    '2009-06-01'),
```

```
(1234, 'status',           'NEW'),
(1234, 'description',      ' 保存処理に失敗する '),
(1234, 'reported_by',      'Bill'),
(1234, 'version_affected', '1.0'),
(1234, 'severity',         ' 機能の損失 '),
(1234, 'priority',         'HIGH');
```

例を挙げましょう。あるバグは、その主キー値1234によって識別されるエンティティです。これはstatusと呼ばれる属性を持ち、バグ 1234の status属性の値は「NEW」です。

このような設計は、**エンティティ・アトリビュート・バリュー**（Entity-Attribute-Value）、略して「EAV」と呼ばれます。**オープンスキーマ、スキーマレス、名前 / 値ペア**と呼ばれることもあります。

テーブルを新たに作成することで、以下のメリットがあると期待されます。

- 両方のテーブルの列数を減らせます。
- 新たな属性をサポートするために、列数を増やす必要がありません。
- 属性が存在しないエンティティの該当列にNULLが入っている、NULLだらけのテーブルになることを防げます。

一見すると、設計が改善されたように思えます。しかし、このようにデータベース構造を単純化しても、処理の複雑さは解消できないのです。

5.2.1 属性を取得するにはどうするか

あなたの上司は、報告されたバグを日別にまとめるレポートを作成する必要があります。従来型のテーブル設計では、Issuesテーブルは date_reported（報告日）のような単純な属性列を持ちます。報告日別にすべてのバグを検索するために、上司は以下のような単純なクエリを使えます。

EAV/anti/query-plain.sql

```
SELECT issue_id, date_reported FROM Issues;
```

EAV設計で上のクエリと同じ情報を取得するには、上司はIssueAttributesテーブルから文字列date_reportedが格納された行をフェッチする必要があります。このクエリは先程より冗長になり、明確さも低下しています。

EAV/anti/query-eav.sql

```
SELECT issue_id, attr_value AS date_reported
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

5.2.2 データ整合性をどう保つか

EAVを使うと、従来型のデータベース設計で得られるいくつもの利点を失ってしまいます。

必須属性を設定できない

上司が正確なプロジェクト報告書を作成するためには、date_reported属性に値が入っていることが必要です。従来型のデータベース設計では、NOT NULL制約を宣言するだけで、その列を必須にできます。

しかしEAV設計では、属性は列ではなく、IssueAttributesテーブルの行に格納されています。attr_name列に文字列date_reportedが入っている行がissue_idごとに存在することを保証するような制約が欲しくなるでしょう。

ところが、SQLはそのような制約をサポートしていないため、行の存在を保証するためにアプリケーションコードを書かなくてはいけません。もしチェックの結果、対応するdate_reported属性の行のないIssueが見つかった場合はどうすれば良いでしょうか。IssueAttributesテーブルにdate_reported属性の行を追加すべきでしょうか。その場合、どんな値を設定すればよいでしょうか。何らかのデフォルト値を使う場合、上司のレポートの精度にどのような影響が出るでしょうか。

SQLのデータ型を使えない

上司は、うまくレポートを作成できずに困っていました。原因は、日付が異なるフォーマットで入力されたり、日付形式ですらない文字列で入力されることが原因でした。従来型のデータベースでは、DATE型で列を定義することで、この問題を防げます。

EAV/anti/insert-plain.sql

```
INSERT INTO Issues (..., date_reported, ...) VALUES (..., 'banana', ...); -- エラー！
```

一方、通常のEAV設計では、IssueAttributes.attr_value列のデータ型を文字列型にします。1つの列にあらゆる種類の属性を格納できるようにするためです。このため、無効なデータを拒否する方法がありません。

EAV/anti/insert-eav.sql

```
INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
 VALUES (1234, 'date_reported', 'banana'); -- エラーにはなりません！
```

EAV設計をさらに拡張して、それぞれのSQLデータ型のために、個別のattr_value列を定義することもあります。様々なデータ型を使えるようにはなりますが、クエリは複雑化してしまいます。

EAV/anti/data-types.sql

```
SELECT issue_id, COALESCE(attr_value_date, attr_value_datetime,
  attr_value_integer, attr_value_numeric, attr_value_float,
  attr_value_string, attr_value_text) AS date_reported
```

```
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

ユーザー定義データ型（UDT）やドメイン（DOMAIN）などをサポートしたければ、さらに列を追加しなければなりません。

参照整合性を強制できない

従来型のデータベースでは、参照テーブルに対する外部キーを定義することによって、一部の属性の値を制限できます。例えば、Bugや Issueの status属性の値は、BugStatusテーブルが格納している値のうちの1つであることを強制できます。

EAV/anti/foreign-key-plain.sql

```
CREATE TABLE Issues (
  issue_id        SERIAL PRIMARY KEY,
  -- 他の列 . . .
  status          VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

EAV 設計では、attr_value列にこの種の制約を設定できません。**参照整合性制約**は、テーブルのすべての行に適用されてしまうからです。

EAV/anti/foreign-key-eav.sql

```
CREATE TABLE IssueAttributes (
  issue_id        BIGINT UNSIGNED NOT NULL,
  attr_name       VARCHAR(100) NOT NULL,
  attr_value      VARCHAR(100),
  FOREIGN KEY (attr_value) REFERENCES BugStatus(status)
);
```

この制約を定義すると、status属性のみではなく、「すべての」属性が BugStatusの値と一致しなければならなくなってしまいます。

属性名を補わなければならない

件の上司のレポートは、まだ不正確なようです。あなたは、属性名の文字列に一貫性がないことに気づきました。あるバグは date_reported と名付けられた文字列を、別のバグは report_date という文字列を使用しています。明らかに、両方とも同じ情報を表すよう意図されたものです。

では、日付を基準にしてバグを数えるにはどうすればよいのでしょうか。

EAV/anti/count.sql

```
SELECT date_reported, COUNT(*) AS bugs_per_date
FROM (SELECT DISTINCT issue_id, attr_value AS date_reported
  FROM IssueAttributes
  WHERE attr_name IN ('date_reported', 'report_date'))
GROUP BY date_reported;
```

しかし、バグがさらに別の名前で属性を格納している場合、どうやってそれを確認すればよいのでしょうか。また、1つのバグが別々の名前で、2回同じ属性を格納することも考えられます。このようなミスを防ぐにはどうすればよいのでしょうか。

1つの方法として、attr_name列に、あらかじめ定義した属性名を格納する参照テーブルへの外部キーを定義することが考えられます。しかし、この方法では、各エンティティに随時その場で新たに定義される属性をサポートできません。EAV設計を使いたい場合、実行時に動的に属性が増えることが一般的なのです。

5.2.3 行を再構築しなければならない

従来型の設計であれば、Issuesテーブルから1行取得すると、その行にすべての属性が列として格納されています（表5-1）。EAV設計でも、1つのIssueを、あたかも従来型のテーブルに格納されているものであるかのように、1つの行として取得したいところです。

表5-1 従来型の設計における結果行

issue_id	date_reported	status	priority	description
1234	2009-06-01	NEW	HIGH	保存処理に失敗する

EAV設計では、各属性がIssueAttributesテーブルの個別の行に格納されるため、すべての属性を行の一部分として取得するには、各属性の行のJOINが必要になります。クエリの作成時には、属性の名前をすべて指定しなければなりません。以下のクエリは、前述の行を再構築したものです。

EAV/anti/reconstruct.sql

```
SELECT i.issue_id,
  i1.attr_value AS date_reported,
  i2.attr_value AS status,
  i3.attr_value AS priority,
  i4.attr_value AS description
FROM Issues AS i
  LEFT OUTER JOIN IssueAttributes AS i1
    ON i.issue_id = i1.issue_id AND i1.attr_name = 'date_reported'
  LEFT OUTER JOIN IssueAttributes AS i2
    ON i.issue_id = i2.issue_id AND i2.attr_name = 'status'
  LEFT OUTER JOIN IssueAttributes AS i3
    ON i.issue_id = i3.issue_id AND i3.attr_name = 'priority'
```

```
  LEFT OUTER JOIN IssueAttributes AS i4
    ON i.issue_id = i4.issue_id AND i4.attr_name = 'description'
WHERE i.issue_id = 1234;
```

属性の1つがIssueAttributesテーブルに行として存在していない場合、**内部結合**（INNER JOIN）を行うと結果がゼロ行になってしまうため、**外部結合**（OUTER JOIN）を使わなくてはいけません。属性の数が増加すると、結合の数も増加し、このクエリの実行コストも指数関数的に増加してしまいます。

5.3 アンチパターンの見つけ方

プロジェクトチームで以下のような言葉を耳にしたら、「EAV（エンティティ・アトリビュート・バリュー）」アンチパターンが採用されている可能性があります。

「このデータベースは、メタデータの変更なしで拡張可能だ。実行時に新しい属性を定義できる」

リレーショナルデータベースは、この類の柔軟性をサポートしていません。任意に拡張できるデータベースを設計したと誰かが主張するなら、それはおそらくEAV設計を用いているからです。

「クエリは、結合を最大いくつまでサポートしているんだっけ？」

データベースの制限を越えるかもしれないと心配するほどの大量の結合を行わなければならないのなら、おそらくそのデータベース設計には問題があります。EAV設計では、この種の問題に頻繁に直面します。

「Eコマースプラットフォームのレポート生成を行うコードの書き方がわからない。コンサルタントを雇わなくてはいけないな」

導入時の初期設定が不要で、裏でデータベースを使っているようなソフトウェアパッケージの多くは、カスタマイズしやすいように、EAV設計を採用しています。このため、レポート作成のような基本的な処理のためのクエリが複雑化し、ときには実現できなくなったりします。

5.4 アンチパターンを用いてもよい場合

リレーショナルデータベースでEAV（エンティティ・アトリビュート・バリュー）アンチパターンの使用を正当化する理由は、簡単には見つかりません。本章のアンチパターンの節で見てきたように、リレーショナルデータベースの多くの長所が失われてしまうからです。しかも、動的な属性のサポートを必要とするアプリケーションの真のニーズにも対応できません。

ほとんどの場合、スキーマレスなデータを必要とするアプリケーションで求められているのは、**ごく少数の（あるいは1つの）テーブルのみ**です。残りの要件は、標準的なテーブル設計で対処できます。プロジェクトを計画する際、EAVの採用による作業の増加やリスクに責任が持てるのなら、消去法的にEAVを選択してもよいでしょう。しかし、経験豊富なデータベースコンサルタントは口を揃えて、「EAVを使ったシステムは1年と経たずに扱いにくくなる」と話しています。

非リレーショナルなデータ管理が必要なら、非リレーショナルな技術を使うべきです。とはいえ、本書は

SQLについての本であり、残念ながらSQLの代替技術についての詳細を説明するスペースはありません。以下にこれらの技術の一部を紹介しますので、興味のある方は自分で調べてみてください。

Berkeley DB
: アプリケーションへの組み込みが容易な、人気の高いKey/Value式のデータベースライブラリです。
http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html

Cassandra
: Facebookで開発された列指向の分散データベースであり、現在ではApacheプロジェクトに寄贈されています。
http://cassandra.apache.org/

CouchDB
: ドキュメント指向のデータベースです。分散型のKey/Valueを格納し、値をJSON形式でエンコードします。
http://couchdb.apache.org/

Hadoop／HBase
: Googleの MapReduceアルゴリズムを実装したオープンソース分散データベースです。大規模な半構造化データストアに対して分散クエリを実行できます。
http://hadoop.apache.org/

MongoDB
: CouchDBと類似した、ドキュメント指向のデータベースです。
http://www.mongodb.org/

Redis
: ドキュメント指向のインメモリーデータベースです。
http://redis.io/

Tokyo Cabinet
: Key/Value型のデータストアで、POSIX DBM、GNU GDBM、Berkeley DBなどに対応しています[†1]。
http://fallabs.com/tokyocabinet/

他にも、多くの非リレーショナル型のプロジェクトが次々と誕生しています。しかし、本来のリレーショナルデータベース設計との比較で見られるEAVの弱点は、これらの代替手法にも当てはまります。すなわち、メタデータが流動的であると、シンプルなクエリの作成が難しくなります。アプリケーションでは、データの構造の解析と処理に多くの労力を要することになってしまいます。

†1 監訳注：Tokyo Cabinet作者の平林幹雄氏は後継プロダクトのKyoto Cabinet（http://fallabs.com/kyotocabinet/）の使用を勧めています。

5.5 解決策：サブタイプのモデリングを行う

EAVの採用が正解だと思っても、実装の前にじっくり再考してみましょう。これまで慣れ親しんできたセオリー通りの考えに従えば、従来型のテーブル設計でも、より簡単に、かつデータの整合性を維持しながら、プロジェクトのデータモデルを設計できることがわかるはずです。

EAVを使わずに、EAVが扱うようなデータを格納する方法はいくつかあります。解決策がうまく機能するのは、**サブタイプの数が限られ、開発者が各サブタイプの属性をよく知っているとき**です。どの解決策を採用するかは、データをどのように検索したいかによるので、状況に応じて適切な設計を選択しましょう。

5.5.1 シングルテーブル継承

最もシンプルな設計は、すべてのタイプの属性を個別の列に格納して、関連するすべてのサブタイプを1つのテーブルに格納することです。加えて1つの属性列を、その行がどのサブタイプであるかを定義するために用います。この例では、サブタイプ識別に使う属性はissue_typeです。属性によっては、すべてのサブタイプに共通のものがありますが、多くの属性はサブタイプ固有のものです。対応する属性を持たないオブジェクトを格納する行には、対応する属性列にNULLを入れなくてはなりません。このため、テーブルには非NULLの値を持つ列がバラバラに点在することになります。

シングルテーブル継承（Single Table Inheritance）という名前は、マーティン・ファウラーの著書『エンタープライズアプリケーションアーキテクチャパターン』[Fow03]に由来しています[†2]。

EAV/soln/create-sti-table.sql

```
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  issue_type       VARCHAR(10), -- 'BUG' または 'FEATURE' が格納される
  severity         VARCHAR(20), -- Bug のみが使う属性
  version_affected VARCHAR(20), -- Bug のみが使う属性
  sponsor          VARCHAR(50), -- FeatureRequest のみが使う属性
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

新しいオブジェクトタイプを追加する際には、データベースは、その新しいオブジェクトタイプの属性を格納できなければなりません。新しいオブジェクトタイプの属性が既存のテーブルに存在しない場合、テーブル定義を変更して列を追加しなければならないため、テーブルによっては列数の実際的な上限に達するか

†2 監訳注：シングルテーブル継承は「Single Table Inheritance」の頭文字を取って「STI」と呼ばれることもあります。Ruby on RailsのSTIサポート機能では、サブタイプ識別列名のデフォルトはtypeで、モデルのクラス名（'Bug'や'FeatureRequest'）が格納されます。

もしれません。

シングルテーブル継承には、どの属性がどのサブタイプに所属するかを定義するメタデータが無いという欠点もあります。アプリケーションは、ある行において、その行に対応するサブタイプに無関係な属性を無視できます。しかし、属性とサブタイプの対応関係は、開発者が自前で管理しなければなりません。メタデータを使えるのであれば、それは本来データベースで管理したほうがよい情報です。

シングルテーブル継承の採用が適切なのは、サブタイプの数とサブタイプ固有の属性の数が少なく、**アクティブレコード**（Active Record）のような単一のテーブルに対するデータベースアクセスパターンを使う必要がある場合です[†3]。

5.5.2 具象テーブル継承

2番目の解決策は、サブタイプごとにテーブルを作成するというものです。すべてのテーブルは、基底型に共通する属性と、それぞれのサブタイプに固有の属性を含んでいます。**具象テーブル継承**（Concrete Table Inheritance）という名前も、同じくマーティン・ファウラーの書籍 [Fow03]に由来します。

EAV/soln/create-concrete-tables.sql

```
CREATE TABLE Bugs (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  severity         VARCHAR(20), -- Bug のみが使う属性
  version_affected VARCHAR(20), -- Bug のみが使う属性
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE FeatureRequests (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  sponsor          VARCHAR(50), -- FeatureRequest のみが使う属性
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

†3 監訳注：アクティブレコード（Active Record）は、前述の『エンタープライズアプリケーションアーキテクチャパターン』[Fow03]で紹介されているパターン名です。

シングルテーブル継承と比較した場合の具象テーブル継承のメリットとして、サブタイプに存在しない属性列を格納する必要がないという点が挙げられます。対象のテーブルに存在しない属性列を参照する場合、データベースは自動的にエラーを返します。例えば、severity列は FeatureRequestsテーブルには存在しません。

EAV/soln/insert-concrete.sql

```
INSERT INTO FeatureRequests (..., issue_id, severity, ...) VALUES ( ... ); -- エラー！
```

具象テーブル継承には、シングルテーブル継承とは異なり、行ごとのサブタイプを表す列（この章の例ではissue_type列）が不要になるというメリットもあります。

しかし、すべてのサブタイプに共通する属性とサブタイプ固有の属性の区別は簡単ではありません。また、共通属性に新しい属性を加える場合、すべてのサブタイプのテーブルを変更しなければなりません。

サブタイプテーブルに格納されたデータが、基底型とサブタイプのどちらに属しているかを示すメタデータはありません。このため、例えばプロジェクトに新しく加わったプログラマーがテーブル定義を見たとき、一部の列がすべてのサブタイプテーブルで共通であることはわかりますが、そこに論理的な関係性が存在するのか、それとも偶然テーブルに類似性があるだけなのかを、メタデータによって判断することはできません。

すべてのオブジェクトを、どのサブタイプであるかにかかわらず取得したい場合には、各サブタイプが別々のテーブルに格納されているために、処理は複雑になります。このクエリを簡単にするために、共通属性のみを選択し、テーブルをUNIONしたビューを定義するという手段があります。

EAV/soln/view-concrete.sql

```
CREATE VIEW Issues AS
  SELECT b.issue_id, b.reported_by, ... 'BUG' AS issue_type
  FROM Bugs AS b
    UNION ALL
  SELECT f.issue_id, f.reported_by, ... 'FEATURE' AS issue_type
  FROM FeatureRequests AS f;
```

具象テーブル継承設計は、すべてのサブタイプをまたいだ検索を実行する頻度が低い場合に適切です。

5.5.3 クラステーブル継承

3番目の解決策は、テーブルをオブジェクト指向のクラスであるかのように見なして、継承を摸倣するという方法です。まず、すべてのサブタイプに共通する属性を含む基底型のテーブルを1つ作ります。次に、サブタイプごとに1つずつ追加のテーブルを作成し、基底型テーブルに対する外部キーの役割を持つ主キーを設定します。**クラステーブル継承**（Class Table Inheritance）という名前も、マーティン・ファウラーの書籍 [Fow03]に由来しています。

EAV/soln/create-class-tables.sql

```
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE Bugs (
  issue_id         BIGINT UNSIGNED PRIMARY KEY,
  severity         VARCHAR(20),
  version_affected VARCHAR(20),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);

CREATE TABLE FeatureRequests (
  issue_id         BIGINT UNSIGNED PRIMARY KEY,
  sponsor          VARCHAR(50),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
```

メタデータによって、1対1の関連が強制されます。子側のテーブルの外部キーは主キーでもあり、すなわち一意であるからです。この解決策では、基底型の属性のみを参照する限り、すべてのサブタイプにまたがる検索を効率良く行えます。検索条件と一致するエントリを基底テーブルから取得した後、各サブタイプテーブルに対してクエリを実行することで、サブタイプ固有の属性を取得できます。

基底テーブルの行から、行がどのサブタイプを表すかを特定する必要はありません。サブタイプの数が少ない限り、これらをすべてJOINするクエリを書けるからです。その結果、シングルテーブル継承のテーブルと同じような、NULLが点在する結果集合が得られます。つまり、サブタイプに存在しない属性はNULLになります。

EAV/soln/select-class.sql

```
SELECT i.*, b.*, f.*
FROM Issues AS i
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id);
```

サブタイプすべてをJOINするクエリは、ビュー定義の候補にもなります。

クラステーブル継承は、すべてのサブタイプに共通する列を参照するクエリが頻繁に実行されるときに適

しています。

5.5.4 半構造化データ

サブタイプの数が多い場合や、頻繁に新しい属性を追加しなければならない場合は、LOB列[†4]を追加し、XMLやJSONなどの形式で属性名と値を共に格納することもできます。マーティン・ファウラーは、著書[Fow03]でこのパターンを**シリアライズLOB（Serialized LOB）**と呼んでいます。

EAV/soln/create-blob-tables.sql

```
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  issue_type       VARCHAR(10), -- 'BUG' または 'FEATURE' が格納される
  attributes       TEXT NOT NULL, -- その他の動的属性が格納される
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

この設計の長所は、拡張性が極めて高いという点です。新しい属性は、いつでもLOB列に格納できます。このテーブルの各行には、まったく異なる属性群を格納することができるため、すべての行ごとに異なるサブタイプを作ることも可能です。

この設計の短所は、このような構造においてSQLが特定の属性にアクセスする手段をほとんど持っていないという点です。LOB列では、行の絞り込み、集約計算、ソートなどの処理のために LOB内の個別の属性を選択することが難しいため、まずLOB全体を1つの値として取得し、属性に分解して解釈するアプリケーションコードを書かなくてはいけません。

この設計は、サブタイプの数を制限できない場合や、新しい属性を随時定義するための高い柔軟性が必要な場合に適しています。

5.5.5 後処理

残念ながら、どうしてもEAVを使わざるを得ない状況があります。プロジェクトを引き継いだ場合や、EAVを使用するサードパーティ製のソフトウェアプラットフォームを会社が採用している場合などです。その場合は、本章の「アンチパターン」の節で紹介した問題が起こる可能性があることを忘れないようにしましょう。また、EAV設計のために追加の作業が必要になることも、あらかじめ想定しておきましょう。

特に気をつけたいのは、データが従来型のテーブルに格納されているかのように、単一行としてエンティティを取得するクエリを書いてしまうことです。そうではなく、特定のエンティティと関連付けられた属性

†4 監訳注：Large Object 型の総称。BLOB型、 CLOB型、 TEXT型などが LOBに該当します。

をクエリの対象とし、それらが格納されているのと同じかたち、つまり行の集合として取得するようにしましょう。

EAV/soln/post-process.sql

```
SELECT issue_id, attr_name, attr_value
FROM IssueAttributes
WHERE issue_id = 1234;
```

このクエリの結果は、表5-2のようになります。

表5-2 EAV属性を行の集合として取得する

issue_id	attr_name	attr_value
1234	date_reported	2009-06-01
1234	description	保存処理に失敗する
1234	priority	HIGH
1234	product	Open RoundFile
1234	reported_by	Bill
1234	severity	機能の損失
1234	status	NEW

このクエリは、ユーザーにとって作りやすく、データベースにとって処理しやすいクエリです。クエリの作成時に属性の数を知らなくても、Issueに関連付けられた属性がすべて返されます。

この形式で結果を扱うには、アプリケーションで結果行をループするコードを書き、オブジェクトのプロパティとして設定する必要があります。以下は、PHPの場合のコードサンプルです。

EAV/soln/post-process.php

```
<?php

$objects = array();
$stmt = $pdo->query(
    "SELECT issue_id, attr_name, attr_value
     FROM IssueAttributes
     WHERE issue_id = 1234");
while ($row = $stmt->fetch()) {
  $id    = $row['issue_id'];
  $field = $row['attr_name'];
  $value = $row['attr_value'];
  if (!array_key_exists($id, $objects)) {
    $objects[$id] = new stdClass();
  }
  $objects[$id]->$field = $value;
}
```

このように、とても大変な仕事になってしまいます。これが、EAVのような「システム内のシステム」が招く結果なのです。SQLでは、異なる属性は異なる列に格納するという方法が確立されています。EAVを使うことは、属性を識別するための新たな方法のレイヤーをSQLの上に構築することを意味します。SQLがEAVの処理をぎこちなく非効率にしか行えないのも、無理はないのです。

メタデータは、メタデータのために用いましょう。

6章
ポリモーフィック関連

「どっちの道に進めばいいのかしら？」ドロシーがつぶやくと、かかしが答えた。「どっちでもいいのさ」
——映画『オズの魔法使い』

バグデータベースに、ユーザーがバグについてのコメントを書き込める機能を追加することにします。バグによっては、多くのコメントが書き込まれることがあるかもしれません。また、どのコメントも必ず、いずれか1つのバグへ関連付かなければなりません。つまり、Bugsと Commentsの間には、1対多の関連が存在しています。図6-1に、この種の単純な関連を持つER図を示します。このテーブルを作成するSQLは、以下のようになります。

Polymorphic/intro/comments.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author_id    BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

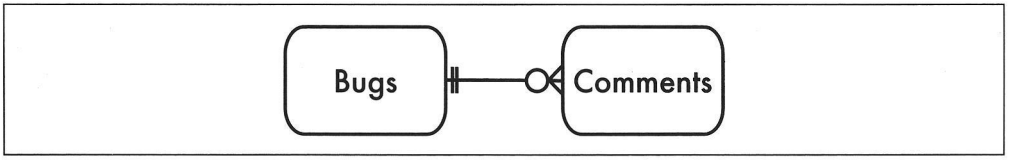


図6-1　単純な関連付け

しかし、コメントを記入できるテーブルは、2つあるかもしれません。例えば、Bugsと FeatureRequests

です。この2つは類似したエンティティですが、それぞれ別のテーブルに格納されています（5章「EAV（エンティティ・アトリビュート・バリュー）」の「具象テーブル継承」の項を参照）。コメント自体は、それがIssueのどちらのタイプ（BugsまたはFeatureRequests）に属するかにかかわらず、1つのテーブル（Comments）に格納したいところです。しかし、複数の親テーブルを参照する外部キーを宣言することはできません。以下のような宣言は、ナンセンスです。

Polymorphic/intro/nonsense.sql

```
  ...
  FOREIGN KEY (issue_id)
    REFERENCES Bugs(issue_id) OR FeatureRequests(issue_id)
);
```

複数のテーブルへのクエリ実行のために、以下のような無効なSQLを書く開発者もいます。

Polymorphic/intro/nonsense.sql

```
SELECT c.*, i.summary, i.status
FROM Comments AS c
INNER JOIN c.issue_type AS i USING (issue_id);
```

しかし、行ごとに異なるテーブルを結合するSQLを書くことはできません。SQL構文では、クエリの実行時にすべてのテーブル名を指定しなければなりません。クエリの実行中に、テーブルは変更できません。なぜ、このような問題が生じてしまったのでしょうか。どうすれば解決できるのでしょうか。

6.1 目的：複数の親テーブルを参照する

映画『オズの魔法使い』のなかで、エメラルドシティを目指して進む主人公のドロシーが、分かれ道にぶつかるシーンがあります。どちらを選べばよいのか迷っていたドロシーは、目の前にいた「かかし」から思いがけないアドバイスをもらって戸惑ってしまいます。かかしは、「どちらに進んでもいい」と答えたのです。

図6-2のER図には、それと同じような関連付けが示されています。子テーブルの外部キーは、ドロシーの目の前にあった道のように分岐しています。このため、Commentsテーブルの行は、FeatureRequestsテーブルの行とBugsテーブルの行のどちらかと一致します。弧線は、排他的な選択が必要であることを示します。つまり、コメントはその参照先を、機能要求（FeatureRequests）またはバグ（Bugs）のどちらかから選ばなければなりません。

6.2 アンチパターン：二重目的の外部キーを使用する

この問題の解決策の1つは、名前を付けられるほど有名なものです。この章のタイトルである、**ポリモーフィック関連**（Polymorphic Associations）です。他に、**プロミスキャス・アソシエーション**（Promiscuous Association：**無差別な関連**）と呼ばれることもあります。複数のテーブルを参照できるからです。

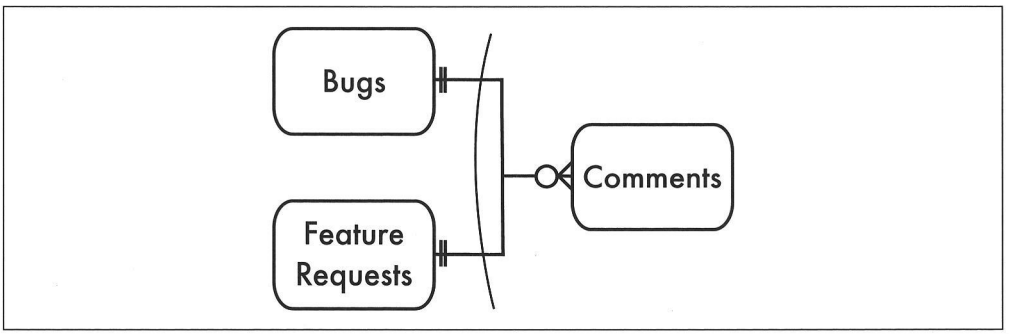


図 6-2 ポリモーフィック関連

6.2.1 ポリモーフィック関連を定義する

ポリモーフィック関連を機能させるには、従来の issue_idのような外部キー列に加えて、文字列型の列を追加します。追加した列には、現在の行が参照する親テーブルの名前を格納します。この例では、issue_type列が新規に追加されます。issue_type列には、この関連付けで対象となる 2 つの親テーブルの名前に応じて、文字列「Bugs」または「FeatureRequests」が格納されます。

Polymorphic/anti/comments.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  issue_type   VARCHAR(20), -- 'Bugs' または 'FeatureRequests' が格納される
  issue_id     BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment      TEXT,
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

1 つ、気づいた点があるのではないでしょうか。そうです、issue_idのための外部キー宣言がありません。**外部キー**では、テーブルを 1 つのみ指定しなければならない（複数のテーブルを指定できない）ため、ポリモーフィック関連を使用しているときは、メタデータで関連付けを宣言できないのです。その結果、Comments.issue_idの値が親テーブルの値と一致することを保証するための **参照整合性制約**を定義できません。

同様に、Comments.issue_type内の文字列データが、データベースに存在するテーブル名に対応していることを保証するメタデータもありません。

データへのメタデータの混入

ポリモーフィック関連アンチパターンには、前章のEAV（エンティティ・アトリビュート・バリュー）アンチパターンと似た特徴があると気付いた人もいるかもしれません。どちらのアンチパターンでも、メタデータオブジェクトの名前が文字列として格納されるのです。EAVでは、属性の列名が文字列としてattr_name列に格納されます。ポリモーフィック関連では、親テーブルの名前が文字列としてissue_type列に格納されます。これは、「**データへのメタデータの混入**」とでも呼ぶべきものです。このコンセプトは、7章「マルチカラムアトリビュート（複数列属性）」でも別の形で現れます。

6.2.2 ポリモーフィック関連へのクエリ実行

Commentsテーブルのissue_idの値は、親テーブルであるBugsとFeatureRequestsの両方の主キー列に存在する場合があります。または、1つの親テーブルには存在するが、もう1つの親テーブルには存在しない場合もあります。このため、子テーブルを親テーブルと結合するとき、正確にissue_typeを使うことが極めて重要です。Bugsテーブルと結合するために格納したissue_idでFeatureRequestsテーブルと結合してはいけません。

例えば、主キー値1234のバグのコメントを取得する場合、以下のように記述します。

Polymorphic/anti/select.sql

```
SELECT *
FROM Bugs AS b INNER JOIN Comments AS c
  ON b.issue_id = c.issue_id AND c.issue_type = 'Bugs'
WHERE b.issue_id = 1234;
```

このクエリは、コメントを付ける対象がBugsテーブルのみに格納されている場合に機能しますが、Commentsが Bugsと FeatureRequestsの両方に関連付けられている場合は、問題に直面します。SQLでJOINを行うには、あらかじめすべてのテーブルを指定しなければなりません。Comment.issue_type列の値に応じて、行単位で切り替えながら、Commentsを2つのテーブルに動的に JOINすることはできないのです。

コメントが付いたバグまたは機能要求を取得するには、両方の親テーブルを**外部結合**（OUTER JOIN）してクエリを実行しなければいけません。しかし、結合条件はComment.issue_type列の値に依存するため、各行で2つのテーブルのうち1つのみしかJOINできません。外部結合を使用するので、結果行の一致しないテーブルの方のフィールドにはNULLが入ります。

Polymorphic/anti/select.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b
    ON b.issue_id = c.issue_id AND c.issue_type = 'Bugs'
  LEFT OUTER JOIN FeatureRequests AS f
```

```
    ON f.issue_id = c.issue_id AND c.issue_type = 'FeatureRequests';
```

結果は、表6-1のようになります。

表6-1 両方の親テーブルを外部結合した結果

c.comment_id	c.issue_type	c.issue_id	c.comment	b.issue_id	f.issue_id
6789	Bugs	1234	クラッシュします	1234	NULL
9876	FeatureRequests	2345	いいアイデア！	NULL	2345

6.2.3 非オブジェクト指向の例

先ほどの例では、これら2つの親テーブル（Bugsと FeatureRequests）は、同じモデルから継承したサブタイプを表していました。ポリモーフィック関連は、親テーブル同士にまったく関係が無い場合にも使われます。例えば、図6-3のEコマース向けのデータベースでは、顧客（Users）と注文（Orders）の2つのテーブルは、住所（Addresses）と関連付けられます。

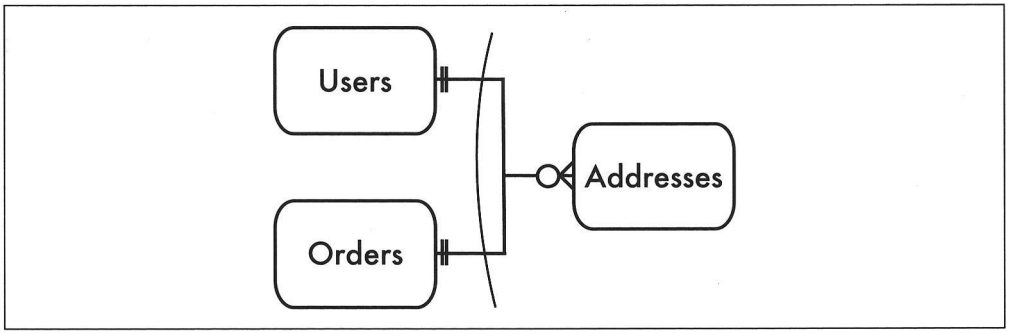


図6-3 アドレスのポリモーフィック関連

Polymorphic/anti/addresses.sql

```
CREATE TABLE Addresses (
  address_id   SERIAL PRIMARY KEY,
  parent       VARCHAR(20), -- 'Users' または 'Orders' が入る
  parent_id    BIGINT UNSIGNED NOT NULL,
  address      TEXT
);
```

Addressesテーブルには、親テーブル Usersまたは Ordersの名前が格納されたポリモーフィック関連が定義されています。Usersと Ordersのどちらか1つを選択しなければならないので、注文商品の出荷先が顧客自身の住所であっても、両方に同じ住所の行を関連付けることはできません。

また、顧客が出荷先住所だけでなく、請求先住所を持つ場合、Addressesテーブルでそれを区別する方法が必要です。他の親テーブルがAddressesテーブルの特別な使い方をする場合にも、用法を定義できるようにしておかなければなりません。このような注意点が次々と出てきます。

Polymorphic/anti/addresses.sql

```
CREATE TABLE Addresses (
  address_id   SERIAL PRIMARY KEY,
  parent       VARCHAR(20), -- 'Users' または 'Orders' が入る
  parent_id    BIGINT UNSIGNED NOT NULL,
  users_usage  VARCHAR(20), -- 'billing' または 'shipping' が入る
  orders_usage VARCHAR(20), -- 'billing' または 'shipping' が入る
  address      TEXT
);
```

6.3 アンチパターンの見つけ方

以下のような言葉を耳にしたら、「**ポリモーフィック関連**」アンチパターンが使われている可能性があります。

「このtaggingスキーマは、タグをデータベース内の『あらゆる』リソースと関連付けられるんだ」

EAVの場合と同じく、無限の柔軟性があるという話は眉唾ものです。

「このデータベース設計では、外部キーは宣言できません」

これも、要注意発言です。外部キーはリレーショナルデータベースの基本的な機能です。適切な参照整合性を伴わない設計は、多くの問題を孕んでいます。

「このentity_type列は、何のためのもの？」「ああ、それはこの列がどのテーブルの値を参照しているかを示すものなんだ」

外部キーでは、すべての行で同じテーブルを参照しなければなりません。

フレームワークによっては、ポリモーフィック関連をサポートしているものがあります。Ruby on Railsは、ActiveRecordの各サブクラスで関連に :polymorphic 属性を宣言することによって、ポリモーフィック関連を実現します。例えば、以下のようにして、Commentsを Bugsと FeatureRequestsに関連付けることができます[†1]。

Polymorphic/recog/commentable.rb

```
class Comment < ActiveRecord::Base
  self.primary_key = 'comment_id'
```

†1 監訳注：Railsのポリモーフィック関連サポートは、タイプ識別列にテーブル名ではなくRailsのモデル名を格納します。つまりこのサンプルの場合には、issue_type列に「Bug」や「FeatureRequest」が格納されます（それぞれ単数形であることに注意してください）。また本書でのテーブル設計はRailsの規約に沿っていないので、主キー名やテーブル名の明示的な宣言のためのコードが増えています。

```
  belongs_to :issue, :polymorphic => true
end

class Bug < ActiveRecord::Base
  self.primary_key = 'issue_id'
  has_many :comments, :as => :issue
end

class FeatureRequest < ActiveRecord::Base
  self.table_name = 'featurerequests'
  self.primary_key = 'issue_id'
  has_many :comments, :as => :issue
end
```

他にも、Javaのフレームワーク Hibernate では、様々なスキーマ宣言によってポリモーフィック関連をサポートします[†2]。

6.4 アンチパターンを用いてもよい場合

なるべくポリモーフィック関連の使用は避け、**外部キー制約**などを用いて、**参照整合性**を保証するようにしましょう。ポリモーフィック関連を使うと、メタデータではなくアプリケーションコードに過度に依存してしまいます。

Hibernate などのオブジェクトリレーショナルマッピング（ORM）フレームワークを用いる場合、このアンチパターンを使わざるを得ない状況があります。このようなフレームワークは、ポリモーフィック関連固有のアプリケーションロジックをカプセル化して参照整合性を維持することで、ポリモーフィック関連の採用によるリスクを低減させています。

成熟した実績あるフレームワークを選べば、ポリモーフィック関連がきちんと実装されていることが期待できるでしょう。しかし、フレームワークを用いずに、手書きでポリモーフィック関連を実装するのは「車輪の再発明（Reinventing the Wheel）」アンチパターン [BMMM98]です。無駄が多く、リスクの高い行為だと言わざるを得ません。

6.4.1 ポリモーフィック関連を意識的に選択するとき[†3]

ポリモーフィック関連を意識的に選択する理由の1つは、「ORMを使った設計を行う際に、子側のテーブルを設計 / 作成する時点でまだ存在しない親テーブルとの関連を作成できる」ことです。

このような状況は、関連の子側のテーブルと親側のテーブルの作成時期が離れているとき、例えばポリモーフィック関連の子側が複数の案件で共通して使われるフレームワークで、親側が個々の案件のアプリケーションである場合や、ポリモーフィック関連の子側がパッケージ製品で、親側がパッケージ購入者のア

†2 http://docs.jboss.org/hibernate/stable/core/manual/en-US/html/mapping.html#mapping-types-anymappingを参照してください（監訳時点ではAnyというアノテーションを使用してポリモーフィック関連を実現できるようです）。Hibernate は Red Hat, Inc. の登録商標です。

†3 監訳注：6.4.1 項は監訳者による追記。

プリケーションである場合などに発生します。

ORMを使わない場合や、親子共に1つのアプリケーションで、親子テーブルがほぼ同時期に設計される場合には、この章のアンチパターンが当てはまります。親子共に設計時に自明の関連が存在するからです。

しかし、ポリモーフィック関連サポート機能を備えているORMを使っていて、関連の子側になる共通クラスを設計した時点で親テーブルが存在せず、最終的にどのような親テーブルといくつ関連を持つか分からない場合で、かつその時点ではもう共通クラスを修正することができないような状況もあります。そのような場合には、この章の「アンチパターン」の節で挙げられているリスクを念頭に置きつつ、設計の1つとしてポリモーフィック関連を検討することもあるのです。とはいえ、この章の解決策の方がより堅牢で望ましい方法なので、多くの場合で設計を再検討するほうが良いでしょう。

6.5 解決策：関連（リレーションシップ）を単純化する

データベースを再設計して、ポリモーフィック関連の弱点を避けつつ、必要なデータモデリング要件を満たさなければなりません。以降の節では、柔軟な関連付けを実現しつつも、メタデータを効果的に用いて整合性を維持する、数少ない方法を紹介します。

6.5.1 参照を逆にする

このアンチパターンの解決策の1つは、問題の本質が何かを考えるとすぐにわかります。すなわち、ポリモーフィック関連では「**本来あるべき関連が、逆さまになっている**」のです。

6.5.2 交差テーブルの作成

子テーブル側であるCommentsの外部キーでは、複数の親テーブルを参照できません。代わりに、複数の外部キーを、Commentsテーブルを参照するために使用しましょう。複数の親テーブルそれぞれに対応した**交差テーブル**を作成し、各交差テーブルではCommentsへの外部キーに加えて、各親テーブルへも同じく外部キーを定義します。この設計を、図6-4のER図に示します。

Polymorphic/soln/reverse-reference.sql

```
CREATE TABLE BugsComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);

CREATE TABLE FeaturesComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES FeatureRequests(issue_id),
```

```
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

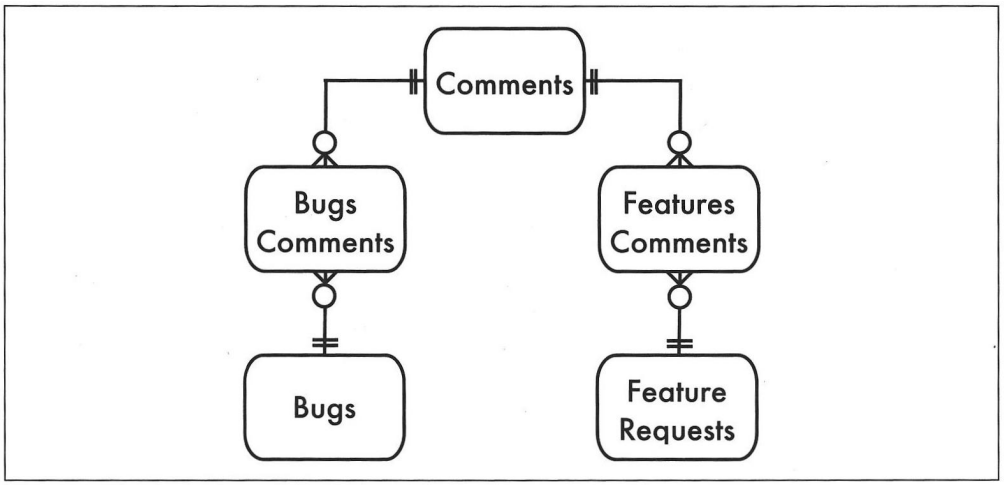


図 6-4 ポリモーフィック関連を反転させる

このような設計であれば、Comments.issue_type列が不要になります。メタデータがデータの整合性を保証してくれるので、関連付けの管理をアプリケーションコードに依存せずに済み、間違いの生じる余地が少なくなります。

6.5.3 交差点に交通信号を設置する

この解決策の潜在的な弱点は、許可したくない関連付けが許可されてしまう可能性がある点です。通常の交差テーブルでは、多対多の関連付けを作成するので、コメントを複数のバグまたは機能要求と関連付けることも可能です。しかし、今回コメントを関連付けたいのは、1つのバグまたは機能要求であるはずです。各交差テーブルのcomment_id列にUNIQUE制約を宣言することで、少なくともこのルールの一部を強制できます。

Polymorphic/soln/reverse-unique.sql

```
CREATE TABLE BugsComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (comment_id),
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

UNIQUE制約により、あるコメントが、交差テーブル内の1行のみから参照されることを保証し、複数のバグまたは複数の機能要求に関連付いてしまうのを防ぎます。しかし、このUNIQUE制約でも、あるコメントが両方の交差テーブル（BugsCommentsと FeaturesComments）から1回ずつ参照され、結果的にコメントがバグと機能要求の両方に関連付けられてしまうことを防げません。この望まない関連付けを防ぐには、アプリケーションコードの方で対処しなければなりません。

6.5.4 両方の「道」を見る

交差テーブルを用いると、特定のバグまたは機能要求を指定したコメントへのクエリを実行できます。

Polymorphic/soln/reverse-join.sql

```
SELECT *
FROM BugsComments AS b
  INNER JOIN Comments AS c USING (comment_id)
WHERE b.issue_id = 1234;
```

両方の交差テーブルと外部結合を行うことで、あるコメントに関連付いているバグまたは機能要求を取得できます。可能性のあるすべての親テーブルを明示する必要がありますが、ポリモーフィック関連アンチパターンで必要になったクエリに比べればシンプルです。また、ポリモーフィック関連とは異なり、交差テーブルを使えば参照整合性を保証できます。

Polymorphic/soln/reverse-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN (
      BugsComments INNER JOIN Bugs AS b
        USING (issue_id)
    ) USING (comment_id)
  LEFT OUTER JOIN (
      FeaturesComments INNER JOIN FeatureRequests AS f
        USING (issue_id)
    ) USING (comment_id)
WHERE c.comment_id = 9876;
```

6.5.5 「道」を合流させる

複数の親テーブルに対するクエリの結果を、親テーブル達があたかも1つのテーブルに格納しているかのように扱いたい場合があります（5章「EAV（エンティティ・アトリビュート・バリュー）」の「シングルテーブル継承」の項を参照）。これは、以下の2つの方法のいずれかで行えます。

最初の方法は、UNIONを用いたクエリです。

Polymorphic/soln/reverse-union.sql

```
SELECT b.issue_id, b.description, b.reporter, b.priority, b.status,
    b.severity, b.version_affected,
    NULL AS sponsor
  FROM Comments AS c
    INNER JOIN (
      BugsComments INNER JOIN Bugs AS b
        USING (issue_id)
    ) USING (comment_id)
  WHERE c.comment_id = 9876;
UNION
  SELECT f.issue_id, f.description, f.reporter, f.priority, f.status,
    NULL AS severity, NULL AS version_affected,
    f.sponsor
  FROM Comments AS c
    INNER JOIN (
      FeaturesComments INNER JOIN FeatureRequests AS f
        USING (issue_id)
    ) USING (comment_id)
  WHERE c.comment_id = 9876;
```

コメントが厳密に1つの親テーブルと関連付けされている場合、このクエリは1行のみを返すことを保証しなければなりません。クエリの結果をUNIONで統合できるのは、列の数とデータ型が同じである場合のみであるため、どちらかの親テーブルに固有の列にはNULLが入る可能性があります。UNIONされる両方のクエリで、同じ順番で列を列挙する必要があります。

もう1つの方法が、SQLのCOALESCE関数を用いたクエリです。この関数は、最初の非NULLの引数を返します。クエリで外部結合を用いているため、FeatureRequestsに関連があり、Bugsに一致する行のないコメントは、b.*のすべてのフィールドをNULLとして返します。同様に、コメントがFeatureRequestsではなくBugsに関連がある場合は、f.*のすべてのフィールドがNULLになります。関連が無い方の親テーブルの値にはNULLが入るので、親テーブルに固有のフィールドたちを、シンプルにCOALESCE関数を使って並べていきましょう。

Polymorphic/soln/reverse-coalesce.sql

```
SELECT c.*,
  COALESCE(b.issue_id,    f.issue_id   ) AS issue_id,
  COALESCE(b.description, f.description) AS description,
  COALESCE(b.reporter,    f.reporter   ) AS reporter,
  COALESCE(b.priority,    f.priority   ) AS priority,
  COALESCE(b.status,      f.status     ) AS status,
  b.severity,
  b.version_affected,
  f.sponsor
```

```
FROM Comments AS c
  LEFT OUTER JOIN (
      BugsComments INNER JOIN Bugs AS b
        USING (issue_id)
    ) USING (comment_id)
  LEFT OUTER JOIN (
      FeaturesComments INNER JOIN FeatureRequests AS f
        USING (issue_id)
    ) USING (comment_id)
WHERE c.comment_id = 9876;
```

どちらのクエリも、かなり複雑であるため、データベースビューの良い候補になります。ビューにすることで、アプリケーションからシンプルに使うことができます。

6.5.6 共通の親テーブルの作成

オブジェクト指向のポリモーフィズムでは、2つのサブタイプを同じように参照できます。これらのサブタイプが、暗黙的に共通の基底型を共有するためです。SQLの世界のポリモーフィック関連アンチパターンは、共通の基底型という非常に重要なエンティティが抜け落ちてしまいます。これは、すべての親テーブルが継承する基底テーブルを作成することで対応できます（5章「EAV（エンティティ・アトリビュート・バリュー）」の「クラステーブル継承」の項を参照）。基底テーブルを参照するために、子であるCommentsテーブルに外部キーを加えます。もうissue_type列は不要です。この解決策のER図を、図6-5に示します。

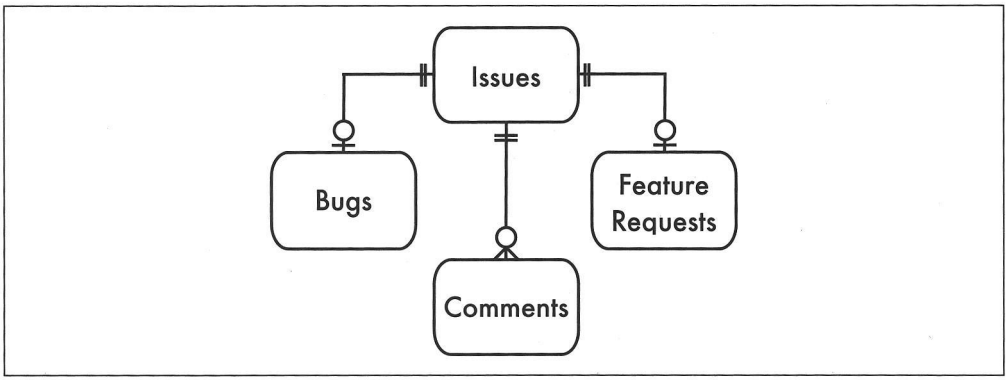


図6-5 基底テーブルへのCommentsの関連付け

Polymorphic/soln/super-table.sql

```
CREATE TABLE Issues (
  issue_id     SERIAL PRIMARY KEY
);
```

```
CREATE TABLE Bugs (
  issue_id     BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);

CREATE TABLE FeatureRequests (
  issue_id     BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);

CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  issue_id     BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment      TEXT,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id),
);
```

Bugsと FeatureRequestsの主キーは、外部キーでもあります。Bugsと FeatureRequestsの主キーは、自身で新たな値を生成せず、Issuesテーブルで生成された疑似キー（issue_id）を参照します。

比較的シンプルなクエリを用いて、特定のコメントが参照しているバグまたは機能要求を取得できます。Issuesテーブルで（issue_id以外の）属性列を定義していない限り、クエリに Issuesテーブルを含める必要はありません。そして、Bugsテーブルと、その先祖である Issuesテーブルの主キー値は同じであるため、Bugsを直接 Commentsに結合できます。値を比較可能な列を使っている限り、列の間に外部キー制約がなくとも、2つのテーブルを JOINできるのです。

Polymorphic/soln/super-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id)
WHERE c.comment_id = 9876;
```

特定のバグのコメントも、同じく簡単に取得できます。

Polymorphic/soln/super-join.sql

```
SELECT *
FROM Bugs AS b
  JOIN Comments AS c USING (issue_id)
WHERE b.issue_id = 1234;
```

Issuesのような基底テーブルを使えば、外部キーによるデータ整合性制約に依存できるというところがポイントです。

> テーブル間の関連（リレーションシップ）には、参照元テーブルと参照先テーブルが常にそれぞれ1つしかないことを忘れないようにしましょう。

7章
マルチカラムアトリビュート（複数列属性）

高貴さと愚かさはしばしばあまりにも深く結びついているので、２つを区別するのは困難である。
——トマス・ペイン（イギリスの社会思想家）

私はこれまでに、連絡先情報を格納するテーブルを数え切れないほど作ってきました。連絡先テーブルには、典型的な列があります。名前、敬称（Mr. や Miss. など）、住所、会社名などです。

電話番号は少しばかり厄介です。たいてい、誰でも複数の電話番号を持っているからです。代表的なのは、自宅、職場、ファックス、携帯電話の４つです。シンプルに考えるなら、連絡先テーブルに４つ列を定義してそれぞれ格納するでしょう。

しかし、さらに電話番号がある場合はどうでしょうか。例えば、アシスタントの電話番号や、２台目の携帯電話の番号を記載しておきたいケースもありますし、頻繁に出張するオフィスの番号を記載したい人もいるでしょう。電話番号には、様々なカテゴリーがありそうです。あまり一般的ではないケースのために列を追加することもできるでしょうが、データ入力画面に、まれにしか使われないフィールドを加えるのは、あまりスマートな方法とは言えません。そもそも、いくつ列を追加すればよいかの判断も難しいところです。

7.1 目的：複数の値を持つ属性を格納する

これは１章「ジェイウォーク（信号無視）」アンチパターンと同じテーマの問題です。すなわち、１つのテーブルに属するべきだと思える属性に複数の値がある場合、それをどう格納するかという問題です。ジェイウォーク（信号無視）アンチパターンでは、複数の値をカンマ区切りフォーマットで連結し、１つの列に格納してしまいました。その結果、値のバリデーション、個々の値の変更や読み取り、個数集計などの処理が難しくなってしまいました。

この章では、新しい例を用いてアンチパターンを説明します。バグデータベースに、バグを分類するための**タグ付け機能**を追加したいとしましょう。バグの分類基準としては、バグが影響を及ぼすソフトウェアのサブシステム（例：印刷、レポート、電子メール）を用いることが考えられますし、欠陥の種類による分類も考えられるでしょう。例えば、クラッシュしてしまうバグには crash、処理速度低下を報告するバグには performance、ユーザーインタフェースの色使いの問題などは cosmetic などのタグを付けるといった具合です。

このタグ付け機能は、1つのバグに複数のタグを付けられるようにしなければなりません。タグは必ずしも相互排他的ではないためです。1つのバグが複数のシステムに影響する場合がありますし、印刷機能のパフォーマンスに影響するようなバグもあるからです。

7.2 アンチパターン：複数の列を定義する

対象の属性には、複数の値を許可しなければなりません。しかしジェイウォーク（信号無視）アンチパターンで見たように、複数値をカンマ区切りで1列に格納すべきではありません。各列には、値を1つのみ格納すべきです。それならば、それぞれ1つのタグを格納する列を複数作成することが、自然な選択のように思えてきます。

Multi-Column/anti/create-table.sql

```
CREATE TABLE Bugs (
  bug_id      SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  tag1        VARCHAR(20),
  tag2        VARCHAR(20),
  tag3        VARCHAR(20)
);
```

あるバグにタグを付けるには、これらの3列のどれか1つに値を入れます。バグに付けるタグの数が3つ未満の場合、未使用の列はNULLのままにします（表7-1）。

Multi-Column/anti/update.sql

```
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

表7-1 未使用の列はNULLのままにする

bug_id	description	tag1	tag2	tag3
1234	保存処理でクラッシュする	crash	NULL	NULL
3456	パフォーマンスの向上	printing	performance	NULL
5678	XMLのサポート	NULL	NULL	NULL

しかし、この設計には問題があります。従来型の設計では簡単にできていたことが、複雑になってしまうのです。

7.2.1 値の検索

例えば、特定のタグが付けられたバグを検索しようとすると、3列すべてを取得しなければなりません。タグ文字列は3つの列のどれにでも格納される可能性があるためです。

Multi-Column/anti/search.sql

```
SELECT * FROM Bugs
WHERE tag1 = 'performance'
   OR tag2 = 'performance'
   OR tag3 = 'performance';
```

performance と printing の両方のタグが付いたバグを検索するとします。検索には以下のようなクエリを用います。ORは ANDより優先度が低いため、括弧の使い方には気をつけなくてはいけません。

Multi-Column/anti/search-two-tags.sql

```
SELECT * FROM Bugs
WHERE (tag1 = 'performance' OR tag2 = 'performance' OR tag3 = 'performance')
  AND (tag1 = 'printing' OR tag2 = 'printing' OR tag3 = 'printing');
```

複数の列から1つの値を検索するという単純なタスクのための構文が、長く、手間暇のかかるものになってしまいました。ただし、IN述語を従来とは少々異なる方法で用いると、この構文をコンパクトにすることができます。

Multi-Column/anti/search-two-tags.sql

```
SELECT * FROM Bugs
WHERE 'performance' IN (tag1, tag2, tag3)
  AND 'printing'    IN (tag1, tag2, tag3);
```

7.2.2 値の追加と削除

複数の列に対する値の追加や削除にも問題が生じます。単純にUPDATEを用いて、1列のみを対象とした変更を行おうとしても、それは安全とは言えません。どの列が空いているかを確認できないからです。確認のためには、対象の行をアプリケーションで取得しなければなりません。

Multi-Column/anti/add-tag-two-step.sql

```
SELECT * FROM Bugs WHERE bug_id = 3456;
```

今回は、取得した結果からtag2が NULLであることが分かったとしましょう。これでようやく UPDATE文を作成できるようになります。

Multi-Column/anti/add-tag-two-step.sql

```
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

テーブルを検索してから更新するまでの間には、他のクライアントが同じステップ（行の読み込みと更

新）を行うリスクがあります。誰が最初に更新を実行したかによって、更新の競合エラーが返されたり、どちらかの更新が上書きされたりする危険があるのです。この2ステップのクエリを避けるためには、複雑なSQLが必要です。

以下のUPDATE文では、**NULLIF関数**を用いて、特定の値に等しい場合に各列をNULLにします。NULLIF関数は、2つの引数が等しい場合にNULLを返します[†1]。

Multi-Column/anti/remove-tag.sql

```
UPDATE Bugs
SET tag1 = NULLIF(tag1, 'performance'),
    tag2 = NULLIF(tag2, 'performance'),
    tag3 = NULLIF(tag3, 'performance')
WHERE bug_id = 3456;
```

以下のUPDATE文は、現在NULLが入っている1番目の列に新しいタグperformanceを追加します。しかし、3列のどれにもNULLがない場合、行は変更されず、新しいタグ値は格納されません。それに、このUPDATE文の作成は手間がかかります。この例では、文字列performanceを6回も繰り返さなくてはなりません。

Multi-Column/anti/add-tag.sql

```
UPDATE Bugs
SET tag1 = CASE
    WHEN 'performance' IN (tag2, tag3) THEN tag1
    ELSE COALESCE(tag1, 'performance') END,
  tag2 = CASE
    WHEN 'performance' IN (tag1, tag3) THEN tag2
    ELSE COALESCE(tag2, 'performance') END,
  tag3 = CASE
    WHEN 'performance' IN (tag1, tag2) THEN tag3
    ELSE COALESCE(tag3, 'performance') END
WHERE bug_id = 3456;
```

7.2.3 一意性の保証

複数の列に同じ値を格納したくはありません。しかし「**マルチカラムアトリビュート（複数列属性）**」アンチパターンでは、それを防ぐことができません。例えば、以下のステートメントを防ぐことは難しいのです。

Multi-Column/anti/insert-duplicate.sql

```
INSERT INTO Bugs (description, tag1, tag2, tag3)
  VALUES ('印刷処理が遅い', 'printing', 'performance', 'performance');
```

†1 NULLIFはSQLの標準関数です。InformixとIngres以外の主要なデータベースで利用できます。

7.2.4 増加する値の処理

「マルチカラムアトリビュート（複数列属性）」アンチパターンには他にも弱点があります。それは、列数が3列では足りなくなるかもしれないという点です。列ごとに1つ値を格納するという設計を続けるには、バグに与えられるタグ数の最大値と同数の列を定義しなければなりません。しかし、テーブルの定義時に、タグ数の最大値を予測するのは困難です。

対処法の1つは、まず最初に適度な数の列を定義し、後で必要に応じて拡張していくというものです。多くのデータベースでは、既存テーブルの構造を変更できるので、例えばまずBugs.tag4を追加し、その後も必要に応じて列を追加していくことができます。

Multi-Column/anti/alter-table.sql

```
ALTER TABLE Bugs ADD COLUMN tag4 VARCHAR(20);
```

しかし、この変更によって以下の3つの問題が生じる可能性があります。

- すでにデータが格納されているテーブル構造を変更するには、同時に接続している他のクライアントからのアクセスを防ぐために、テーブル全体をロックする必要があるかもしれません。
- データベース製品の実装によっては、テーブル構造を変更する場合に、まず新規テーブルを定義し、既存テーブルからデータをコピーして、最後に既存テーブルを削除するという処理を行います。対象のテーブルに多くのデータがある場合、このデータ転送には長い時間がかかることがあります。
- マルチカラムアトリビュート（複数列属性）の列たちにさらに列を加えると、このテーブルを使うすべてのアプリケーションのすべてのSQL文を編集し、新しい列を扱うようにしなければなりません。これは正確さが求められ、時間がかかるタスクです。クエリの編集をミスしてしまうと、検出が難しいバグが生じる可能性があります。

Multi-Column/anti/search-four-columns.sql

```
SELECT * FROM Bugs
WHERE tag1 = 'performance'
  OR tag2 = 'performance'
  OR tag3 = 'performance'
  OR tag4 = 'performance'; -- このステートメントを追加しなければならない
```

7.3 アンチパターンの見つけ方

ユーザーインタフェースやプロジェクトのドキュメントに、複数の値を割り当てられるものの、値の個数の上限が定められている属性の説明があった場合、「マルチカラムアトリビュート（複数列属性）」アンチパターンが用いられていると考えられます。

属性には、意図的に選択肢の数を限定しているものがあります。しかし、多くの属性は、本来そのような

制限を持たないものです。制限が任意であったり、正当な理由が見当たらない場合も、このアンチパターンが用いられている可能性があります。

アンチパターンが使われている疑いは、以下のような発言からも得られます。

「サポートすべきタグの最大数はいくつ？」

タグのような複数の値を持つ属性のために、テーブルに列をいくつ定義すべきか悩んでいると考えられます。

「SQL で同時に複数の列を検索する方法は？」

複数の列の中から特定の値を検索しようと思っているならば、1つの論理的な属性値が複数の列に格納されてしまっていると考えられます。

アンチパターンに共通するパターン

ジェイウォーク（信号無視）アンチパターンとマルチカラムアトリビュート（複数列属性）アンチパターンには共通項があります。これらの2つのアンチパターンは、「複数の値を持つ可能性がある属性を格納する」という同じ問題に立ち向かっているのです。

先に見たジェイウォーク（信号無視）アンチパターンの例は、多対多関連を扱うものでした。この章のマルチカラムアトリビュート（複数列属性）アンチパターンの例は、1対多関連に関するものです。ただし注意すべき点は、どちらのアンチパターンも、多対多と1対多の両方の関連付けのために使われる可能性があることです。

7.4 アンチパターンを用いてもよい場合

場合によっては、属性値の選択肢を限定できる場合があります。そのようなときは、選択肢の順番が重大な意味を持つことがあります。例えば、複数のユーザーアカウントと関連付けられるものの、各関連の役割が異なるようなBugsエンティティを設計した場合です。1番目の列にはバグを報告したユーザー、2番目の列にはバグの修理を割り当てられたプログラマー、3番目の列には修正の確認を割り当てられた品質管理エンジニアのアカウントを格納する、という具合です。これらの列の値（アカウントID）そのものには互換性がありますが、実際にはその意味合いと使い方によって、論理的に異なる属性となります。

この例の場合、3つの属性に対応した3つの列をBugsテーブルで定義するのが不適切であるとは言えません。この章で記述した欠点はそれほど大きなものにはなりません。各IDは別の意味合いで扱われる可能性が高いためです[†2]。ときどき、3列すべてに対してクエリを実行する必要が生じること（例：対象のバグに関連する全員の名前を記載するレポートの作成）があり、処理がやや複雑になるものの、その他の多くのケースではシンプルな処理が可能です。

もう1つの方法は、Bugsテーブルから Accountsテーブルへの複数の関連を持つ**従属テーブル**の作成です。新しいテーブルには列を追加して、各アカウントがそのバグに対して持つ役割がわかるようにします。しか

†2 監訳注：ただし、このような場合には、列の順番に意味を持たせるよりも、それぞれreported_by、assigned_to、verified_byのような、関連の意味を示す名前を付けた外部キー列にする方が良いでしょう。

し、この構造は5章「EAV（エンティティ・アトリビュート・バリュー）」で説明した問題が発生する可能性があります。

7.5 解決策：従属テーブルを作成する

1章「ジェイウォーク（信号無視）」で見たように、最善の解決策は、属性を格納する列を1つ持つ**従属テーブル**を作成し、属性を格納することです。属性値を複数の列ではなく、複数の行に格納するのです。また、従属テーブルで外部キーを定義し、親であるBugsテーブルの行に値を関連付けます。

Multi-Column/soln/create-table.sql

```
CREATE TABLE Tags (
  bug_id       BIGINT UNSIGNED NOT NULL,
  tag          VARCHAR(20),
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

INSERT INTO Tags (bug_id, tag)
  VALUES (1234, 'crash'), (3456, 'printing'), (3456, 'performance');
```

個々のバグに付けられたタグはすべて従属テーブルの1つの列に格納されるので、あるタグを付けられたバグの検索は簡単になります。

Multi-Column/soln/search.sql

```
SELECT * FROM Bugs INNER JOIN Tags USING (bug_id)
WHERE tag = 'performance';
```

特定の2つのタグが付けられたバグ検索といった少し複雑な処理も、簡単に記述できます。

Multi-Column/soln/search-two-tags.sql

```
SELECT * FROM Bugs
  INNER JOIN Tags AS t1 USING (bug_id)
  INNER JOIN Tags AS t2 USING (bug_id)
WHERE t1.tag = 'printing' AND t2.tag = 'performance';
```

「アンチパターン」の節で登場した方法よりも、はるかに容易に関連付けの追加や削除を行えます。単純に、従属テーブルへ行を挿入するか、削除すればよいのです。値を追加する場所を探すために、空いている列を探す必要はもうありません。

Multi-Column/soln/insert-delete.sql

```
INSERT INTO Tags (bug_id, tag) VALUES (1234, 'save');

DELETE FROM Tags WHERE bug_id = 1234 AND tag = 'crash';
```

PRIMARY KEY制約によって、重複を禁止できます。あるタグが1つのバグに付与されるのは1回のみです。重複するタグの挿入を試みると、SQLは重複キーエラーを返します。

Bugsテーブルに tagN列を3つだけ定義していたときは、バグ1つあたりに登録できるタグ数も3つに限定されていました。いまや従属テーブルを用いることで、バグに必要な数だけのタグを付けられるようになったのです。

同じ意味を持つ値は、1つの列に格納するようにしましょう。

8章
メタデータトリブル（メタデータ大増殖）

こいつらを船から追い出してくれ。どれだけ人手がかかってもいい。とにかく外に出すんだ。
——ジェームズ・T・カーク（スタートレックシリーズ、宇宙船エンタープライズ号の船長）

私の妻は、Oracle PL/SQL と Java のプログラマーとして数年間働いていたことがあります。彼女は、仕事を楽にすることを目指したデータベース設計が、結果的に仕事を増やしてしまう例を話してくれました。

彼女の会社の営業部門は、Customersテーブルを使用していました。このテーブルには、顧客の連絡先、業種、営業収益などのデータが格納されています。

Metadata-Tribbles/intro/create-table.sql

```
CREATE TABLE Customers (
  customer_id   NUMBER(9) PRIMARY KEY,
  contact_info  VARCHAR(255),
  business_type VARCHAR(20),
  revenue       NUMBER(9,2)
);
```

営業部門では、取引が活発な顧客を知るために、年ごとに営業収益を分類する必要が生じました。採用されたのは、年ごとのデータを格納する列を新たに追加するという解決策です。列名には、対象の年を用います[†1]。

Metadata-Tribbles/intro/alter-table.sql

```
ALTER TABLE Customers ADD (revenue2002 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2003 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2004 NUMBER(9,2));
```

しかし実際には、注目している一部の顧客の分だけしかデータは入力されず、ほとんどの行ではこれらの年間営業収益の列はNULLのままとなりました。妻の会社のプログラマーたちは、大部分が未使用のこれら

†1　監訳注：このALTER文ではOracleの文法が使われています。

の列に、他の情報を格納できないかと考え始めたほどです。

毎年この列を1つ追加する必要がありました。Oracleの表領域を管理する権限はデータベース管理者が持っていたため、毎年何度も会議を開いて、データの移行、表領域の再構成、新しい列の追加について検討しなければならず、多くの時間と費用が浪費される結果となりました。

8.1 目的：スケーラビリティを高める

どのようなデータベースクエリでも、データ容量が増えるにつれてパフォーマンスは低下します。現時点では数千行ものデータを即座に返すクエリでも、データの増加とともに、いずれは満足できるパフォーマンスが得られなくなる地点に到達します。インデックスを賢く使うことで状況は改善しますが、それでもデータは増え続け、いつかはクエリの実行速度に影響を与えるのです。

この章の目的は、クエリの実行速度を劣化させずに、データが増加し続けるテーブルに対応できるよう、データベースの構造を設計することです。

8.2 アンチパターン：テーブルや列をコピーする

テレビシリーズの『スタートレック』[†2]には、**トリブル**と呼ばれる、全身を体毛に覆われた小さな動物が登場します。ペットとして宇宙船エンタープライズ号で飼い始めたときには、トリブルはとても可愛い動物でした。しかしすぐに、手に負えないほど繁殖力が強いことが明らかになります。船内では、次々に増加するトリブルの管理が深刻な問題になります。

増え続けるトリブルを、どこで飼えばいいのか、誰が世話をするのか。結局、カーク船長は大量のトリブルを船で飼うことはできないと判断し、エンタープライズ号からのトリブルの駆除を最優先するよう、クルーへ命令しました。

私たちは経験上、他の条件がすべて同じであれば、行数が少ないテーブルへのクエリ実行の方が、行数が多い場合よりも速く処理できることを知っています。しかしこれは、求めている処理が何かにかかわらず、「すべてのテーブルの行は少ない方がいい」という誤った考えを導く危険があります。その結果、以下の2つのアンチパターンに陥ってしまうのです。

- 行数の多いテーブルを、複数のテーブルに分割する（あるテーブルの属性の区別しやすいデータ値に基づいてテーブルを命名する）
- 列を複数列に分割する（別の属性の区別しやすい値に基づいて列を命名する）

しかし、この方法には代償が伴います。すべてのテーブルの行数を少なくしようとするために、列数が多すぎるテーブルを作成したり、行数が少ない多数のテーブルを作成しなければならなくなります。そしてどちらの場合でも、テーブル数や列数は次第に増加していきます。新しいデータの値によって、また新しいスキーマオブジェクトを作成しなければならないからです。

†2 スタートレックおよび関連するロゴは、CBS Studios Incの登録商標です。

メタデータへのデータの混入

テーブル名に年を追加するということは、データの値をメタデータ識別子と組み合わせるということです。

このアンチパターンは、5章「EAV（エンティティ・アトリビュート・バリュー）」アンチパターンと6章「ポリモーフィック関連」アンチパターンで見た、「データへのメタデータの混入」の逆です。EAV（エンティティ・アトリビュート・バリュー）やポリモーフィック関連では、メタデータ識別子（列名やテーブル名）を文字列データとしてテーブルに格納しました。

7章「マルチカラムアトリビュート（複数列属性）」アンチパターンと、この「メタデータトリブル（メタデータ大増殖）」アンチパターンは、データの値から列やテーブル名を作成します。つまり、「**メタデータへのデータの混入**」です。これらのアンチパターンを用いると、解決する問題以上に多くの問題を作り出すことになります。

8.2.1 テーブルの増殖

データを複数のテーブルに分割するには、どの行がどのテーブルに属するかを定義するポリシーが必要になります。例えばdate_reported列を使って、データを年ごとに分割できます。

Metadata-Tribbles/anti/create-tables.sql

```
CREATE TABLE Bugs_2008 ( . . . );
CREATE TABLE Bugs_2009 ( . . . );
CREATE TABLE Bugs_2010 ( . . . );
```

データベースへ行を挿入する際には、挿入する値に応じて、挿入先テーブルを選択しなければなりません。

Metadata-Tribbles/anti/insert.sql

```
INSERT INTO Bugs_2010 (..., date_reported, ...) VALUES (..., '2010-06-01', ...);
```

2010年が終わり、2011年の1月1日になったとしましょう。すると、Bugs_2011テーブルの作成を忘れていたために、新しいバグレポートを追加しようとするたびにアプリケーションからエラーが発生するようになります。

Metadata-Tribbles/anti/insert.sql

```
INSERT INTO Bugs_2011 (..., date_reported, ...) VALUES (..., '2011-02-20', ...);
```

つまり、新しいデータのために、新たなメタデータオブジェクトを作成しなければならない場合があるのです。しかし、これはSQLにおけるデータとメタデータの通常の関係ではありません。

8.2.2 データの整合性を管理する

ある日上司が、1年間に報告されたバグ数の合計が合わないと言ってきました。あなたは原因を調べ、2010年のバグが誤ってBugs_2009テーブルに入ってしまっていることに気づきました。以下のクエリは、常に空の結果を返すべきです。空でない結果が返る場合は、何か問題が生じていることを意味します。

Metadata-Tribbles/anti/data-integrity.sql

```
SELECT * FROM Bugs_2009
WHERE date_reported NOT BETWEEN '2009-01-01' AND '2009-12-31';
```

テーブル名に関連するデータが自動的に入るように制限する方法はありませんが、各テーブルでCHECK制約を宣言することはできます。

Metadata-Tribbles/anti/check-constraint.sql

```
CREATE TABLE Bugs_2009 (
  -- 他の列. . .
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) = 2009)
);

CREATE TABLE Bugs_2010 (
  -- 他の列. . .
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) = 2010)
);
```

新たにBugs_2011テーブルを作るときは、CHECK制約の値の修正を忘れないようにしなければなりません。修正を誤ると、受け入れるべき行を拒絶するテーブルを作成することになってしまいます。

8.2.3 データの同期

ある日、会社のカスタマーサポートのアナリストから、バグレポートの日付変更を依頼されました。データベースでは2010年1月3日に報告されたと記録されているのですが、実際にはこのバグはその1週間前の2009年12月27日にあるユーザーからFAXで報告されたものでした。あなたは、シンプルなUPDATE文で日付を変更できると考えました。

Metadata-Tribbles/anti/anomaly.sql

```
UPDATE Bugs_2010
SET date_reported = '2009-12-27'
WHERE bug_id = 1234;
```

しかしこの修正によって、対象の行はBugs_2010テーブルにおける不正な行になってしまいます。シンプルなUPDATE文がデータ不正を生じさせてしまう、ごく稀なケースといえます。修正するために、対象テー

ブルから行を削除し、別のテーブルに挿入しなければなりません。

Metadata-Tribbles/anti/synchronize.sql

```
INSERT INTO Bugs_2009 (bug_id, date_reported, ...)
  SELECT bug_id, date_reported, ...
  FROM Bugs_2010
  WHERE bug_id = 1234;

DELETE FROM Bugs_2010 WHERE bug_id = 1234;
```

8.2.4 一意性の保証

分割されたすべてのテーブルの間で、主キー値が一意であることを保証する必要があります。行をあるテーブルから別のテーブルに移動する場合、主キーの値が他のテーブルの主キーの値と重複していないことを確認しなくてはなりません。

シーケンスオブジェクトをサポートするデータベースを使用する場合、同一のシーケンスオブジェクトを用いて、分割されたすべてのテーブルのために主キーの値を生成できます。シーケンスオブジェクトが無く、テーブルごとに一意のID生成のみをサポートするデータベースを使用している場合、この処理が難しくなります。主キー値を作るためだけに、テーブルを1つ定義しなければなりません[†3]。

Metadata-Tribbles/anti/id-generator.sql

```
CREATE TABLE BugsIdGenerator (bug_id SERIAL PRIMARY KEY);

INSERT INTO BugsIdGenerator (bug_id) VALUES (DEFAULT);
ROLLBACK;

INSERT INTO Bugs_2010 (bug_id, . . .)
  VALUES (LAST_INSERT_ID(), . . .);
```

8.2.5 テーブルをまたいだクエリ実行

予想できたことですが、上司から複数のテーブルを参照するクエリを依頼されました。例えば、報告された年にかかわらず、すべての未完了のバグの合計を算出するクエリです。このような場合には、UNIONを使用してすべての分割テーブルのバグの集合を構築し、あたかも1つのテーブルであるかのようにクエリを実行できます。

Metadata-Tribbles/anti/union.sql

```
SELECT b.status, COUNT(*) AS count_per_status FROM (
  SELECT * FROM Bugs_2008
    UNION
  SELECT * FROM Bugs_2009
```

†3 監訳注：この例はMySQLを使っています。

```
    UNION
  SELECT * FROM Bugs_2010 ) AS b
GROUP BY b.status;
```

年月が経過し、Bugs_2011などのテーブルを作成していくにつれ、新たに作成したテーブルを参照するために、アプリケーションコードを修正しなければなりません。

8.2.6 メタデータの同期

今度は上司から、バグの修正にかかった時間を記録するための列を追加するように依頼されました。

Metadata-Tribbles/anti/alter-table.sql

```
ALTER TABLE Bugs_2010 ADD COLUMN hours NUMERIC(9,2);
```

テーブルが分割されている場合、新しい列の追加はALTER文による変更対象の1つのテーブルにしか適用されず、他のテーブルに新しい列は追加されません。

前節で見たような、分割されたテーブル全体に対してUNIONクエリを使う場合、新たな問題が生じます。UNIONで統合できるのは、これらのテーブルが同じ列を持つ場合のみなのです。列が異なる場合には、ワイルドカード「*」を使用せずに、すべてのテーブルに共通する列の名前を列挙しなければなりません。

8.2.7 参照整合性の管理

Commentsなどの従属テーブルが Bugsテーブルを参照する場合、従属テーブルで外部キーを定義できません。外部キーは単一のテーブルを指定しなければなりませんが、親テーブルが複数に分割されているからです。

Metadata-Tribbles/anti/foreign-key.sql

```
CREATE TABLE Comments (
  comment_id        SERIAL PRIMARY KEY,
  bug_id            BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs_????(bug_id) -- 親テーブルが分割されている
);
```

分割されたテーブルは、親テーブルとしてだけでなく、従属テーブルとしても問題が発生します。例えば、Bugs.reported_byは Accountsテーブルを参照します。ある人が報告したすべてのバグを、その報告年にかかわらず検索したい場合、以下のようなクエリが必要になってしまいます。

Metadata-Tribbles/anti/join-union.sql

```
SELECT * FROM Accounts a
INNER JOIN (
    SELECT * FROM Bugs_2008
```

```
    UNION ALL
    SELECT * FROM Bugs_2009
    UNION ALL
    SELECT * FROM Bugs_2010
) t ON a.account_id = t.reported_by;
```

8.2.8 メタデータトリブル列の特定

テーブルだけではなく、列もまたメタデータトリブルになる場合があります。この章の冒頭で見たように、私たちは増殖する性質を持つ列をテーブルに定義してしまうことがあります。

本書のバグデータベースで列がメタデータトリブルになる状況を挙げるとするなら、プロジェクトメトリクスの要約データを記録するテーブルが挙げられます。例えばメトリクスの小計を別々の列に格納するようなテーブルでは、列bugs_fixed_2011を加える必要が生じるのは時間の問題でしょう。

Metadata-Tribbles/anti/multi-column.sql

```
CREATE TABLE ProjectHistory (
  bugs_fixed_2008  INT,
  bugs_fixed_2009  INT,
  bugs_fixed_2010  INT
);
```

8.3 アンチパターンの見つけ方

以下の発言を耳にしたら、「**メタデータトリブル（メタデータ大増殖）**」アンチパターンの兆しかもしれません。

「じゃあ、～ごとにテーブル（または列）を作る必要があるんだね」

データベースについて、「～ごとに」という言葉がこのように使われているとき、ある列の値を用いてテーブルを分割している疑いがあります。

「このデータベースがサポートしているテーブル（または列）の最大数は？」

適切なデータベース設計を行っている限り、ほとんどのデータベース製品では、必要な数を十分に上回るだけのテーブルと列を使えます。上限を超えるかもしれないと心配しているときは、設計を再検討すべきです。

「今朝アプリケーションが新規データの追加に失敗した理由がわかった。新しい年のデータを格納するためのテーブルを作成し忘れていたんだ」

メタデータトリブルを用いた場合に生じる失敗の代表例です。新しいデータが新たなデータベースオブジェクトを必要とするとき、前もってこれらのデータベースオブジェクトを定義しておかなければ、予期せぬ障害が発生するリスクが高まります。

「複数テーブルを1回で検索するためのクエリの実行方法は？ 全部のテーブルの列は共通しているんだけど」

同一構造を持つ複数テーブルを検索する必要が頻繁にある場合は、初めから1つのテーブルにデータを格納しておくべきです。その際に、行を区別するための属性列を1つ追加すれば良いのです。

「テーブル名のパラメータをどうやって渡せばいい？ 年が動的に付加されるテーブル名にクエリを実行する必要があるんだ」

データが1つのテーブルに格納されていれば、このような処理はそもそも不要です。

8.4 アンチパターンを用いてもよい場合

テーブル分割が適切な場合の1つは、過去データを最新のデータから分離するようなアーカイブが目的のときです。一般的に、データが最新のデータではなくなった場合、これらの過去データに対してクエリを実行する必要性は大幅に低下します。現在のデータと過去のデータとを合わせてクエリ実行する必要が無い場合、過去データを別の場所にコピーし、最新データを保存するテーブルから削除するのは適切な処理だと言えるでしょう。互換性を持つ構造のテーブルに過去データを保管し、まれに生じる分析に用いるようにすることで、最新データへのクエリ実行のパフォーマンスは大幅に向上します。

WordPress.com でのデータベースシャーディング

MySQL Conference & Expo 2009 にて、私は WordPress.com のデータベースアーキテクト、Barry Abrahamson と昼食を共にしました。WordPress.com は、ブログ作成ソフト向けのホスティングサービスを提供する人気サイトです。

Barry はブログのホスティングを開始したとき、すべての顧客のデータを単一のデータベースで管理していたと教えてくれました。各ブログサイトのコンテンツの容量は、当時はさほど大きくなかったのです。1つのデータベースのみであれば、管理がしやすいという根拠に基づいた判断でした。

この方法は、初めはうまく機能しましたが、すぐに扱うデータが大規模なものになっていきました。現在同サイトでは、300 台のデータベースサーバで 7 百万件のブログをホスティングしています。各サーバーには、顧客のサブセットが割り当てられています。

Barry がサーバーを追加するとき、個々のユーザーのブログに属するデータを単一のデータベースから分割していく作業は極めて困難でした。ユーザー単位でデータを分割していくつかのデータベースに格納することで、各ブログの別サーバーへの移動ははるかに容易になりました。ユーザーの出入りが激しくなり、アクセスや更新が頻繁に行われるブログと、そうでないブログが出てくるにつれ、複数のサーバで負荷のバランス調整を行うことが Barry にとって重要な仕事になりました。

テラバイト級のデータを格納する 1 台のデータベースよりも、適度なサイズに分割された複数のデータベースの方が、バックアップとリストアを容易に行えます。例えば、ユーザーが電話で、不適切なデータ入力のためにデータが破損したと言ってきたとします。もしすべてのユーザーのデータが単一の巨大なデータベースバックアップに入っているならば、特定の顧客のデータのみをリストアするのは極めて困難になるでしょう。

データモデリングの観点からは、単一データベースにすべてを保存するのが正しい方法と見なされるかもしれません。しかし、適切にデータベースを分割することで、単一データベースでは管理が難しいサイズにまで大きくなったデータベースの管理タスクが容易になるのです。

8.5 解決策：パーティショニングと正規化を行う

テーブルサイズが巨大化した場合に、手作業でテーブルを分割せずに、パフォーマンスを改善するもっと良い方法があります。**水平パーティショニング**、**垂直パーティショニング**、**従属テーブルの導入**などです。

8.5.1 水平パーティショニングの使用

水平パーティショニングあるいはシャーディングと呼ばれる機能を用いることで、テーブル分割の欠点に悩まされることなく、巨大なテーブルを分割するメリットを得られます。行を分割するいくつかのルールを定めて論理テーブルを定義すれば、あとはデータベースが必要な作業を行ってくれます。テーブルは物理的には分割されていますが、あたかも1つのテーブルを扱うように、SQLステートメントを実行できます。

分割の方法は、テーブルごとに柔軟に定義できます。例えば、MySQL 5.1のパーティション機能を使うには、CREATE TABLEステートメントのオプション部にパーティション方法を定義します。

Metadata-Tribbles/soln/horiz-partition.sql

```
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY,
  -- 他の列 . . .
  date_reported DATE
) PARTITION BY HASH ( YEAR(date_reported) )
  PARTITIONS 4;
```

この例は、この章で既に見た「date_reported列の年に基づいた行の分割」の例と同様のパーティショニングが行われます。しかし、手作業でテーブルを分割する場合に比べ、date_reported列の値が更新された場合でも、行が不適切な分割テーブルに格納されない点、また個別の分割テーブルを参照する必要がなく、Bugsテーブルに対してクエリを実行できる点が優れています。

この例では、行の格納に用いる物理的なテーブルの数は、4つに固定されています。データベースが4年を越える期間のデータを扱う場合は、どれか1つのパーティションに2年以上のデータが格納されます。さらに年月が経過しても、同じような分割ルールが継続して適用されます。さらに細かい分割が必要だと思われるほどデータ量が増加するまでは、新たなパーティションを加える必要はありません。

パーティショニングは、SQL標準では定義されていません。このため、各種データベース製品では、独自の方法でパーティショニング機能を実装しています。パーティショニングの用語や構文、独自機能は、データベースによって異なります。ただし現在では、ほぼすべての主要なデータベース製品で、何らかの形でパーティショニングがサポートされています。

8.5.2 垂直パーティショニングの使用

水平パーティショニングがテーブルを行で分割するのに対し、**垂直パーティショニング**は列でテーブルを分割します。列でのテーブル分割は、列の一部のサイズが大きい場合や、めったに使用されない場合にメリットがあります。

BLOB列とTEXT列のサイズは可変であり、通常は非常に大きいことが考えられます。多くのデータベース

製品では、格納と検索の効率化のために、これらのデータ型を持つ列を、対象行の他の列とは自動的に分けて格納します。このため、テーブルのBLOBやTEXT列を参照しないクエリを実行する場合、他の列に効率的にアクセスできます。しかし、クエリで列のワイルドカード（*）を使うと、BLOBやTEXTを含む、すべての列がテーブルから取得されます。

例えば、本書のバグデータベースのProductsテーブルには、各製品のインストーラーのバイナリコピーを格納することが考えられます。これらのファイルは通常、自己解凍形式のアーカイブファイルです（Windowsでは.exe、Macでは.dmgなどの拡張子のファイルです）。ファイルサイズは大きい場合が多いですが、BLOB列では巨大なサイズのバイナリデータも格納できます。

論理的には、インストーラーファイルはProductsテーブルの属性であるべきです。しかしこのテーブルに対するほとんどのクエリでは、インストーラーを取得する必要は無いはずです。このような、まれにしか使用しない大きなデータをProductsテーブルに格納していると、ワイルドカード（*）を用いた全列の取得を頻繁に行う場合に、意図しないパフォーマンス問題が生じることがあります。

この問題への対処策は、BLOB列をProductsテーブルとは別の従属テーブルに格納することです。このテーブルの主キーをProductsテーブルに対する外部キーとして定義し、製品の行につき最大でも1行であることを保証します。

Metadata-Tribbles/soln/vert-partition.sql

```
CREATE TABLE ProductInstallers (
  product_id      BIGINT UNSIGNED PRIMARY KEY,
  installer_image BLOB,
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

この例は、説明したい点を強調するために極端なものになってはいますが、列を別のテーブルに格納するメリットを示しています。例えば、MySQLのMyISAMストレージエンジンでは、テーブルへのクエリ実行は、列が固定サイズを持つときに、最も効率的です。VARCHARは可変長データ型であるため、テーブルにこのデータ型の列が1つ存在することで、この長所が失われてしまいます。すべての可変長データ型の列を別テーブルに格納すると、主テーブルへのクエリが効率的になります（場合によっては、わずかしか効果がないこともあります）。

Metadata-Tribbles/soln/separate-fixed-length.sql

```
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY, -- 固定長データ型
  summary       CHAR(80),           -- 固定長データ型
  date_reported DATE,               -- 固定長データ型
  reported_by   BIGINT UNSIGNED,    -- 固定長データ型
  FOREIGN KEY  (reported_by) REFERENCES Accounts(account_id)
);

CREATE TABLE BugDescriptions (
```

```
  bug_id      BIGINT UNSIGNED PRIMARY KEY,
  description  VARCHAR(1000),      -- 可変長データ型
  resolution   VARCHAR(1000),      -- 可変長データ型
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

8.5.3 従属テーブルの導入

7章「マルチカラムアトリビュート（複数列属性）」の解決策と同様、メタデータトリブル列の解決策は、**従属テーブル**の導入です[†4]。

Metadata-Tribbles/soln/create-history-table.sql

```
CREATE TABLE ProjectHistory (
  project_id  BIGINT,
  year        SMALLINT,
  bugs_fixed  INT,
  PRIMARY KEY (project_id, year),
  FOREIGN KEY (project_id) REFERENCES Projects(project_id)
);
```

プロジェクトごとに年別の列を持たせて強引に1行に収めるのではなく、プロジェクトと年の組み合わせごとに1行となるようにテーブルを定義しましょう。このようにすると、翌年以降、新たな年をサポートするために列の定義を加える必要がありません。このテーブル構造なら、時の経過と共に、行を必要に応じて格納していくことができます。

データにメタデータを増殖させないように気をつけましょう。

†4 監訳注：サマリーテーブルである ProjectHistory テーブルの改善であることに注意してください。

II部
データベース物理設計のアンチパターン

9章
ラウンディングエラー（丸め誤差）

10.0を 0.1倍しても、 1.0になることは滅多にない。
——ブライアン・カーニハン（ベル研究所に在籍していた計算機科学者。
『プログラミング言語 C』 著者。プログラミング言語 awkの開発者の 1 人）

あなたは上司から、バグ修正に要した人件費のレポート機能を作成するように依頼されました。プロジェクトでのバグ修正全体に要した時間と各プログラマーの時給に基づいてコストを算出するというものです。Accountsテーブルの hourly_rate列には各プログラマーの時給を、 Bugsテーブルの hours列には各バグの修正に要した時間をそれぞれ格納します。hourly_rate列からプログラマーの時給を取得し、それに hours列のバグ修正時間を掛けた値が、バグ修正にかかったコストです。

Rounding-Errors/intro/cost-per-bug.sql

```
SELECT b.bug_id, b.hours * a.hourly_rate AS cost_per_bug
FROM Bugs AS b
  INNER JOIN Accounts AS a ON b.assigned_to = a.account_id;
```

このクエリを作成するためには、Bugsテーブルと Accountsテーブルに新しい列を追加する必要があります。追加する列では、コストを正確に記録するために小数値を扱わなければなりません。あなたは追加する新しい列を、小数値をサポートする FLOAT型で定義しました。

Rounding-Errors/intro/float-columns.sql

```
ALTER TABLE Bugs ADD COLUMN hours FLOAT;

ALTER TABLE Accounts ADD COLUMN hourly_rate FLOAT;
```

バグの作業記録とプログラマーの時給表を使ってデータを入力し、レポートをテストして機能を無事追加できたことを確認してから、あなたはその日の仕事を終えました。

翌日、上司がプロジェクトのコストレポートを持ってあなたの所にやってきました。

「計算が間違っているんだ」上司は憮然とした表情で言いました。「確認のために、自分で手で計算してみ

たんだ。数ドルの誤差ではあるが、君のレポートの計算値は間違っていた。いったいどういうことだ？」冷や汗が流れてきました。こんな単純な計算を、どうして間違えてしまったのでしょうか。

9.1 目的：整数の代わりに小数値を使用する

整数型は便利なデータ型です。しかし、このデータ型が格納できるのは、1や 327、-19などの整数のみであり、2.5などの小数値は表現できません。整数より精度の高い計算が必要な場合には、別のデータ型が必要です。例えばドルを基準にした金額計算では、19.95ドルのように、小数点２位の値が使用されます。

この章の目的は、整数以外の数値を格納し、数学的な計算を行うことです。それから、言うまでもないことではありますが、もう１つ目的があります。そう、**正しく計算を行うこと**です。

9.2 アンチパターン：FLOAT データ型を使用する

ほとんどのプログラミング言語には、floatや doubleと呼ばれる、実数を表すデータ型があります。SQLにも同じ名前で類似のデータ型があります。float型を用いたプログラミングに慣れているプログラマーの多くは、小数値のデータが必要な場合には当然のようにSQLの **FLOAT データ型**を選択します。

SQLの FLOATデータ型は、他のプログラミング言語の float型と同じように、**IEEE 754 標準**に従って実数を**２進数形式**でエンコードします。FLOAT型を効果的に使用するには、その浮動小数点数の特徴を理解しておく必要があります。

9.2.1 丸めが避けられない

多くのプログラマーは、「10進数で記述できるすべての値を、２進数として格納できるわけではない」という浮動小数点数の特性を、あまり意識していません。浮動小数点数における一部の数値は、やむを得ず近似値に丸めなくてはならないのです。

丸めの振舞いを理解するために、３分の１のような有理数を 0.333…のような循環小数で表す場合を考えてみましょう。正確な値は小数では表現できません。なぜなら桁数を無限に書く必要があるためです。桁数とは数の精度です。つまり循環小数では**無限精度**が必要になります。

この問題への妥協策は、可能な限りオリジナルに近い値、例えば 0.333などの **有限精度**の値を使うことです。このため、意図していた値と正確には一致しなくなります。

$\frac{1}{3} + \frac{1}{3} + \frac{1}{3}$ = 1.00

0.333 + 0.333 + 0.333 = 0.999

３分の１の近似値である 0.333の精度をさらに上げてから３倍したとしても、真値である 1.0にはなりません。これが、循環小数を持つ数を表すために、やむを得ず有限精度を使うという妥協策です。

$\frac{1}{3} + \frac{1}{3} + \frac{1}{3}$ = 1.000000

0.333333 + 0.333333 + 0.333333 = 0.999999

つまり、あなたが頭に思い浮かべることのできる数のなかには、有限精度では表現できないものがあるのです。「でも、どうせ実際に無限の桁数を入力することはできないのだから、入力する数字には有限精度のものしかないことになる。入力できる値は有限精度の値として正確に格納されるので、問題はないはずだ」と考える人もいるかもしれません。しかし、残念ながらそうではありません。

IEEE 754では、浮動小数点数を2進数形式で表現します。2進数形式で無限精度が必要な値は、10進数で表現される値とは異なります。10進数では有限精度で表せる値、例えば59.95を2進数で正確に表すには、無限精度が必要です。FLOATデータ型で無限精度を扱うことはできないため、2進数形式で格納できる近似値を使います。その近似値を10進数形式で表すと、59.950000762939になります。

値によっては、両方の形式で有限精度を偶然使用できるものがあります。理論上、IEEE 754形式における数値の格納の詳細を理解していれば、ある10進値がどのように2進数で表されるかを予測できます。しかし実際のところ、浮動小数点数を使用するたびに、この計算を行う人はまずいません。データベースのFLOAT列にある値が、すべて有限精度で表せるとは保証できません。このため、アプリケーションでは、対象の列にあるすべての値は丸められている可能性を想定しなければなりません。

データベースによっては、DOUBLE PRECISION（倍精度）やREAL（実数）と呼ばれるデータ型もあります。これらのデータ型や先ほどのFLOATがサポートする精度はデータベースの実装によって異なりますが、すべて2進数の有限精度で浮動小数点数を表すので、同じような丸めの振舞いを見せます。

9.2.2 SQLでのFLOATの使用

データベースによっては、この不正確さを埋め合わせて、意図した値を表示するものがあります。

Rounding-Errors/anti/select-rate.sql

```
SELECT hourly_rate FROM Accounts WHERE account_id = 123;
```

結果：59.95

しかしFLOAT列に格納された実際の値は、正確にはこの値とは異なるかもしれません。例えば、値を10億倍すると、差異があることがわかります。

Rounding-Errors/anti/magnify-rate.sql

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE account_id = 123;
```

結果：59950000762.939

このクエリで返される結果は、59950000000.000であると予想したかもしれません。いま見た結果は、値59.95がIEEE 754の2進数形式の有限精度で表せる値に丸められていたことを示します。この場合の誤差は1000万分の1以下であり、多くの計算では十分な精度だといえます。

しかし計算によっては、求めている結果を得られない場合があります。例えば、FLOAT型の列の値と等価性比較を行う場合です。

Rounding-Errors/anti/inexact.sql

```
SELECT * FROM Accounts WHERE hourly_rate = 59.95;
```

結果：一致する行はありません。

hourly_rateに実際に格納されていた値は、59.95よりもごくわずかに大きかったことを先ほど見ました。このため、account_idが123の行の hourly_rate列に再度値 59.95を割り当てても、リテラル値 59.95とは等価になりません。

この問題の一般的な回避策は、浮動小数点数同士がごく小さな範囲内にある場合、それらを「実質的に等しいもの」として扱うという方法です。一方の値からもう一方の値を引き、絶対値を返すABS関数を用いて符号を取り除きます。結果がゼロの場合、2つの値は正確に等しいことを意味します。結果が十分に小さい場合、2つの値を実質的に等しいものとして扱うのです。以下のクエリでは、一致する行を見つけることができます。

Rounding-Errors/anti/threshold.sql

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.000001;
```

しかし、差異はまだ大きいので、より高い精度で比較すると失敗してしまいます。

Rounding-Errors/anti/threshold.sql

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.0000001;
```

適切な閾値は状況に応じて変わります。10進数形式の値と、丸められた2進数形式の値の差異は値によって異なるからです。

FLOATの性質に起因する精度の問題は、集約計算を行う場合にも生じます。例えば、SUM関数を用いて列内の浮動小数点数を合計する場合、すべての値が丸められているために、ひとつひとつの誤差が累積されます。

Rounding-Errors/anti/cumulative.sql

```
SELECT SUM( b.hours * a.hourly_rate ) AS project_cost
FROM Bugs AS b
INNER JOIN Accounts AS a ON b.assigned_to = a.account_id;
```

浮動小数点数の誤差累積の影響は、和ではなく、積を計算する場合にさらに大きくなります。もともとの差異は小さく見えますが、計算によってみるみる増大していきます。例えば、値1に 1.0を掛けると、結果は常に1です。1.0を何回掛けようとも、結果は同じです。しかし、掛ける数の実際の値が 0.999の場合、結果は異なります。1に連続で1000回 0.999を掛けると、結果は約 0.3677です。さらに掛ける回数を増やすと、差異はさらに大きくなります。

掛け算を何度も連続して行う典型的な例が、金融システムにおける利子の複利計算です。浮動小数点数を用いると、初めはごく小さく見えた誤差が、計算を繰り返すうちに、まさに複利で増えていく利子のように増大していきます。金融系のアプリケーションでは、正確な値を使うことが極めて重要なのです。

IEEE 754 形式への適合

浮動小数点数の２進数標準形式が提案されたのは、1979 年に遡ります。この標準は 1985 年に規格化され、ソフトウェアやプログラミング言語、マイクロプロセッサーで広く用いられるようになりました。

この形式には、浮動小数点数のエンコードのための３つのフィールドがあります。**仮数部（fraction）**のフィールド、**指数部（exponent）**のフィールド、１ビットの**符号（sign）**フィールドです。

IEEE 754 の長所の１つは、指数を用いることで、非常に小さい、あるいは非常に大きい小数値を表すことができる点です。この形式は、実数をサポートするだけではなく、固定小数点形式において整数を使って表現される値よりも広い範囲をサポートできます。倍精度の形式は、さらに大きな範囲の値をサポートします。このため、これらの形式は科学技術計算用のアプリケーションでよく用いられています。

しかし、小数値が最も多く用いられるのは、おそらく金額の計算のためです。金額の計算では、IEEE 754 を使う必要はありません。この章で説明する 10 進数形式でも、容易かつ正確に金額を扱えるからです。

IEEE 754 形式についての良い参考資料には、Wikipedia の該当記事[†1] や、David Goldberg の記事『What Every Computer Scientist Should Know About Floating-Point Arithmetic』[Gol91]などがあります。Goldberg の記事は PDF[†2] でも参照できます。

9.3 アンチパターンの見つけ方

実質上、FLOAT、REAL、DOUBLE PRECISIONなどのデータ型を用いると、「完全に正確な値」は**期待できない**と考えるべきです。つまり、「**ラウンディングエラー（丸め誤差）**」アンチパターンに陥っている可能性があります。浮動小数点数を使うほとんどのアプリケーションは、IEEE 754 規格でサポートされる値の範囲を必要としていません。

SQL で FLOATデータ型を使うことは、自然なことだと見なされています。ほとんどのプログラミング言語に見られるデータ型と同じ名前を持っているからです。しかし、このデータ型よりも相応しい選択肢があるのです。

9.4 アンチパターンを用いてもよい場合

FLOATは、 INTEGERや NUMERICなどのデータ型がサポートするよりも広い範囲の実数を扱う場合に適しています。FLOATが最適なものの代表例は、科学技術計算を行うアプリケーションです。

Oracle では、FLOATデータ型は指定された精度を持つ数値として用いられます。Oracle の BINARY_FLOAT データ型が、IEEE 754 に準拠した概数です。

†1 http://en.wikipedia.org/wiki/IEEE_754-1985
†2 http://www.validlab.com/goldberg/paper.pdf

9.5 解決策：NUMERICデータ型を使用する

FLOATやその類似したデータ型の代わりに、SQLデータ型の NUMERICまたは DECIMALを用いて、固定精度の小数点数を表すようにしましょう[†3]。

Rounding-Errors/soln/numeric-columns.sql

```
ALTER TABLE Bugs ADD COLUMN hours NUMERIC(9,2);
ALTER TABLE Accounts ADD COLUMN hourly_rate NUMERIC(9,2);
```

これらのデータ型は、列の定義に指定した精度で数値を格納します。精度はVARCHARデータ型の長さの指定で用いる構文と同様に、データ型への引数として指定します。**精度**（precision）とは、この列の値として使用可能な10進数の桁の総数です。精度に9を指定すると、123456789のような値は格納できますが、1234567890は格納できなくなります[†4]。

このデータ型の第2引数にはスケールを指定します。**スケール**（scale）とは、小数点以下に格納できる桁数です。スケールは精度の桁数に含まれるため、精度が9で、スケールに 2を指定した場合は、1234567.89は格納できますが、12345678.91や 123456.789のような値は格納できません。

列定義で指定する精度とスケールは、対象となる列のすべての行に適用されます。つまり、例えばある行にはスケール 2で値を格納し、別の行にはスケール 4の値を格納するようなことはできません。SQLでは常に、列のデータ型はすべての行に適用されます（例えば VARCHAR(20)で定義された列では、すべての行でこの長さの範囲内の文字列を格納するのと同じことです）。

NUMERICおよび DECIMAL型の長所は、FLOATデータ型とは異なり、有理数を丸めることなく格納できる点です。値 59.95を格納するとき、その値が正確に格納されていることを信頼できるのです。格納された値とリテラル値 59.95との等価性を比較すると、比較は成功します。

Rounding-Errors/soln/exact.sql

```
SELECT hourly_rate FROM Accounts WHERE hourly_rate = 59.95;
```

結果：59.95

また、値に10億を掛けても、期待した値が得られます。

Rounding-Errors/soln/magnify-rate-exact.sql

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE hourly_rate = 59.95;
```

結果：59950000000

NUMERIC型と DECIMAL型は同一の振舞いをします。これらのデータ型の間に違いはありません。また、DECも DECIMALのシノニム（同義語）です。

---

†3 監訳注：なお、Oracleでは NUMBERデータ型が一般的に使用されます。

†4 データベース製品によっては、列のサイズは内部的に、最も近いバイト、ワード、ダブルワードに切り上げられます。このため、NUMERIC型の列の最大値は、指定した精度より多くの桁を持つかもしれません。

これらのデータ型を使用しても、3分の1のような、無限精度が必要な値を格納することはできません。しかし、少なくとも私たちは、10進数形式におけるこのような無限小数の取り扱いには慣れています[†5]。

正確な10進数値が必要な場合は、NUMERICデータ型を用いましょう。FLOATデータ型では、10進法の有理数の多くを表現できません。つまり、FLOATは概数として扱われるべきなのです。

> できる限り、FLOAT型は使わないようにしましょう。

†5 監訳注：例えば、アプリケーションとしては、適切な精度とスケールを選択することで、概算値を使って計算しても十分な結果を得ることができます。

10章
サーティワンフレーバー（31のフレーバー）

科学が有効なのは、変数が少なく、それらを列挙できるとき――
つまり、変数の組み合わせが明確であるときだ。
――ポール・ヴァレリー（フランスの作家、詩人）

連絡先情報のテーブルで、値の種類が少ない列といえば、**敬称**（salutation）列が典型的です。「Mr.」「Mrs.」「Ms.」「Dr.」「Rev.（聖職者）」などを入れておけば、ほぼすべての人の敬称に対応できるでしょう。値リストの指定は、列のデータ型または制約の宣言時に行えます。宣言によって、salutation列への無効な文字列の入力も防止できます。

31-Flavors/intro/create-table.sql

```
CREATE TABLE PersonalContacts (
  -- 他の列. . .
  salutation VARCHAR(4)
    CHECK (salutation IN ('Mr.', 'Mrs.', 'Ms.', 'Dr.', 'Rev.')),
);
```

他に入れておくべき敬称がないのであれば、もうこれで何も問題ないはずです。

しかし、残念ながら上司から思わぬニュースを聞かされました。会社がフランスに支社を開設するというのです。つまり、「M.」や「Mme.」、「Mlle.」などのフランス語の敬称が新たに必要になります。あなたには、新しい敬称をサポートするために連絡先テーブルを変更するというタスクが与えられました。慎重に作業しなければならず、無停止ではテーブル定義を修正できないかもしれません。

おまけに、上司からは、来月ブラジルにも支社を開設する予定だとも聞かされました。

10.1 目的：列を特定の値に限定する

列に格納できる値を限定された値に制限することは、とても有用です。列に無効な値が含まれていないことを保証できるので、列をシンプルに使うことができます。

例えば、本書のサンプルデータベースのBugsテーブルにある status列では、バグの状態を「NEW」、

「IN PROGRESS」、「FIXED」などの値で示しているとしましょう。バグステータスの重大度はプロジェクトにおけるバグ管理の方法によって異なりますが、ポイントは、status列にはこれらの値のうちのどれかが必ず格納されるということです。

データベースが無効なデータ入力を拒絶できるのが理想的です。

31-Flavors/obj/insert-invalid.sql

```
INSERT INTO Bugs (status) VALUES ('NEW'); -- OK

INSERT INTO Bugs (status) VALUES ('BANANA'); -- エラー！
```

バスキン・ロビンス社のサーティワンアイスクリーム

1953年、バスキン・ロビンス社は、毎月、日替わりで違う種類のアイスクリームが食べられるアイスクリームパーラーを始めました。現在も「サーティワンアイスクリーム」として有名なこのアイスクリームパーラーは、「31のフレーバー」というキャッチフレーズを何年も使い続けました。

それから60年ほど経過した現在、同社は21種類の定番フレーバー、12種類のシーズン限定フレーバー、16種類の地域限定フレーバー、そして様々な「今月のフレーバー」を提供しています。当初、日替わりで毎月31種類という不変のアイスクリームフレーバーを提供していた同社は、選択肢を拡張し、それらを組み替え可能な可変のセットにしたのです。

同じことは、あなたがデータベースの設計を担当するプロジェクトでも起こる可能性があります。むしろ、そうなるものだと想定しておいた方がよいくらいです。

10.2 アンチパターン：限定する値を列定義で指定する

多くの人が、有効なデータ値を列の定義時に指定するという方法をとります。列の定義はメタデータの一部です。つまり、テーブル構造それ自体の定義の一部です。

例えば、列にはCHECK制約を定義できます。CHECK制約は、制約条件がFALSEになるような値の挿入や更新を拒否します。

31-Flavors/anti/create-table-check.sql

```
CREATE TABLE Bugs (
  -- 他の列．．．
  status   VARCHAR(20) CHECK (status IN ('NEW', 'IN PROGRESS', 'FIXED'))
);
```

MySQLは、列の取りうる値を指定された値セットの中に制限する、ENUMと呼ばれる非標準のデータ型をサポートしています。

31-Flavors/anti/create-table-enum.sql

```
CREATE TABLE Bugs (
  -- 他の列 . . .
  status   ENUM('NEW', 'IN PROGRESS', 'FIXED')
);
```

MySQLの実装では、ENUMの取りうる値は文字列として宣言されますが、内部的に列に格納されるのは、定義されたリスト中の文字列の位置です。このためデータサイズはコンパクトになりますが、この列でクエリをソートすると、文字列のアルファベット順ではなく、値リストの定義順でソートされます。この振舞いは、開発者を驚かせます。

ドメイン（DOMAIN）や**ユーザー定義型（UDT）**を使い、列に入力する値を特定の値の中のどれかに限定する方法もあります。ひとたび定義すると、同じユーザー定義ドメインやデータ型をデータベース内の他の列にも適用できるので便利です。ただし、これらの機能をサポートするRDBMSは、まだ多くはありません。

許可する値を指定したトリガーを記述し、statusがこれらの値のどれかに一致しなければ、エラーを返すようにすることもできます。

しかし、ここまで見てきた解決策には、共通の短所があります。以降の節では、これらの問題の一部を見ていきます。

10.2.1 中身は何だろう

いまあなたは、バグ管理システムのバグレポート編集画面を開発しているとしましょう。あなたは、ユーザーがインタフェースに従って有効なステータス値の1つを選べるように、ステータス値を選択肢として表示するドロップダウンリストを採用しました。status列に現在許可されている列挙値を取得するには、どのようなクエリを実行すればよいでしょうか。

おそらくあなたは、以下のようなシンプルなクエリを用いて現在使用中のすべての値を取得することを思いついたのではないでしょうか。

31-Flavors/anti/distinct.sql

```
SELECT DISTINCT status FROM Bugs;
```

しかし、現時点のすべてのバグのステータスがNEWである場合、このクエリからはNEWしか返されません。この結果を用いて画面に表示するステータス値の選択肢を生成しようとすると、「ニワトリが先か、卵が先か」問題に遭遇してしまうのです。画面からは、バグのステータスを現在使用されている値以外に変更することができません。

有効なステータス値の完全なリストを取得するには、その列のメタデータ定義を取得する必要があります。ほとんどのSQLデータベースでは、メタデータ取得のためのシステムビューをサポートしています。しかし、システムビューを使ったクエリは複雑になる場合があります。例えば、MySQLのENUMデータ型

を使用している場合、INFORMATION_SCHEMAシステムビューを検索するには、以下のようにクエリを記述します。

31-Flavors/anti/information-schema.sql

```
SELECT column_type
FROM information_schema.columns
WHERE table_schema = 'bugtracker_schema'
  AND table_name = 'bugs'
  AND column_name = 'status';
```

従来型のINFORMATION_SCHEMAの結果セットでは、完全な列挙値を取得することができません。得られるのは、CHECK制約の定義やENUMデータ型を含む文字列です。例えば、前述のクエリでは、MySQLは文字列「ENUM('NEW', 'IN PROGRESS', 'FIXED')」（カッコ、カンマ、一重引用符を含む）が格納された、LONGTEXT型の列を返します。status値を画面上の選択部品に表示するためには、この文字列を解析し、一重引用符内の個々の値を抽出するアプリケーションコードを書かなくてはなりません。

CHECK制約、ドメイン（DOMAIN）、あるいはユーザー定義型（UDT）のメタデータを扱うクエリは、次第に複雑化します。すると、多くのプログラマーは、アプリケーションコードの側にも値のリストを定義し、手でメンテナンスするという選択をします。しかし、メンテナンスを怠り、アプリケーションデータがデータベースメタデータと同期しなくなると、あっというまに不具合がプロジェクトを襲います。

10.2.2 新しいフレーバーの追加

最も一般的な変更は、有効値の追加または削除です。しかし、ENUMの値やCHECK制約を追加または削除するための構文は無く、新たな値セットで列を再定義するしか方法はありません。以下に、MySQLのENUMの新規status値として、DUPLICATEを追加する場合の例を示します。

31-Flavors/anti/add-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'FIXED', 'DUPLICATE');
```

追加する際には、追加前の列定義の有効値が、NEW、IN PROGRESS、FIXEDであることを知っていなければなりません。つまり前述したような、現在の列挙値を取得するという問題に直面することになります。

データベース製品によっては、テーブルが空でない限り、列定義を変更できないものがあります。このため、テーブルへのアクセスをブロックしたうえで、テーブルの内容をダンプし、テーブルを再定義し、保存したデータを読み込まなくてはならない場合があります。この処理はあまりにも一般的なので、ETL（抽出=Extract、変換=Transform、ロード=Load）という名前を与えられています。その他のデータベース製品は、既に使われているテーブルを再構築するALTER TABLEコマンドをサポートしています。ただし、このコマンドを用いたとしても、列定義の変更は、複雑でコストのかかるものであることに変わりはありません。

あるべき論で言うならば、メタデータの変更、すなわち、テーブルと列の定義の変更は頻繁に行うべきではなく、行う際にはテストと品質保証も合わせて実施しなければなりません。ENUMの値の追加または削除のためにメタデータを変更しなければならない場合、適切なテストを省略したり、短期間でソフトウェアエンジニアリングのあらゆる技法を駆使する努力が必要になるでしょう。どちらの場合も、変更のリスクが高まり、プロジェクトが不安定になる可能性があります。

10.2.3 昔ながらの味は色褪せない

値の1つを廃止する場合、過去のデータに影響が出る場合があります。例えば、バグ管理システムの品質管理プロセスを変更し、従来のFIXEDを、CODE COMPLETEとVERIFIEDの2つのステージに分けるとします。

31-Flavors/anti/remove-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'CODE COMPLETE', 'VERIFIED');
```

列挙値からFIXEDを削除する場合、statusがFIXEDになっている既存のバグはどう扱えばよいのでしょうか。すべてVERIFIEDに変更すべきなのでしょうか。あるいは、廃止する値の代わりにNULLや何らかのデフォルト値を設定すべきなのでしょうか。

廃止された値を参照している既存の行を、そのままにしておかなければならない場合もあります。廃止されたステータス値をユーザーインタフェースに表示しないようにして、ユーザーによる誤選択を防ぐには、どうすればよいのでしょうか。

10.2.4 移植が困難

CHECK制約、ドメイン（DOMAIN）、ユーザー定義型（UDT）は、各種データベース製品間で仕様が統一されていません。ENUMデータ型はMySQLに固有の機能です。また、データベースによって、列定義で指定できるリストの長さの上限が異なる場合があります。同様にトリガーもデータベースによって異なります。複数のデータベースをサポートする必要がある場合、こうした製品ごとの違いは、解決策の選択を難しくするでしょう。

10.3 アンチパターンの見つけ方

ENUMやCHECK制約を使っていて問題に遭遇するのは、値が固定されていないときです。ENUMの採用を検討するときは、値の変更が予定されているか、再び変更する可能性がないか、よく考えましょう。変更が予定されていたり、変更の可能性があるならば、おそらくENUM型は使うべきではありません。

以下のような言葉も、「**サーティワンフレーバー（31のフレーバー）**」アンチパターンの発生を示唆しています。

「アプリケーションメニューの選択肢を1つ追加するには、データベースをオフラインにしなければならない。すべてがうまくいけば、30分もかからないはずだ」

列の定義時に値が指定されていると考えられます。メニュー追加のような変更のために、サービスを中断する必要は本来ないはずです。

「status列は、以下の値のうちの1つを持つことができる。このリストの修正が必要になることは、まずないはずだ」

曖昧な表現です。状況が変われば修正が必要になってしまうことが考えられます。

「アプリケーションコード側の値リストが、ビジネスルールと同期していない。これで何度目だろう」

異なる場所で情報を二重管理していると、このようなリスクが生じます。

10.4 アンチパターンを用いてもよい場合

前述したように、ENUMの採用は、値セットが変わらない限り、あまり問題にはなりません。値セットを取得するためにメタデータを検索するのは困難ではあるものの、値が変わらないのであれば、アプリケーションコードの側にも値セットを保持することができるからです。

ENUMが最適なのは、有効値の変更が不要だと断言できる場合です。例えば、相互排他的な2つの値（例：左／右、有効／無効、オン／オフ、内部／外部）を指定する場合などです。

また、ENUMのようなメカニズムを実装する代わりに、CHECK制約を用いて様々なことが実現できます。例えば、start時刻が、end時刻よりも小さな値であることをチェックできます。

10.5 解決策：限定する値をデータで指定する

列の値を制限するための、もっと良い方法があります。まず**参照テーブル**BugStatusを作成し、許可する値を1行に1つずつstatus列に格納します。次に、Bugs.statusに外部キー制約を宣言し、いま作成した新しいテーブルを参照させます。

31-Flavors/soln/create-lookup-table.sql

```
CREATE TABLE BugStatus (
  status  VARCHAR(20) PRIMARY KEY
);

INSERT INTO BugStatus (status) VALUES ('NEW'), ('IN PROGRESS'), ('FIXED');

CREATE TABLE Bugs (
  -- 他の列...
  status  VARCHAR(20),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
);
```

これで、Bugsテーブルに行の挿入や更新を行うときは、BugStatusテーブルに存在する status値を使わなくてはならないようにできました。このようなステータス値の強制は ENUMや CHECK制約でもできますが、この解決策では、より柔軟な方法をいくつも採用できます。

10.5.1 値セットの取得

ENUMデータ型ではメタデータに値セットを格納しましたが、この解決策ではデータに値セットを格納します。参照テーブルのデータは、SELECTで取得できます。このため、データセットとして値を取得し、ユーザーインタフェースに表示するのがずっと簡単になります。ユーザーに見せるリストをソートすることもできます。

31-Flavors/soln/query-canonical-values.sql

```
SELECT status FROM BugStatus ORDER BY status;
```

10.5.2 参照テーブルの値の更新

参照テーブルを使えば、普通の INSERT文で値を追加できます。テーブルへのアクセス遮断も不要です。列の再定義や、システムの予告停止、ETL処理も不要です。しかも、参照テーブルの現在の値セットをすべて把握していなくても、値の追加や削除を行えます。

31-Flavors/soln/insert-value.sql

```
INSERT INTO BugStatus (status) VALUES ('DUPLICATE');
```

外部キーに ON UPDATE CASCADEオプションを指定して宣言すれば、値の名前の変更も簡単に行えます[†1]。

31-Flavors/soln/update-value.sql

```
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS';
```

10.5.3 廃止された値のサポート

Bugsテーブルの行から参照されている場合、参照テーブルの行は削除できません。status列の外部キーが参照整合性を強制するため、値は参照テーブルに存在しなくてはならないからです。

ただし、廃止された値を区別するために、参照テーブルに新たに属性列を追加できます。これにより、Bugs.status列に履歴系データを保管できるようになります。また、廃止された値と、ユーザーインタフェースに表示される有効値を厳密に区別できるようにもなります。

31-Flavors/soln/inactive.sql

```
ALTER TABLE BugStatus ADD COLUMN active
  ENUM('INACTIVE', 'ACTIVE') NOT NULL DEFAULT 'ACTIVE';
```

†1 監訳注：ON UPDATE CASCADEに関しては4章「キーレスエントリ（外部キー嫌い）」を参照してください。

値を廃止するには、DELETEではなく UPDATEを用います。

31-Flavors/soln/update-inactive.sql

```
UPDATE BugStatus SET active = 'INACTIVE' WHERE status = 'DUPLICATE';
```

ユーザーインタフェースに表示する値を取得するときは、active列がACTIVEのステータス値のみを取得します。

31-Flavors/soln/select-active.sql

```
SELECT status FROM BugStatus WHERE active = 'ACTIVE';
```

値ごとの追加属性がサポートされていないENUM型やCHECK制約に比べると、柔軟性の高い処理が可能になります。

10.5.4 移植が容易

ENUMデータ型、CHECK制約、ドメイン（DOMAIN）やユーザー定義型（UDT）と異なり、参照テーブルを用いた解決策は、外部キー制約を用いた参照整合性の宣言という標準的なSQL機能を利用したものであり、移植も容易です。

値は行に格納するので、参照テーブルには、実質的に無制限の数の値を格納できます。

> 列に入力する値を限定するときは、値セットが固定されている場合はメタデータを、流動的な場合はデータを用いましょう。

11章
ファントムファイル（幻のファイル）

ある理論が唯一の可能性だと思えるときは、その理論か、
解決すべき問題のどちらかを真に理解していないと考えたほうがいい。
——カール・ポパー（イギリスの哲学者）

データベースサーバーを悲劇が襲いました。移動中のラックを誤って倒してしまい、中に格納していたハードディスクドライブを破損させてしまったのです。幸い怪我人はいませんでしたが、大多数のハードディスクドライブが駄目になりました。ラックを倒した場所のフリーアクセスフロアさえも壊れるほどの大きな衝撃でした。ただし、インフラ部門はこうした事態に備えていました。重要なシステムのバックアップを毎日作成していたのです。インフラ部門は素早く新しいサーバーを配置し、データベースを復元しました。

しかし、スモークテスト[†1]の実施中、早くも問題が見つかりました。アプリケーションでは、画像を多くのデータベースエンティティと結びつけています。ところが、画像が1つも表示されません。あなたはすぐにインフラ部門の担当者に電話をしました。

「データベースを、最新バックアップの時点まで復元したことは検証済みです」担当者は言いました。「画像ファイルはどこに格納していたのですか？」

このアプリケーションでは、画像をデータベースの外に格納していました。ファイルは外部のファイルシステムに格納し、データベースには画像ファイルへのファイルパスを格納し、アプリケーションはそのパスの参照先の画像ファイルを開いていました。

あなたは答えました。「画像はファイルとして格納しています。格納先は、データベースのデータファイルの格納場所と同じで、/varディレクトリ以下に格納しています」

担当者は残念そうに言いました。「特に指示をしてもらわない限り、/varファイルシステムのファイルはバックアップしていません。もちろん、データベースのデータファイルはすべてバックアップします。しかし/var配下にある他のファイルは基本的にログやキャッシュデータ、一時ファイルなどですので、デフォルトではこれらをバックアップしていないのです」

†1　監訳注：「スモークテスト（smoke test）とは、定義・計画した全テストケースのサブセット。プログラムの必須機能が正常に動作することを確認するのが目的で、コンポーネントやシステムの主要機能を網羅し、細かな点は無視する」『ソフトウェアテスト標準用語集（日本語版）Version 2.1.J01』[JST12]より。

あなたは意気消沈しました。製品カタログ向けのデータベースでは、11,000個以上の画像が使われていました。これらの大部分は、おそらくどこかには存在するでしょう。しかしそれらの場所を1つずつ突き止め、再度整形し、加えてウェブ検索用のサムネイル画像を作成するのは、数週間を要する大仕事になりそうです。

11.1 目的：画像をはじめとする大容量メディアファイルを格納する

最近では、画像をはじめとするメディアファイルが、多くのアプリケーションで使われています。これらのメディアは、データベース内に格納されたエンティティと結びつけられることがあります。例えば、コメントを投稿するユーザーが、自分の顔写真やアバターを表示できるような機能があります。本書を通してサンプルとして使っているバグ管理データベースでは、バグの発生状況を説明するために、スクリーンショットを添付する必要性が生じるでしょう。

この章の目的は、画像を格納し、それらをユーザーアカウントやバグのようなデータベースエンティティと結びつけることです。データベースでこれらのエンティティを取得した際に、関連付けられた画像を取得する機能がアプリケーションに必要です。

11.2 アンチパターン：物理ファイルの使用を必須と思い込む

概念的には、画像はテーブル上の属性です。例えば、Accountsテーブルには、portrait_image列を定義することがあるでしょう。

Phantom-Files/anti/create-accounts.sql

```
CREATE TABLE Accounts (
  account_id      SERIAL PRIMARY KEY,
  account_name    VARCHAR(20),
  portrait_image  BLOB
);
```

同様に、同じタイプの複数の画像を従属テーブルに格納できます。例えば、各バグは、バグを説明するための複数のスクリーンショットを持つかもしれません。

Phantom-Files/anti/create-screenshots.sql

```
CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          SERIAL NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

ここまでは、特に問題はありません。しかし画像のデータ型を選択するとなると、意見が分かれます。本書ですでに見てきたように、画像のバイナリデータは、BLOBデータ型に格納できます。しかし、BLOBを用いず、画像をファイルシステムにファイルとして格納し、ファイルパスをVARCHARとしてデータベースに格納する人も多くいます。

Phantom-Files/anti/create-screenshots-path.sql

```
CREATE TABLE Screenshots (
  bug_id           BIGINT UNSIGNED NOT NULL,
  image_id         BIGINT UNSIGNED NOT NULL,
  screenshot_path  VARCHAR(100),
  caption          VARCHAR(100),
  PRIMARY KEY (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

この問題は、ソフトウェア開発者の間で熱い議論の的になっています。どちらの解決策にも、長所があります。しかし、大多数のプログラマーの間では、ファイルを常にデータベースの外部に格納すべきであるという考えで意見が一致しています。つまり、私はかなりの少数派の方になってしまうかもしれません。しかし、外部にファイルを格納するという設計には、いくつもの大きなリスクがあります。以降の節で、それを説明していきます。

11.2.1 ファイルの削除時における問題

最初は、一種のガベージコレクションの問題です。画像がデータベースの外にある場合、データベースで画像へのパスを含む行を削除しても、そのパスの指定先のファイルは自動的に削除されません。

Phantom-Files/anti/delete.sql

```
DELETE FROM Screenshots WHERE bug_id = 1234 and image_id = 1;
```

参照元の行を削除すると同時に、対応する画像ファイルも自動的に削除するようにアプリケーションを設計していない限り、「孤児」となったファイルが蓄積されていきます。

11.2.2 トランザクション分離の問題

通常、データの更新や削除においては、トランザクションをCOMMITするまで変更は他のクライアントには見えません。

しかし、ファイルがデータベースの外部にある場合、状況がやや異なります。ファイルを削除すると、他のクライアントはすぐにそのファイルにアクセスできなくなります。ファイルの内容を変更すると、トランザクションがコミットされる前に、他のクライアントはその変更されたファイルを目にすることになります。

Phantom-Files/anti/transaction.php

```
<?php
$stmt = $pdo->query("DELETE FROM Screenshots
                     WHERE bug_id = 1234 AND image_id =1");

unlink('images/screenshot1234-1.jpg');

// 他のクライアントは、データベースの行を参照しているが
// 画像ファイルはもう参照できない

$pdo->commit();
```

現実的には、この種の問題が生じる頻度はそれほど高くはありません。また、この例に限れば、その影響も軽微なものです。ウェブアプリケーションで画像ファイルが欠けていることは、それほど珍しくはありません。しかし次のシナリオでは、深刻な結果を招く可能性があります。

11.2.3 ロールバック時における問題

一般的な設計では、エラーが発生した際にはトランザクションをロールバックします。ロールバックは、アプリケーションロジックが変更のキャンセルを要求した場合にも行われます。

例えば、DELETEステートメントによって行をデータベースから削除することで、対応するスクリーンショットファイルを削除する場合を考えてみましょう。この変更をロールバックする場合、データベースの行削除はロールバック対象なので元に戻りますが、ファイルはロールバックの対象ではないので、削除されたままになってしまいます。

Phantom-Files/anti/rollback.php

```
<?php
$stmt = $pdo->query("DELETE FROM Screenshots
                     WHERE bug_id = 1234 AND image_id =1");

unlink("images/screenshot1234-1.jpg");

$pdo->rollback();
```

データベースの行は元に戻りましたが、画像ファイルは元に戻せません。

11.2.4 データベースのバックアップツール使用時における問題

ほとんどのデータベース製品は、使用中のデータベースのバックアップを支援するためのクライアントツールを提供しています。例えば、MySQLではmysqldump、Oracleではrman、PostgreSQLではpg_dump、SQLiteでは.dumpなどのコマンドを使用できます。こういった専用の**バックアップツール**が重要

なのは、もし他のクライアントがデータベースに同時に変更を行っている場合、バックアップに中途半端な変更や参照整合性の矛盾が入ったり、バックアップファイルが破損して復旧に使えなくなってしまったりすることを防止できるという点です。

ただ、バックアップツールには、VARCHAR列に格納されたパス名が参照する先のファイルをバックアップ対象にする方法がわかりません。このため、データベースをバックアップする際、2段階のプロセスを行う必要が出てきます。まずデータベースのバックアップツールを実行し、次に外部画像ファイル向けにファイルシステム用のバックアップツールを実行するというものです。

バックアップに外部ファイルを含めたとしても、コピーされた画像ファイルが、データベースのバックアップに用いたトランザクションと同期していることは保証できません。アプリケーションでは、画像ファイルの追加や変更が随時行われます。データベースのバックアップ開始直後に、画像ファイルの追加や変更が行われる可能性もあるのです。

11.2.5 SQLアクセス権限使用時における問題

外部ファイルには、GRANTや REVOKEなどのSQLステートメントで割り当てるアクセス権限が適用されません。SQLにおけるアクセス権限は、テーブルや列へのアクセスを管理しますが、データベース内で文字列で指定された外部ファイルを対象にすることはできません。

11.2.6 ファイルはSQLデータ型ではない

screenshot_pathに格納されたパスは単なる文字列です。データベースは、その文字列が正当なパス名であることを検証しません。同じく、データベースは指定したパスにファイルが存在することを検証できません。ファイル名が変更されたり、ファイルの移動や削除が行われても、データベース内の文字列は自動的には更新されません。文字列をパス名として取り扱うロジックは、アプリケーションコードに依存します。

Phantom-Files/anti/file-get.php

```
<?php

define('DATA_DIRECTORY', '/var/bugtracker/data/');

$stmt = $pdo->query("SELECT image_path FROM Screenshots
                     WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();
$image_path = $row[0];

// 実際の画像の読み込み -- パスが正しいものでありますように！
$image = file_get_contents(DATA_DIRECTORY . $image_path);
```

データベースの長所の1つは、データの整合性を維持してくれることです。データを外部ファイルに格納すると、この利点が失われてしまうだけでなく、データベースで処理すべきチェックを行うためにアプリケーションコードを書かなければなりません。

11.3 アンチパターンの見つけ方

「**ファントムファイル（幻のファイル）**」アンチパターンの兆候を調べるには、少しばかりの調査が必要です。プロジェクトに、ソフトウェア管理者向けのガイダンス文書がある場合や、設計を担当したプログラマーと話をする機会がある場合（設計したのはあなたかもしれません）、以下のような質問への答えを探してみてください。

- 「データバックアップとリストアの手順は確立されているか」、「バックアップの検証方法はどうか」、「クリーンサーバーやバックアップを実施したサーバーとは別サーバーで、データのリストアをテストしたか」
- 「画像が増える一方になるのではなく、不要になったときにシステムから削除されるような仕組みになっているか」、「不要な画像ファイルを削除する手順はあるか」、「その手順は自動化されているか、それとも手動か」
- 「画像ファイルへのアクセス権を持っているアプリケーションユーザーは誰か」、「アクセス権はどのように適用されているか」、「権限を持たない画像を参照するリクエストをした場合、ユーザーには何が表示されるか」
- 「画像に対する変更をキャンセルできるか」、「キャンセルできる場合、アプリケーションは画像を前の状態に復旧できるか」

たいていの場合、「ファントムファイル（幻のファイル）」アンチパターンに陥っているプロジェクトは、これらの質問への適切な答えを持っていません。もちろん、すべてのアプリケーションで画像ファイル用の堅牢なトランザクション管理やSQLアクセス制御が必要なわけではありません。バックアップ中にデータベースをオフラインにする、十分な根拠があるかもしれません。しかし、これらの質問への答えが曖昧であったり、即答できないようであるならば、そのプロジェクトではきちんと検討されないまま、外部ファイルの使用が採用されている可能性が高いと考えてもよいでしょう。

11.4 アンチパターンを用いてもよい場合

画像のようなデータサイズの大きいオブジェクトをデータベース外部のファイルに格納することには、いくつもの正当な理由があります。

- データベースの容量を減らせます。整数や文字列型のようなシンプルなデータ型と比べ、画像はサイズが大きくなる傾向があります。
- データベースのバックアップが短時間で終了し、かつバックアップファイルの容量も抑えられます。画像ファイルは別のバックアップ手順の中でファイルシステムからコピーする必要があるものの、画像をデータベース内部に格納しないことで、巨大なサイズのデータベースをバックアップするよりも管理がしやすくなります。

- 画像ファイルがデータベース外にあれば、プレビューや編集が容易になります。例えば、すべての画像を一括で編集する場合、画像がデータベースの外部に格納されていれば、処理が極めてやりやすくなります。

これらの利点がプロジェクトにとって特に重要であり、この章で述べてきた問題点も深刻なものにはならないと判断できる場合は、画像をデータベースの外部に格納するという方法を採用してもよいでしょう。

また、データベース製品によっては、外部ファイルをある程度透過的に参照できる特殊なSQLデータ型をサポートしています。このデータ型は、OracleではBFILE、SQL Server 2008ではFILESTREAMと呼ばれています。

11.4.1 常に2つの設計を検討する

私は1992年に、あるプロジェクトで、画像をデータベースの外部に格納するアプリケーションを設計したことがあります。プロジェクトの目的は、技術者向けカンファレンスで使用する登録者用アプリケーションの開発でした。カンファレンスの参加者が会場に到着したとき、デジタルカメラで参加者の写真を撮影し、その写真を参加者の登録情報に加えて、参加者のIDカード用に印刷するというアプリケーションです。

私が担当したアプリケーションはかなり単純でした。画像を挿入または更新できるのは、1台のクライアントアプリケーションのみでした（シャッターの瞬間にまばたきしてしまったり、写真の出来が気に入らない場合、登録時に写真を撮り直して、置き換えることができます）。高度なトランザクション処理、複数クライアントからの同時アクセス対応、ロールバックなどの要件はありませんでした。SQLアクセス権限も使用していませんでした。画像のプレビューもシンプルで、データベースから画像をフェッチする必要もありませんでした。

私がこのプロジェクトに取り組んでいた時代は、アプリケーションとデータベースの処理能力は、今日の技術が提供するものよりはるかに低いものでした。そうした様々な技術的制約があったことを考えれば、画像ファイルを画像用のディレクトリ群に格納して、アプリケーションコードで画像ファイルを管理するのも妥当な方法だったと思います。

アプリケーションでの画像処理の方法を計画するときは、この章の「アンチパターン」の節で記述した問題がどう影響するのか、よく検討する必要があります。画像を外部ファイルに格納することが常に最善の方法であるというプログラマーの一般論にただ従うのではなく、十分な情報に基づき、よく検討したうえで判断をしましょう。

11.5 解決策：必要に応じてBLOB型を採用する

この章の「アンチパターン」の節で見た問題のいずれかが該当する場合は、画像を外部ファイルではなく、データベースの内部に格納することを考えるべきです。ほぼすべてのデータベース製品は、あらゆるバイナリデータを格納できるBLOBデータ型をサポートしています。

Phantom-Files/soln/create-screenshots.sql

```
CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
```

```
  image_id BIGINT   UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

この方法で画像をBLOB列に格納する場合、「アンチパターン」の節で見たすべての問題を解決できます。

- 画像データはデータベースに格納されるので、読み込みのための追加のステップは不要です。誤ったファイルパスを格納するというリスクもありません。
- 行を削除すると、自動的に画像も削除されます。
- 画像の変更は、コミットするまで、他のクライアントには見えません。
- トランザクションをロールバックすると、画像を以前の状態に復元できます。
- 行の更新時は、ロックが行われます。このため、他のクライアントは同時に同じ画像を更新できません。
- データベースのバックアップには、すべての画像が含まれます。
- SQLの権限管理によって、行と画像へのアクセスを管理できます。

BLOBの最大サイズはデータベース製品によって異なりますが、いずれの場合も、ほとんどの画像を格納するための十分な容量があります。各種のデータベースは、BLOBまたは類似のデータ型をサポートしています。例えばMySQLは、最大16メガバイトのデータを格納できるMEDIUMBLOB型があります。16メガバイトもあれば、たいていの画像ファイルは格納できるでしょう。またOracleは、LONG RAW（最大2ギガバイト）、BLOB（9.2までは最大4ギガバイト、10.1以降は最大128テラバイト）と呼ばれるデータ型をサポートしています。他のデータベース製品でも類似のデータ型を利用できます。

多くの場合、データベースに格納する前の元画像はファイルとして存在しているでしょう。データベースには、これらの画像ファイルをBLOB列に読み込むための方法が必要です。いくつかのデータベースでは、外部ファイルをロードする関数を提供しているものがあります。例えばMySQLのLOAD_FILE関数は、ファイルの読み込みと、内容のBLOB列への格納に使用できます。

Phantom-Files/soln/load-file.sql

```
UPDATE Screenshots
SET screenshot_image = LOAD_FILE('images/screenshot1234-1.jpg')
WHERE bug_id = 1234 AND image_id = 1;
```

BLOB列の内容をファイルに保存することもできます。例えば、MySQLにはSELECTステートメントのオプション句があり、列名や行の終了などの装飾を付けずに、クエリ結果をそのままファイルに格納できます。

Phantom-Files/soln/dumpfile.sql

```
SELECT screenshot_image
INTO DUMPFILE 'images/screenshot1234-1.jpg'
FROM Screenshots
WHERE bug_id = 1234 AND image_id =1;
```

BLOBから画像データをフェッチして、そのまま出力することもできます。ウェブアプリケーションでは、画像のようなバイナリのコンテンツも出力できます。ただし、適切にコンテントタイプを設定する必要があります。

Phantom-Files/soln/binary-content.php

```
<?php
header('Content-type: image/jpg');

$stmt = $pdo->query("SELECT screenshot_image FROM Screenshots
                     WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();

print $row[0];
```

データベース外部のリソースは、データベースでは管理できないことに注意しましょう。

12章
インデックスショットガン（闇雲インデックス）

この機械を使って結果を求めるときはいつでも、1つの問いが生じる。
最短で結果を得るためには、どのような計算方法を用いればよいかという問いだ。
哲学者の生涯からの言葉、1864年
――チャールズ・バベッジ（19世紀のイギリスの数学者）

ある日、主任データベース管理者（DBA）から電話がかかってきました。「もしもし、ちょっといいかな？ 君の力を貸して欲しいんだ」受話器から、データセンターの換気音と共に、きついオクラホマ訛りの大声が聞こえます。

「もちろんです」あなたは、要件は何だろうかと少々不安を覚えながらも、そう答えました。

「実は、データベースがサーバーを重くしているのが問題になっているんだ」DBAが続けました。「確認のためにデータセンターに入ったんだが、何が問題かがわかってきた。インデックスが作成されていないテーブルがあるかと思えば、すべての列にインデックスが作成されているテーブルもある。ともかく、データベースをなんとかしなければならない。サーバーをすべて君に任せてもいい。他に時間を取れる人間がいないんだ」

「申し訳ないですが、データベースにはそれほど詳しくはないんです」あなたはDBAを落ち着かせようとしながら答えました。「最適化は、最善の推測をしたうえで実行しているつもりです。しかし、私たちが実行しているのは、あなたたちデータベースのエキスパートなら知っている方法ではないかと思います。そちらでデータベースのチューニングはできないのでしょうか？」

「もちろん、あらゆる手段は講じたよ。でも、うまくいかないからこそ、まだここにいるんだ。最後の手段は、君のアプリケーションを調整することだが、君はそれを望んでいないだろう。ここで我々が推測してもしょうがない。だから、君に電話をして、アプリケーションの調整によってデータベースのパフォーマンスを改善できないかを尋ねているというわけなんだ」

話がよく飲み込めませんでした。あなたは用心しながら尋ねます。「どういうことでしょう？ 先ほど言った通り、私たちのチームには、専門的なデータベースの知識を持つものはいないのですが」

「それは問題じゃない」DBAは笑いながら言いました。「君はアプリケーションを知っているだろう？ そこがポイントだ。私たちは、アプリケーションについてはよくわからないからね。私の部下の1人に、君が

適切なツールを使えるように準備させる。そうすれば、ボトルネックを修正できるはずだ。君には、ちょっとしたメンタリングが必要なだけなんだ」

12.1 目的：パフォーマンスを最適化する

パフォーマンスの問題は、データベース開発者にとって最大のテーマだと言ってもいいでしょう。データベース関連のカンファレンスでも、パフォーマンス改善のためのツールや技法をテーマにしたセッションがいくつもあるのが普通です。皆できる限りの手は打っているものの、さらに少しでもパフォーマンスを上げたいと考えているのです。私は、信頼性、セキュリティ、正確さを高めるためのデータベースの設計方法やSQLの書き方について講演をすることがありますが、質疑応答で参加者から出る唯一の質問が、「その方法は、パフォーマンスに影響を与えますか？」というものであることも珍しくはありません。

データベースのパフォーマンスを改善する最善の方法は、**インデックス**を効果的に使用することです。インデックスはデータベースが使うデータ構造で、対象とするデータベース列の値が、データベースに実際に格納されている行の位置に関連付けられています。テーブルを最初から最後まで検索する方法に比べ、インデックスを用いることで検索を容易に行えるようになります。

しかし、インデックスをいつ、どのように使うべきかをきちんと理解しているソフトウェア開発者は多くありません。データベースのマニュアルや専門書に、どのような場合にインデックスを使うべきかについての明確な説明が記載されていることも、めったにありません。開発者は推測に基づいて、効果的にインデックスを使う方法を模索するしかないのです。

12.2 アンチパターン：闇雲にインデックスを使用する

推測に基づいてインデックスを使えば、間違った選択をしてしまうことは避けられません。インデックスを使うか否かの判断をインデックスを理解しないまま行うと、以下の3つのミスのどれかが起こります。

- インデックスをまったく定義しないか、少ししかインデックスを定義しなくなってしまう。
- インデックスを多く定義し過ぎるか、役立たないインデックスを定義してしまう。
- インデックスを活用しないクエリを実行してしまう。

12.2.1 インデックスをまったく定義しない

データベース関連の書物や記事には、インデックスの更新によってデータベースにオーバーヘッドが生じるとよく書いてあります。INSERT、UPDATE、DELETEを実行するたびに、データベースはテーブルのインデックスのデータ構造を更新し、インデックスを用いて適切な行セットを検索できるようにしなければなりません。

私たちは、オーバーヘッドとは無駄なものであるという考えに慣らされています。このため一部の開発者は、データベースのインデックス更新によってオーバーヘッドが発生することを知ると、そのオーバーヘッドを排除しようとします。つまり、インデックスそのものを使用しなければ良いと考えるのです。しかしこれでは、インデックスにはオーバーヘッドを正当化するだけのメリットがあるという事実を見逃してしまう

ことになります。

すべてのオーバーヘッドが無駄なわけではありません。例えば、直接収益を生み出すわけではない総務や法務、経理、設備管理などに会社が費用をかけるのは、それらが会社の成功にとって欠かせないものであるからです。インデックスも同じなのです。

通常のアプリケーションでは、テーブルに対するクエリ発行回数の方が、テーブルの更新回数よりも何百倍も多いものです。インデックスを用いてクエリを実行するたびに、インデックスの維持のために生じたオーバーヘッドを取り戻しているのです。

またインデックスは、目的の行を素早く見つけられるという点で、UPDATEステートメントや DELETEステートメントにも役立ちます。例えば、bug_id列に設定したインデックスは、以下のステートメントで役立ちます。

Index-Shotgun/anti/update.sql

```
UPDATE Bugs SET status = 'FIXED' WHERE bug_id = 1234;
```

インデックスを定義していない列で値を検索するステートメントでは、一致する行を探すためにテーブル全体を検索しなければなりません。

Index-Shotgun/anti/update-unindexed.sql

```
UPDATE Bugs SET status = 'OBSOLETE' WHERE date_reported < '2000-01-01';
```

インデックスは規格ではない

意外かもしれませんが、**ANSI SQL 規格**はインデックスについて何も言及していません。SQL 言語は、データ格納の実装と最適化に関しては何も規定していないのです。このため、各データベース製品は、インデックスを独自に実装しています。

ほとんどのデータベース製品には、CREATE INDEX構文または類似の構文がありますが、各ベンダーは他にも様々な独自技術をインデックスに加えています。インデックスの機能に関する標準はありませんし、インデックスのメンテナンス、自動的なクエリ最適化、クエリ計画レポート、EXPLAINなどのコマンドに関する標準も、実は無いのです。

インデックスを有効に活用するには、使用しているデータベース製品のドキュメントを良く読まなくてはなりません。ベンダーによって、インデックスの構文や機能は大きく異なることがあります。ただし、インデックスの論理的なコンセプトは共通しています。

12.2.2 インデックスを多く定義し過ぎる

インデックスのメリットを得られるのは、インデックスを使うクエリを実行するときです。使用されないインデックスを作るメリットはありません。例を示しましょう。

Index-Shotgun/anti/create-table.sql

```
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY,
  date_reported DATE NOT NULL,
  summary       VARCHAR(80) NOT NULL,
  status        VARCHAR(10) NOT NULL,
  hours         NUMERIC(9,2),
① INDEX (bug_id),
② INDEX (summary),
③ INDEX (hours),
④ INDEX (bug_id, date_reported, status)
);
```

この例には、役立たないインデックスがいくつかあります。

① bug_id

ほとんどのデータベースは、主キーのインデックスを自動的に作成するので、明示的にインデックスを定義するのは冗長です。このインデックスにはメリットがないだけではなく、追加のオーバーヘッドになる可能性があります。インデックスの自動作成は、各種データベース製品が独自のルールに従って行います。詳しくはデータベースのドキュメントで確認してください。

② summary

VARCHAR(80)などの長い文字列を格納するデータ型へのインデックスは、よりコンパクトなデータ型のインデックスと比べてサイズが大きくなります。それに、サマリー情報が格納された列に対して検索やソートを実行することはあまり考えられません。

③ hours

同じく、この列の特定の値を検索することは、ほとんどないと考えられます。

④ bug_id, date_reported, status

複合インデックスにはいくつものメリットがあります。しかし、多くの場合は冗長であったり、使用頻度が極めて低くなってしまいがちです。また、複合インデックスでは列の順序がとても重要です。検索条件、結合条件、ソート順において、列を定義した順（左から右）に使わなければなりません。

どのインデックスがクエリに役立つかがわからないために、テーブルのすべての列、すべての列の組み合わせにインデックスを作成する人もいます。テーブルすべてにインデックスを作成してもメリットが得られる保証はなく、オーバーヘッドを増やしてしまうことになります。

絶対に勝てるはずの賭け

コメディアンのビル・コスビーが、ラスベガスで過ごした休暇中の面白いエピソードを披露していました。カジノに繰り出してはみたものの、何度賭けても負け続ける彼は、ホテルに戻る前に、どうしても1回は勝たないと気が済まないと考えます。コスビーは200ドルをチップに換え、ルーレットのテーブルに向かうと、赤と黒、すべてのマスにチップを置きました。これなら、ボールがどのポケットに落ちても必ず勝てるはずです。ディーラーが、回転するホイールにボールを入れました。ボールは……、ホイールを飛び出して床に落ちました。

12.2.3 インデックスが役立たないとき

インデックスにおけるもう1つの誤りは、インデックスを使えないクエリを実行することです。高速にクエリを実行できる魔法のような列の組み合わせやインデックスのオプションを求めて、手当たり次第にインデックスを作成してしまう開発者がいます。

この過ちは、電話帳のたとえを使うとよく理解できます。例えば、名字が「チャールズ」の人を電話帳で調べるのは簡単です。電話帳では名字の順番に名前が記載されているからです。

しかし、名前（ファーストネーム）が「チャールズ」の人を調べたいときは、このメリットが得られません。どのような名字の人でも、名前が「チャールズ」である可能性があるので、電話帳全体を検索しなければならなくなるのです。

電話帳は、名字、名前の順に並んでいます。last_name列、first_name列からなる複合インデックスも、これと同じです。名前で検索したい場合、このインデックスは役に立ちません。

Index-Shotgun/anti/create-index.sql

```
CREATE INDEX TelephoneBook ON Accounts(last_name, first_name);
```

インデックスからメリットを得られないクエリの例として、以下も挙げられます。

```
SELECT * FROM Accounts ORDER BY first_name, last_name;
```

電話帳のシナリオと同じです。last_name列の後に first_name列を指定する複合インデックスは、first_nameを第1基準にするソートには役立ちません。

```
SELECT * FROM Bugs WHERE MONTH(date_reported) = 4;
```

date_reported列にインデックスを作成したとしても、インデックスの順番は、月を基準にした検索に役立ちません。このインデックスの順番は、年ごとに1月1日を開始日とした連続した日付に基づきます。どの年にも4番目の月があるため、インデックスには月の値が4の行が点在しています。

普通の列と同じように、インデックスを式や生成列に対して定義できるデータベースもあります。ただし、そのインデックスは使用する前に定義しなければならず、かつ定義時に指定した式でのみ役立ちます。

```
SELECT * FROM Bugs WHERE last_name = 'Charles' OR first_name = 'Charles';
```

特定の名前（ファーストネーム）で検索したい場合の問題に戻りましょう。定義したインデックスの順番とは関係なく、値は様々な場所に点在しています。上のクエリの結果は、以下のクエリの結果と同じです。この例のインデックスは、名字の特定には役立ちますが、名前の特定には役立ちません。

```
SELECT * FROM Bugs WHERE last_name = 'Charles'
UNION
SELECT * FROM Bugs WHERE first_name = 'Charles';
```

```
SELECT * FROM Bugs WHERE description LIKE '%crash%';
```

このLIKE述語は、文字列のどの部分とも一致する可能性があるため、ソートされたインデックスデータ構造は役立ちません。

12.3 アンチパターンの見つけ方

次のような言葉を耳にしたら、それは散弾銃のようにむやみやたらにインデックスを作成する「**インデックスショットガン（闇雲インデックス）**」アンチパターンの徴候であると言えます。

「このクエリを高速化するにはどうすればいい？」

おそらく、SQLについて最も多く投げかけられる質問です。しかしこの質問には、テーブルの説明、インデックス、データ量、パフォーマンスの測定と最適化などについての詳細がありません。こうした文脈がなければ、どのような答えも単なる当てずっぽうに過ぎません。

「フィールド全部にインデックスを定義したのに、なぜ実行速度が速くならないのだろう？」

「インデックスショットガン（闇雲インデックス）」アンチパターンの典型的な症例です。できる限り多くのインデックスを作成していますが、それは暗やみに向かって銃を撃っているようなものです。

「データベースを遅くすると書いてあるのを何かで読んだことがある。それ以来、インデックスは使わないようにしている」

多くの開発者は、パフォーマンス改善に役立つ万能薬を探しています。しかし、そのような単純な法則は存在しません。

低選択性のインデックス

選択性（selectivity）は、データベースインデックスについての統計値です。テーブルの総行数に対するインデックス中の個別の値の数の割合として表されます。

```
SELECT COUNT(DISTINCT status) / COUNT(status) AS selectivity FROM Bugs;
```

選択性が低いと、インデックスの有効性も低下します。なぜそうなるのか、比喩を使って考えてみましょう。

本には、異なるタイプのインデックス、すなわち「索引」があります。本の索引では、各エントリには、その語が現れるページ番号が記載されます。その語が頻繁に使われている場合、複数のページ番号がリストされます。探している箇所をすべて見るには、リストにあるページを 1 つずつ開いていかなくてはなりません。

あまりにも多くのページに登場する語は、索引の対象にはなりません。本のなかに何百回も登場する語を、わざわざ索引で調べる必要はないからです。

データベースのインデックスでも、ある値がテーブルの多くの行に現われる場合、インデックスを読み込むよりも、すべてのテーブルをスキャンする方が楽です。実際、このような状況でインデックスを使用しても、コストが増えてしまうことになります。

理想的には、インデックスの選択性を記録し、メリットの無いインデックスを使わないようにすることが望ましいと言えます。

選択性（selectivity）について[†1]

このコラムで述べられている選択性（selectivity）の定義と計算式は、MySQL 特有のものに思われます。MySQL 以外の DB（Oracle、SQL Server、DB2、Informix、PostgreSQL、Firebird など）では、選択性は「テーブルの行の総数に対し、抽出条件を適用した結果の行数が占める割合」であり、計算式は以下のようになります。

```
SELECT (SELECT COUNT(status) FROM Bugs WHERE status = '特定の値') / COUNT(status) AS
selectivity FROM Bugs;
```

このようなクエリからインデックス値のヒストグラムを生成し、抽出条件に対して最適化を行うためのプランニングを行います。ヒストグラムを取得していない場合は、インデックス値が等しく分布しているとみなし、以下の計算式で選択性を算出します。

```
SELECT 1 / COUNT(DISTINCT status) AS selectivity FROM Bugs;
```

ただし、この式は抽出条件が等価であるという場合に限ったもので、条件が非等価であれば計算式も変わります。このあたりはデータベースの実装ごとに異なり、PostgreSQL などでは、条件に応じて複数の選択性関数を利用します。

上記コラムでは「選択性が低いと、インデックスの有効性も低下します」と述べていますが、MySQL 以外の DB では正反対であり、選択性が高いとインデックスの有効性が低下することになります。

また、このインデックスの有効性に関しても、インデックスのデータ構造が **B-Tree**である場合を想定したものであり、**ビットマップインデックス**の場合は、その有効性は逆に働く点にも、注意が必要です。

参考 URL

- MySQL：http://dev.mysql.com/doc/refman/5.5/en/glossary.html#glos_selectivity
- Oracle：http://www.oracle.com/technetwork/jp/ondemand/db-new/sql-254794-ja.pdf
- SQL Server：http://technet.microsoft.com/ja-jp/library/ms172984%28v=sql.105%29.aspx

†1 監訳注：監訳者による追記。

- DB2：http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.explain.doc/doc/c0021374.html
- Informix：http://publib.boulder.ibm.com/infocenter/idshelp/v115/topic/com.ibm.rtree.doc/sii-dblade-82086.htm
- PostgreSQL：http://www.postgresql.jp/document/9.1/html/row-estimation-examples.html
- Firebird：http://www.firebirdfaq.org/faq167/

12.4 アンチパターンを用いてもよい場合

汎用的なデータベースを設計する必要がある場合、どのようなクエリを最適化しなければならないかを理解していなければ、作成すべき最適なインデックスがどれかはわかりません。メリットを得られるはずのインデックスを作成していなかったり、インデックスを作成したものの、後で不要であることがわかる場合もあるでしょう。しかし、できる限り、十分な情報に基づいた検討をすべきです。

12.5 解決策：「MENTOR」の原則に基づいて効果的なインデックス管理を行う

突き詰めれば、「インデックスショットガン（闇雲インデックス）」アンチパターンとは、十分な根拠なくインデックスを作成したり、作成しなかったりすることだと言えます。データベースを分析し、インデックスを作成するための、あるいは作成しないための、正当な理由を見つけましょう。

最適なインデックス作成を行うためのチェックリスト、「MENTOR」を紹介しましょう。MENTORとは、Measure（測定）、Explain（解析）、Nominate（指名）、Test（テスト）、Optimize（最適化）、Rebuild（再構築）の頭文字をとったものです。

データベースがボトルネックだとは限らない

ソフトウェア開発者の世界では、データベースは常にアプリケーションの処理速度を遅くし、パフォーマンスの問題の原因になるという根強い考えがあります。しかし、常にそうだとは限りません。

あるとき上司から、あるアプリケーションの実行速度が遅い理由を調べるように指示されたことがあります。その上司は、原因はデータベースの欠陥だと主張していました。プロファイリングツールを用いてアプリケーションコードを調べたところ、アプリケーションはその処理時間の80パーセントを、HTML出力から値を入れておくべきフォームを特定するためのパース処理に費やしていることがわかりました。パフォーマンスの問題は、データベースクエリとは何の関係もなかったのです。

パフォーマンス問題の原因についての仮説を立てる前に、ソフトウェア診断ツールを実行するようにしましょう。単なる推測では、適切な最適化は実践できません。

12.5.1 Measure（測定）

情報がなければ、情報に基づく判断はできません。ほとんどのデータベースには、SQLのクエリ実行時間を記録する方法があります。これによって、最大のコストがかかっている操作を識別できます。以下に例を示しましょう。

- Microsoft SQL ServerとOracleには、SQLのトレース機能と、トレース結果のレポートと分析のためのツールがあります。このツールはMicrosoftではSQL Server Profiler、OracleではTKPROFと呼ばれます[†2]。
- MySQLとPostgreSQLは、指定された閾値より実行時間が長くかかったクエリを記録できます。MySQLでは、この機能は**スロークエリログ**と呼ばれます。スロークエリログの設定パラメータlong_query_timeのデフォルト値は10秒です。PostgreSQLにも、類似した設定変数log_min_duration_statementがあります。またPostgreSQLには、クエリログの分析と注目すべきクエリの識別を支援するツール、pgFouineもあります[†3]。

アプリケーションで最も多くの時間を消費するクエリを特定していれば、最適化で最大のメリットを得るために、どこに注目すべきかがわかります。ボトルネックになっているのは1つのクエリのみで、他のクエリは効率的に実行されていることがわかるかもしれません。そのような場合には、その最も遅いクエリからまず最適化に着手します。

実行に最も時間がかかるクエリであっても、そのクエリの実行頻度が低い場合には、アプリケーションにとって最大のコストであるとは限りません。他の単純なクエリの実行頻度が高いために、実行時間に大きく影響している場合もあります。そのような場合には、実行頻度が高いクエリの最適化に注目すれば、労力に見合ったメリットを得やすくなるでしょう。

クエリパフォーマンスの測定中は、クエリ結果のキャッシュをすべて無効化しましょう。クエリ結果のキャッシュは、クエリの実行やインデックスの使用を迂回して結果を返すように設計されているため、正確な測定が行われないのです。

本番環境に配備したアプリケーションに対して**プロファイリング**を行うことで、より正確な情報を取得できます。実際にユーザーによってアプリケーションが使われているときに、稼働中のデータベースに対して、どのコードがどのくらい時間を消費しているかについての統計値を収集します。新たなボトルネックが生じていないことを確認するために、定期的にプロファイリングデータを監視しましょう。

ただし、プロファイリングツールを使用する際にはオーバーヘッドが生じるため、測定終了後にプロファイラを無効化するか、報告レートを低くしましょう。

12.5.2 Explain（解析）

最もコストがかかるクエリを特定した後は、クエリの処理が遅くなっている原因を解析します。データ

†2 監訳注：Oracle 11.1以降にはリアルタイムSQL監視機能もあります（要Enterprise Edition）。http://docs.oracle.com/cd/E16338_01/server.112/b56312/instance_tune.htm#CACGEEIF

†3 http://pgfouine.projects.postgresql.org/

ベースは、**クエリ実行計画**（Query Execution Plan：QEP）と呼ばれるクエリ最適化機能によって、クエリ実行にどのインデックスを使うかを判断しています。このQEPの分析結果のレポートを取得します（表12-1）。

表12-1 各データベース製品におけるQEP取得方法

データベース	QEPレポート機能
IBM DB2	EXPLAIN、db2explnコマンドまたはVisual Explain
Microsoft SQL Server	SET SHOWPLAN_XMLまたはDisplay Execution Plan
MySQL	EXPLAIN
Oracle	EXPLAIN PLAN
PostgreSQL	EXPLAIN
SQLite	EXPLAIN

QEPレポートに含まれる情報やレポートの形式についての規格はありません。一般的なQEPは、クエリに関連するテーブル、オプティマイザーによるインデックスの選択理由、テーブルへのアクセスの順番などを示します。レポートには、クエリの各段階で生成された行数などの統計データが含まれることもあります。

サンプルのSQLクエリから、QEPレポートを取得してみましょう。

index-Shotgun/soln/explain.sql

```
EXPLAIN SELECT Bugs.*
FROM Bugs
INNER JOIN (BugsProducts INNER JOIN Products USING (product_id))
  USING (bug_id)
WHERE summary LIKE '%crash%'
  AND product_name = 'Open RoundFile'
ORDER BY date_reported DESC;
```

表12-2のMySQL QEPレポートの key列は、このクエリが BugsProductsの主キーインデックスのみを使用していることを示しています。また Extra列は、クエリが一時テーブルで結果をソートしており、インデックスのメリットを得られていないことを示しています。

表12-2 MySQLクエリ実行計画

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	Bugs	ALL	PRIMARY, bug_id	NULL	NULL	NULL	4650	Using where; Using temporary; Using filesort
1	SIMPLE	BugsProducts	ref	PRIMARY, product_id	PRIMARY	8	Bugs.bug_id	1	Using index
1	SIMPLE	Products	ALL	PRIMARY, product_id	NULL	NULL	NULL	3	Using where; Using join

LIKE述語は、Bugsテーブル全体のスキャンを強制します。また、Productsテーブルの product_name列にはインデックスがありません。このクエリを改善するには、product_name列に新しいインデックスを作成し、かつLIKE述語の代わりに全文検索機能[†4]を導入しましょう。

QEPレポートの情報は、ベンダーによって異なります。上記の例にあるレポートの読み方を知るには、MySQLのマニュアルページ「Optimizing Queries with EXPLAIN」を参照してください[†5]。

12.5.3 Nominate（指名）

クエリのQEPを読んで、クエリがインデックスを使わないでテーブルにアクセスしている箇所を探しましょう。

一部のデータベースには、クエリのトレース統計値を収集し、多くの変更を提案してくれるツールがあります。ツールの提案には、クエリに効果がある新規インデックスの作成も含まれます。例えば以下のようなツールがあります。

- IBM DB2 Design Advisor
- Microsoft SQL Server Database Engine Tuning Advisor
- MySQL Enterprise Query Analyzer
- Oracle SQL Tuning Advisor

自動的な提案機能がなくても、インデックスがどのような場合にクエリに役立つかを見分ける方法を学ぶことができます。QEPレポートを解釈するために、データベースのドキュメントをよく読みましょう。

カバーリングインデックス

クエリに必要な列がすべてインデックスに含まれている場合、テーブルからデータ行を読み込む必要はありません。

電話帳の索引が、名前とページ番号だけで構成されていると仮定してみましょう。名前を調べた後、実際の電話番号を調べるには、名前が参照するページを開かなければなりません。このような回りくどい方法ではなく、ワンステップで情報が調べられるようにすべきです。名前が順番通りに記載され、そこに求めている属性（電話番号や住所など）が記載されているからこそ、電話帳は便利なのです。

これが**カバーリングインデックス**の仕組みです。本来のインデックスには不要な追加列を含むように、インデックスを定義できます。

```
CREATE INDEX BugCovering ON Bugs
  (status, bug_id, date_reported, reported_by, summary);
```

クエリがインデックスのデータ構造に含まれている列のみを参照する場合、データベースはインデックス

†4 全文検索に関しては、16章「ブアマンズ・サーチエンジン（貧者のサーチエンジン）」を参照してください。

†5 http://dev.mysql.com/doc/refman/5.5/en/using-explain.html

のみを読み込んでクエリ結果を生成します。

```
SELECT status, bug_id, date_reported, summary
FROM Bugs WHERE status = 'OPEN';
```

つまりデータベースは、このテーブルの対応する行を読み込む必要がありません。カバーリングインデックスは、すべてのクエリに対して使うことはできませんが、使用できる場合はパフォーマンスを飛躍的に向上させます。

12.5.4 Test（テスト）

テストは重要なステップです。インデックスの作成後、再びクエリのプロファイリングを行うのです。大事なのは、変更が効果をもたらしたことを確認してから作業を終了することです。

テストは、上司に適切な報告をしたり、最適化が正当なものであったことを示すためにも役立ちます。あなたの1週間の仕事ぶりを、「パフォーマンスの問題を改善するためにあらゆる手段を講じましたが、その結果どのような効果があったのかについては、しばらく様子を見なければなりません」と報告するよりも、「使用頻度の高いテーブルに新たなインデックスを1つ作成すべきだと判断し、重要なクエリ複数でパフォーマンスを38パーセント改善しました」と報告する方が良いはずです。

12.5.5 Optimize（最適化）

インデックスはコンパクトで、使用頻度の高いデータ構造であるため、**キャッシュメモリ**に格納されやすくなります。メモリ上のインデックスにアクセスすることによって、ディスクI/Oを伴う読み込みよりもはるかにパフォーマンスを改善できます。

データベースサーバーでは、キャッシュに割り当てるシステムメモリの量を設定できます。多くのデータベースでは、データベースが様々なシステム上でうまく機能するように、キャッシュバッファサイズを極めて低く設定しています。このため多くの場合、キャッシュサイズを増やしたくなります。

キャッシュに割り当てるメモリの量は、どれくらいにすべきなのでしょうか。この問いに対する、絶対的な答えはありません。データベースのサイズと、利用できるシステムメモリの量によって異なるからです。

使用頻度の高いデータやインデックスのキャッシュへの格納をデータベースに依存するのではなく、インデックスをあらかじめキャッシュメモリにロードしておくことでメリットが得られる場合もあります。例えば、MySQLでは、LOAD INDEX INTO CACHEステートメントを使用します。

12.5.6 Rebuild（再構築）

インデックスは、バランスが取れているときに、最も効果的です。長期にわたって行の更新や削除を行うことで、インデックスは次第に不均衡になっていきます。これは、ファイルシステムが次第に断片化していくのと似ています。実際には、最適化されたインデックスと、不均衡なインデックスの間には、大きな違いがない場合もあります。しかし、できる限りインデックスの効率を高めたいのであれば、定期的にメンテナンスを実施する価値はあります。

インデックスに関するほとんどの機能と同様に、各種データベース製品では、インデックスのメンテナンスについて、ベンダー固有の用語、構文、機能を使用しています（表 12-3）。

表 12-3 各データベース製品におけるインデックスのメンテナンスコマンド

データベース	インデックスのメンテナンスコマンド
IBM DB2	`REBUILD INDEX`
Microsoft SQL Server	`ALTER INDEX ... REORGANIZE`、`ALTER INDEX ... REBUILD`、または `DBCC DBREINDEX`
MySQL	`ANALYZE TABLE`または `OPTIMIZE TABLE`
Oracle	`ALTER INDEX ... REBUILD`
PostgreSQL	`VACUUM`または `ANALYZE`
SQLite	`VACUUM`

インデックスを再構築する頻度はどれくらいが適切なのでしょうか。「週に 1 回」のような一般的な答えを聞いたことがあるかもしれません。しかし実際には、すべてのアプリケーションに当てはまる答えはありません。それは、不均衡が生じるかもしれないテーブルに対する変更の頻度によって異なります。また、テーブルのサイズや、インデックスの最適化によってそのテーブルが得られるメリットの重要度によっても異なります。例えば、使用頻度の低い、巨大なサイズのテーブルに対して数時間をかけて最適化を実行しても、パフォーマンスの改善は 1 パーセントのみであるとします。この場合に、最適化をすべきかどうかの判断は、あなたに委ねられます。あなたこそが、誰よりもデータと運用要件について知っている人間だからです。

インデックスの効果を最大限に高めるために必要な情報はベンダーによって異なるため、使用しているデータベース製品の詳しい知識が必要です。データベースマニュアル、書籍や雑誌、ブログやメーリングリストなどからも積極的に情報収集しましょう。最も重要なルールは、**推測のみに基づいて、闇雲にインデックスをつけてはならない**ということです。

> データとクエリについての理解を深め、MENTOR の原則に基づいてインデックスを管理しましょう。

Ⅲ部
クエリのアンチパターン

13章
フィア・オブ・ジ・アンノウン（恐怖のunknown）

世の中には「わかっているとわかっている」ことがある。つまり、私たちはそれを知っていることを知っている。同じく、「わからないとわかっている」こともある。知らないということを知っているということだ。
だが、「わからないということをわかっていない」ものもある。
私たちはそれを知らないし、それを知っていないということも知らない。
——ドナルド・ラムズフェルド（アメリカの政治家、元国防長官）

本書のバグデータベースのAccountsテーブルには、first_name列とlast_name列があります。以下のように文字列連結演算子を用いると、ユーザーのフルネームを1列に整形できます[†1]。

Fear-Unknown/intro/full-name.sql

```
SELECT first_name || ' ' || last_name AS full_name FROM Accounts;
```

いま、上司からデータベースの修正を依頼されたとしましょう。上司の要求は、ユーザーのミドルネームのイニシャルをテーブルに追加することでした（同姓同名のユーザーがいることが問題なのかもしれません。ミドルネームのイニシャルを用いることは、こうした混乱を避ける良い方法です）。これはとても簡単な修正でした。次にあなたは、ミドルネームのイニシャルを持つ少数のユーザーのために、手作業でイニシャルを追加しました。

Fear-Unknown/intro/middle-name.sql

```
ALTER TABLE Accounts ADD COLUMN middle_initial CHAR(2);

UPDATE Accounts SET middle_initial = 'J.' WHERE account_id = 123;
UPDATE Accounts SET middle_initial = 'C.' WHERE account_id = 321;

SELECT first_name || ' ' || middle_initial || ' ' || last_name AS full_name
FROM Accounts;
```

†1　監訳注：MySQLで文字列連結を行うには、文字列連結演算子||の代わりにCONCAT関数を使うか、または「set @@session.sql_mode='PIPES_AS_CONCAT';」を実行する必要があります。

突然、アプリケーションはフルネームをまったく表示しなくなりました。実際には、よく注意してみると、すべての行ではありませんでした。ミドルネームのイニシャルを登録した少数のユーザーのフルネームだけは、正常に表示されています。それ以外のユーザーの名前が空っぽなのです。

ミドルネームを持たないユーザーのデータはどうなってしまったのでしょうか。上司が事象に気付き、データが失われたとパニックを起こし始める前に、あなたは問題を修正できるでしょうか。

13.1 目的：欠けている値を区別する

データベースのデータに値が無いような状況は、必然的に生じます。列の一部にまだわからない値がある状態で行を挿入しなければならない場合もありますし、正当な状況において意味のある値を持たない列もあります。SQLは、そのような**無**（null）や**不明**（unknown）、**適用不能**（inapplicable）に相当する値を、**予約語** NULLで表現します。

SQLのテーブルとクエリで、NULLを効果的に使用する方法はたくさんあります。

- 行の作成時点では不明な値に、NULLを使用できます。例えば、在職中の従業員の退職日などです。
- ある行において**適用可能**（applicable）な値が他にない場合に、その列にNULLを使用できます。例えば、電気自動車の燃料効率などです。
- 無効な値が入力された場合に、関数はNULLを返せます。例：DAY('2009-12-32')[†2]
- 外部結合（OUTER JOIN）は、一致しないテーブルの列のためにNULLをプレースホルダーとして使用します。

この章の目的は、NULLを含む列に対してクエリを書くことです。

13.2 アンチパターン：NULLを一般値として使う、または一般値をNULLとして使う

SQLでのNULLの振舞いに困惑するソフトウェア開発者は少なくありません。多くのプログラミング言語と異なり、SQLはNULLをゼロ、FALSE、空文字列とは異なる特殊な値として扱います。これはSQL標準および主要なデータベース製品のすべてに当てはまります。

13.2.1 式でNULLを扱う

開発者を困惑させる例として、NULLを含む列や式で計算する場合が挙げられます。例えば以下の場合、多くのプログラマーは、hours列に修正時間の見積りが入っていないバグに対しては結果が10になると予想するでしょう。しかし、クエリはNULLを返します。

Fear-Unknown/anti/expression.sql

```
SELECT hours + 10 FROM Bugs;
```

†2 監訳注：DAYはMySQLの関数です。ちなみにPostgreSQLで同様の処理 SELECT EXTRACT(DAY FROM DATE '2012-10-32');を行うと、NULLではなくエラーを返します。

NULLはゼロと同じではありません。不明（unknown）よりも10大きな値を持つ数は、引き続き不明（unknown）です。

NULLは長さゼロの文字列でもありません。SQL標準では、文字列とNULLを連結するとNULLが返されます（ただし、OracleとSybaseでは振舞いが異なります[†3]）。

NULLはFALSEと同じものでもありません。AND、OR、NOTを用いた論理式でも、多くの開発者を困惑させる結果が返されます。

13.2.2 NULLを許容する列の検索

以下のクエリは、assigned_to列に値123を持つ行のみを返します。assigned_to列に他の値またはNULLを持つ行は返されません。

Fear-Unknown/anti/search.sql

```
SELECT * FROM Bugs WHERE assigned_to = 123;
```

ということは、以下のクエリは先程のクエリの補集合、つまり上のクエリで返されなかった行がすべて返されるのではないかと思えます。

Fear-Unknown/anti/search-not.sql

```
SELECT * FROM Bugs WHERE NOT (assigned_to = 123);
```

しかし、このクエリもassigned_to列にNULLが割り当てられた行を返しません。NULLを用いた比較は、TRUEやFALSEではなく、すべて不明（unknown）を返します。NULLの否定ですらもNULLのままです。

NULLまたは非NULLを検索しようとして、以下のような間違いをしてしまうことは珍しくありません。

Fear-Unknown/anti/equals-null.sql

```
SELECT * FROM Bugs WHERE assigned_to = NULL;
SELECT * FROM Bugs WHERE assigned_to <> NULL;
```

WHERE句の条件は、式がTRUEであるときにのみ満たされます。しかしNULLとの比較は決してTRUEにはならず、結果は不明（unknown）です。等式であれ不等式であれ、比較結果はTRUEではなく、不明（unknown）です。このため、上の2つのクエリのどちらも、assigned_to列がNULLの行を返さないのです。

13.2.3 プリペアドステートメントでNULLを扱う

また、**プリペアドステートメント**でパラメータ化したSQLで、NULLを一般値のように扱うことも困難で

†3 監訳注：Oracleは文字列連結演算子||やCONCAT関数を使って文字列とNULLの連結を行うと、文字列と空の文字列を連結した場合と同等の結果になります。

す[†4]。

Fear-Unknown/anti/parameter.sql

```
SELECT * FROM Bugs WHERE assigned_to = ?;
```

このクエリは、パラメータに一般的な整数値を渡すときには、予期する結果を返します。しかしNULLリテラルは、パラメータには使えません。

13.2.4 NULLの使用を避ける

NULLの扱いでクエリが複雑化することを嫌って、NULLを一切使わないようにデータベースを設計するソフトウェア開発者がたくさんいます。NULLの代わりに、**不明**（unknown）または**適用不能**（inapplicable）を意味する値を新たに定義するのです。

「NULLなんて大嫌い！」

ソフトウェア開発者のジャックは、クライアントからNULLを使わないデータベースの開発を依頼されました。とにかく「NULLが嫌い」と主張するクライアントは、NULLを用いることでアプリケーションコードにエラーが生じやすくなると言います。ジャックは私に、NULLを使わずに欠けている値を表すにはどうすればよいかと尋ねました。

私はジャックに、欠けている値を表すことこそがNULLの目的であると言いました。他のどのような値を選択したとしても、その値を特別なものとして扱うためにアプリケーションコードを修正しなければなりません。

このクライアントのNULLに対する態度は、正しいとは言えません。例えば、ゼロ除算エラーを防止するためのコードを書くのが嫌だからといって、ゼロを扱うことをすべて禁止するのが得策ではないのと同じことです。

以下の例では、本来はNULLを許容する列である assigned_toと hoursに NOT NULL制約を宣言しています。何が問題かを考えてみましょう。

Fear-Unknown/anti/special-create-table.sql

```
CREATE TABLE Bugs (
  bug_id            SERIAL PRIMARY KEY,
  -- 他の列 . . .
  assigned_to       BIGINT UNSIGNED NOT NULL,
  hours             NUMERIC(9,2) NOT NULL,
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

†4 プリペアドステートメントに関しては、詳しくは20章「SQLインジェクション」を参照してください。

例えば、不明（unknown）な値を表すために、-1を使用する場合を考えます。

Fear-Unknown/anti/special-insert.sql

```
INSERT INTO Bugs (assigned_to, hours) VALUES (-1, -1);
```

hours列のデータ型は NUMERICです。このため「未指定」を意味するために用いることができるのも数値のみです。この値が本来の数値として意味を持つものであってはならないので、-1という負の数が選択されました。しかし値 -1は、 SUMや AVGのような計算を混乱させます。 WHERE句や特殊な CASE式を使用して、この値を持つ行を除外しなければなりません。なんとも皮肉なことに、NULLを禁止したのは、こうした複雑な処理を避けたいと考えていたからなのです。

Fear-Unknown/anti/special-select.sql

```
SELECT AVG( hours ) AS average_hours_per_bug FROM Bugs
WHERE hours <> -1;
```

また、他の列では値 -1が意味を持つかもしれないため、各列にはケースバイケースで異なる値を選択しなければなりません。さらに各列で使用する特殊値を記憶したり文書化しておく必要もあります。

今度は assigned_to列を見てみましょう。この列は、 Accountsテーブルの外部キーです。バグが報告されているものの、修正担当者がまだ割り当てられていないときには、どのような非 NULLの値を使えるでしょうか。この非 NULLな値は必ず Accountsテーブルの行を参照しなければなりません。このため「誰でもない」や「未割当て」を意味するプレースホルダー行を Accountsテーブルに作成する必要があります。実際のユーザーアカウントへの参照がないことを表すために参照用のアカウントを作るのも、なんとも皮肉なことです。

列に NOT NULL制約を宣言するのは、値のない列がその行にとって意味をなさない場合に限るべきです。例えば、Bugs.reported_by列には必ず値が必要です。すべてのバグは、誰かによって報告されたものだからです。しかしバグにまだ修正担当者が割り当てられていない場合はあります。このとき、欠けている値は NULLであるべきなのです。

13.3 アンチパターンの見つけ方

あなたや、あなたのチームの誰かが以下のような問題について話をしていたら、NULLの不適切な扱いに起因する「**フィア・オブ・ジ・アンノウン（恐怖の unknown）**」アンチパターンに陥っているかもしれません。

「**assigned_to列（または他の列）で、値が設定されていない行はどうやって見つけたらいい？**」

NULLの行を見つけるために等価演算子を使うことはできません。この章の後半で、 IS NULL述語の使い方を説明します。

「一部のユーザーのフルネームが、アプリケーションでブランク表示されてしまう。でも、データベースにはデータがあるんだ」

NULLを含む文字列を連結したと思われます。この場合に返されるのはNULLです。

「このプロジェクトに費やされた総時間のレポートには、修正したバグの一部しか含まれていない。優先度を割り当てたものしか対象になっていないぞ」

おそらく、時間を合計するための集約クエリのWHERE句には、priorityがNULLの場合にTRUEにならない式が含まれています。非等価比較で意外な結果が返されたときは、注意が必要です。例えば、priorityがNULLの行には、式 priority <> 1はマッチしません。

「Bugsテーブルで不明（unknown）を表すために使用していた文字列が、今後は使えなくなってしまった。会議を開いて、新しい特殊値を何にするか、その値を使うためのデータ移行とコード修正に必要な時間を見積らなければならない」

列のデータ型の定義域において正当な値になる可能性のある値を特別なフラグ値として割り当てると、こうした事態が発生しかねません。フラグとして設定した値が、プロジェクトの経過と共に、その値本来の意味で必要になってしまうことがあるのです。

NULLの取り扱いにおける問題の認識は簡単ではありません。アプリケーションのテストでは、テストデータの設計でエッジケースを想定しない限り、この問題が明らかにはなることはめったにありません。しかし、アプリケーションの稼働後は、データは予測できない様々な形をとります。データにNULLが含まれる可能性がある場合は、何かしらの問題が発生すると想定すべきです。

NULLはリレーショナルか

SQLにおけるNULLについては、ちょっとした論争があります。リレーショナル理論を考案したコンピューター科学者**エドガー・F・コッド**は、欠けている値を表すためのNULLの必要性を認めていました。しかし、同じくコンピューター科学者の**クリス・J・デイト**は、SQL標準で定義されたNULLの振舞いの一部には、リレーショナル理論と矛盾するエッジケースがあることを示しました。

実際には、ほとんどのプログラミング言語は、コンピューター科学理論の完ぺきな実装にはなり得ません。SQL言語はNULLをサポートしていますが、そこには良い点も悪い点もあるのです。この章でも、SQLにおけるNULLのリスクの一部を見てきました。しかし、これらのエッジケースにうまく対応し、効果的にNULLを使う方法を学ぶことは可能なのです。

13.4 アンチパターンを用いてもよい場合

NULLを用いることそれ自体がアンチパターンなのでは**ありません**。NULLを一般値として使用したり、一般値をNULLに相当するものとして扱うことが「フィア・オブ・ジ・アンノウン（恐怖のunknown）」アンチパターンなのです。

NULLを一般的な値として扱わなくてはならない状況の1つは、外部データの読み込みや書き出しをする

場合です。テキストファイルのカンマ区切りフィールドでは、値はすべてテキストとして表現されます。例えば、MySQLでテキストファイルからデータをロードするために使用するmysqlimportツールでは、\NでNULLを表します。

同様に、ユーザーの入力も直接はNULLを表現できません。ユーザー入力を受け取るアプリケーションには、特別な入力内容をNULLにマッピングする方法を提供するものがあります。例えばMicrosoft .NETのバージョン2.0以降では、ウェブユーザーインタフェースのためにConvertEmptyStringToNullと呼ばれるプロパティをサポートしています。このプロパティを持つパラメータとバインドフィールドは、自動的に空文字列をNULLに変換します。

欠けている値を、複数の個別に意味がある値としてサポートする必要があるときは、NULLがうまく機能しない場合があります。例えば、まだ修正担当者が割り当てられていないバグと、既に退職した担当者に割り当てられていたバグとを区別したい場合、それぞれの状態に個別の値を割り当てなくてはなりません。

13.5 解決策：NULLを一意な値として使う

NULLの問題のほとんどは、SQLの**3値論理**の振舞いについてのよくある誤解に基づいています。他のほとんどのプログラミング言語に実装されている従来型の**2値論理**（真偽のロジック）に慣れているプログラマーにとって、3値論理はやっかいな存在です。この仕組みを学ぶことで、SQLクエリでのNULLをうまく取り扱えるようになります。

13.5.1 スカラー式でのNULL

スタンという30歳の男性がいるとします。もう1人の男性オリバーの年齢は不明（unknown）です。スタンがオリバーよりも年上かと尋ねられた場合、唯一の可能な答えは「わかりません」です。同じく、スタンとオリバーが同い年であるかどうか、スタンとオリバーの年齢の合計はいくつかを尋ねられた場合も、答えは「わかりません」としか答えようがありません。

次に、同じく年齢が不明（unknown）なチャーリーという男性がいるとします。オリバーとチャーリーが同い年であるかどうかを尋ねた場合、答えはやはり「わかりません」です。これが、NULL = NULLのような比較の結果がNULLになる理由です。

表13-1に、プログラマーが期待する結果と、予想に反する実際の結果のいくつかの例を示します。

表13-1 スカラー式でのNULL

式	予想した結果	実際の結果	理由
NULL = 0	TRUE	NULL	NULLはゼロではない
NULL = 12345	FALSE	NULL	不明な値が、ある値と等しいかどうかはわからない
NULL <> 12345	TRUE	NULL	不明な値が、ある値と等しくないかどうかはわからない
NULL + 12345	12345	NULL	NULLはゼロではない
NULL || 'string'	'string'	NULL	NULLは空文字列ではない
NULL = NULL	TRUE	NULL	不明な値と不明な値が等しいかどうかはわからない
NULL <> NULL	FALSE	NULL	不明な値と不明な値が等しくないかどうかはわからない

もちろん、これらの例は予約語NULLを使うときのみではなく、値がNULLであるどのような列や式にも当てはまります。

13.5.2 論理式でのNULL

NULLの論理式での振舞いを理解するためのカギは、「NULLはTRUEでもFALSEでもない」という概念です。

表13-2に、先ほどと同様に、プログラマーが期待する結果と、予想に反する実際の結果のいくつかの例を示します。

表13-2 論理式でのNULL

式	予想した結果	実際の結果	理由
NULL AND TRUE	FALSE	NULL	NULLはFALSEではない
NULL AND FALSE	FALSE	FALSE	AND FALSEの真理値はすべてFALSEになる
NULL OR FALSE	FALSE	NULL	NULLはFALSEではない
NULL OR TRUE	TRUE	TRUE	OR TRUEの真理値はすべてTRUEになる
NOT (NULL)	TRUE	NULL	NULLはFALSEではない

確かにNULLはTRUEではありませんが、それはFALSEであることを意味しません。もしそうであれば、NOTをNULLに適用したときの結果がTRUEになるはずです。しかし、そのようにはなりません。NOT(NULL)は、NULLを返します。これは、論理式でNULLを扱う人を混乱させる原因になっています。

13.5.3 NULLの検索

ある値とNULLを比較するとき、等式も不等式もTRUEを返さないため、NULLの検索には他の手段が必要になります。古くからSQL標準で定義されているIS NULL述語は、対象データがNULLの場合TRUEを返します。同様に、その反対のIS NOT NULLは、対象データがNULLの場合にFALSEを返します。

Fear-Unknown/soln/search.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL;
SELECT * FROM Bugs WHERE assigned_to IS NOT NULL;
```

また、SQL-99標準では、比較述語IS DISTINCT FROMが定義されています。この述語は、一般的な非等価演算子<>のように機能します。かつ、対象データがNULLの場合でも、常にTRUEまたはFALSEを返します。IS DISTINCT FROMによって、値の比較前にIS NULLで比較しなければならない面倒な式を書かずに済みます。以下の2つのクエリは同等です。

Fear-Unknown/soln/is-distinct-from.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL OR assigned_to <> 1;
SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM 1;
```

IS DISTINCT FROM述語は、リテラル値またはNULLを渡したいプリペアドステートメントでも使うことができます。

Fear-Unknown/soln/is-distinct-from-parameter.sql

```
SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM ?;
```

IS DISTINCT FROMのサポートは、データベース製品によって異なります。PostgreSQL、IBM DB2、Firebirdはサポートしていますが、OracleとMicrosoft SQL Serverはまだサポートしていません。MySQLはIS DISTINCT FROMと同等な、独自の演算子 <=>を提供しています。

誤った方法が正しい結果を導く例

NULLを許容する列が、偶然、意図した振舞いを見せる場合があります。例を紹介しましょう。

```
SELECT * FROM Bugs WHERE assigned_to <> 'NULL';
```

NULLを許容する列assigned_toが、予約語NULLではなく、文字列'NULL'（引用符に注意）と比較されています。

assigned_toがNULLの場合、それを文字列'NULL'と比較してもTRUEにはなりません。このため、NULL行はクエリ結果から除外されます。つまり、偶然、プログラマーの意図した結果が得られることになります[†5]。

他には、整数型の列を文字列'NULL'と比較する場合がこれに該当します。文字列'NULL'のような文字列の整数値は、ほとんどのデータベース製品でゼロになります。assigned_toの各行の整数値はゼロよりも大きいことが想定されるため、文字列'NULL'とは等しくなりません。結果として、これらの行はクエリ結果に含められます。

このように、NULLを引用符で囲むというよくある間違いをしたことで、偶然にも意図した結果が得られる場合があります。ただし、残念ながらこの偶然が他の検索に当てはまることはありません（例：WHERE assigned_to = 'NULL'）。

13.5.4 列にNOT NULL制約を宣言する

NULLがアプリケーションのポリシーに反する場合や、その列においてNULLが意味をなさない場合には、列にNOT NULL制約を宣言するのが良いでしょう。アプリケーションコードに頼るのではなく、データベースで一貫した制約を強制する方がより良い方法だと言えます。

例えばBugsテーブルのエントリでは、date_reported列、reported_by列、status列が必ず非NULLの値を持つのが妥当です。また、Commentsテーブルなどの子テーブルの列には、既存のバグを参照する、非NULLのbug_id列が必ず含まれるべきです。これらの列に対しては、NOT NULL制約を宣言すべきです。

†5 監訳注：このクエリはMySQLの場合には機能しますが、PostgreSQLの場合にはエラーになります。ERROR: invalid input syntax for integer: "NULL"

すべての列へDEFAULTを定義することを推奨する人もいます。DEFAULTを定義すると、INSERTステートメントで列が省略された場合、その列にはNULLではなくデフォルト値が格納されます。ただしこの方法は、効果的な列とそうでない列があります。例えば、Bugs.reported_byは NULLであってはいけませんが、この列のためにどのようなデフォルト値を宣言できるというのでしょうか。NOT NULL制約を必要としながらも、論理的なデフォルト値を持たない列があることは、正常かつ一般的なことなのです。

13.5.5 動的なデフォルト

クエリによっては、ロジックを単純化するために列や式の値が非NULLであることを強制したくなるかもしれません。とはいえ、あらかじめ値を格納しておきたいわけでもありません。必要なのは、特定のクエリのみで、対象の列や式にアドホックなデフォルト値を設定することです。この場合に便利なのがCOALESCE関数です。この関数は可変長引数を取り、最初の非 NULLの引数を返します。

この章の冒頭で見たユーザーのフルネームの連結の例では、COALESCE関数を使えばミドルネームのイニシャルの代わりに空白文字を使う式を書くことができます。こうすることで、イニシャルがNULLの場合でも、式全体の結果はNULLにはなりません。

Fear-Unknown/soln/coalesce.sql

```
SELECT first_name || COALESCE(' ' || middle_initial || ' ', ' ') || last_name
  AS full_name
FROM Accounts;
```

COALESCEはSQL標準で定義されている関数です。データベース製品によっては、同様の関数を、NVLやISNULLのような名前で提供しています。

データ型を問わず、欠けている値にはNULLを用いるようにしましょう。

14章
アンビギュアスグループ（曖昧なグループ）

知性は可能と不可能を区別する。理性は有意義なものと無意味なものを区別する。
可能ではあるが無意味なものも存在する。
——マックス・ボルン（ドイツ人の理論物理学者）

ある日あなたの上司が、バグデータベース内の進行中のプロジェクトと中止されたプロジェクトを把握する必要が出てきたとしましょう。報告された直近のバグを製品ごとにまとめたレポート生成機能の開発を依頼されたあなたは、同じproduct_idを共有するバグのグループごとに、date_reported列の最大値を計算するMySQLのクエリを書きました。レポートには表14-1のように表示されます。

表14-1　直近のバグを製品ごとにまとめたレポート

product_name	latest	bug_id
Open RoundFile	2010-06-01	1234
Visual TurboBuilder	2010-02-16	3456
ReConsider	2010-01-01	5678

細かい性格の上司は、レポートに列挙されているそれぞれのバグを実際に調べました。そして、レポートで製品「Open RoundFile」の最新のバグであると記載された行が、bug_idを調べると最新ではないことに気づきました。関係するデータをすべて表示すると、不一致がはっきりとわかります（表14-2）。

表14-2　結果の不一致

product_name	date_reported	bug_id
Open RoundFile	2009-12-19	1234
Open RoundFile	2010-06-01	2248
Visual TurboBuilder	2010-02-16	3456
Visual TurboBuilder	2010-02-10	4077
Visual TurboBuilder	2010-02-16	5150
ReConsider	2010-01-01	5678
ReConsider	2009-11-09	8063

この行のbug_idが、
この行の日付と一致していない

この問題を、どのように説明すればよいのでしょうか。ある製品では問題が生じ、別の製品では問題ないのはなぜでしょうか。どうすれば期待するレポートを作成できるでしょうか。

14.1 目的：グループ内で最大値を持つ行を取得する

SQLを学ぶプログラマーは、ある段階で、クエリでGROUP BYを用いて集約関数を行グループに適用し、グループごとに1つの行を取得する方法を習得します。これは少量のコードで複雑なレポートを簡単に作成できる、強力な機能です。

例えば、バグデータベース内で、製品ごとの最新のバグ報告日付を取得するクエリは、以下のようになります。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

自然な発想として、以下のようにクエリを拡張して、製品ごとに最新の報告日付を持つバグのIDを取得したくなります。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

しかし、このクエリはエラーになるか、または信頼性の低い結果を返します。この現象は多くのSQLプログラマーを困惑させます。

この章の目的は、グループ内の最大値（または最小値や平均値）だけでなく、**その最大値が見つかった行の他の属性も取得するクエリ**を実行することです。

14.2 アンチパターン：非グループ化列を参照する

この「**アンビギュアスグループ（曖昧なグループ）**」アンチパターンの根本原因は単純です。そしてその原因は、SQLのグループ化クエリの振舞いに関する多くのプログラマーの誤解を明らかにします。

14.2.1 単一値の原則（Single-Value Rule）

各グループの行は、GROUP BYで指定する1つ以上の列と同じ値を持つ行です。例えば、以下のクエリでは、product_idの個別の値それぞれに1つの行グループが存在します。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

このクエリのSELECT句の選択リストに列挙されるすべての列は、行グループごとに単一の値の行でなければなりません。これは「**単一値の原則**（Single-Value Rule）」と呼ばれます。GROUP BY句で列挙された列は、グループ化される行数にかかわらず、グループごとに単一値であることが保証されます。

MAX関数もまた、各グループで単一値をもたらすことが保証されます。MAX関数はグループ内のすべての行について引数に指定された式を評価し、結果の最大値を返します。

しかし、選択リストに列挙されているものの、GROUP BY句には列挙されていない列では事情が異なります。データベースは、これらの列でグループ内のすべての行に同じ値があることを、常には保証できないのです。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

この例では、あるproduct_idに対応したbug_idには複数の値が存在します。交差テーブルBugsProductsで、製品に複数のバグが関連付けられているためです。製品ごとに1つの行グループに丸めてしまうグループ化クエリでは、行グループ内に含まれるbug_idのすべての値を表す方法がありません。

GROUP BYで指定されていない列では、グループごとの単一値の保証がないため、データベースはそれらの列を単一値の原則に反するものと見なします。ほとんどのデータベース製品では、集約関数の引数に渡す以外の方法で、GROUP BY句に指定されていない列をそのまま返してしまうと、エラーになります。

ただし、MySQLとSQLiteは、他のデータベース製品と振舞いが異なります。詳しくは「アンチパターンを用いてもよい場合」の節で説明します。

14.2.2 SQLがクエリの意図を汲んでくれるとは限らない

プログラマーによくある誤解は、「別の列に対してMAX関数が使われているから、どのbug_idを出力したいのかをSQLが適切に判断してくれるだろう」と考えてしまうことです。ほとんどのプログラマーは、クエリが最大値を取得するときに、ついでに選択リストに指定された他の列の値も、その最大値の存在する同じ行から取得してきてくれると思っています。

しかし残念ながら、以下のような場合、SQLにはグループ内のどの行からbug_idの値を取得すべきか判断することはできません。

- date_reported列で2つのバグが同じ報告日付を持ち、かつその日付がグループの最大値である場合、クエリはどちらのbug_id値を返せばよいのでしょうか。
- クエリで2つの異なる集約関数を実行した場合（例えばMAXとMIN）、集約関数の値はおそらくグループ内の2つの異なる行から取得されるでしょう。クエリはどちらのbug_idを返せばよいのでしょうか。

Groups/anti/maxandmin.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  MIN(date_reported) AS earliest, bug_id
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

- 集約関数で返された値と一致する行がテーブルにない場合、bug_idの値はどうすれば良いでしょうか。このような状況は、AVG、COUNT、SUMなどの関数を使う場合によく見られます。

Groups/anti/sumbyproduct.sql

```
SELECT product_id, SUM(hours) AS total_project_estimate, bug_id
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

これらの例は、単一値の原則がなぜ重要なのかを教えてくれます。この原則に反するクエリが必ずしも曖昧な結果を返すというわけではありませんが、多くの場合は曖昧になるでしょう。もしデータベースが曖昧なクエリと曖昧でないクエリを区別し、データが曖昧であると判断した場合のみエラーを返せるのであれば、それはよいことのように思えるかもしれません。しかし、今度はアプリケーションが不安定になってしまいます。データの状態によって、同じクエリが成功したりエラーになったりしてしまうからです。

14.3 アンチパターンの見つけ方

ほとんどのデータベース製品では、単一値の原則に反するクエリを作成すると、ただちにエラーが返されます。以下は、各種データベース製品で表示されるエラーメッセージの例です。

Firebird 2.1

```
Invalid expression in the select list (not contained in either an aggregate function or the
GROUP BY clause)
```

IBM DB2 9.5

```
An expression starting with "BUG_ID" specified in a SELECT clause, HAVING clause, or ORDER
BY clause is not specified in the GROUP BY clause or it is in a SELECT clause, HAVING
clause, or ORDER BY clause with a column function and no GROUP BY clause is specified.
```

Microsoft SQL Server 2008

```
Column 'Bugs.bug_id' is invalid in the select list because it is not contained in either an
aggregate function or the GROUP BY clause.
```

MySQL 5.5（SQLモードにONLY_FULL_GROUP_BYを指定した場合） †1

```
ERROR 1055 (42000): 'sqlap.Bugs.bug_id' isn't in GROUP BY
```

†1 監訳注：MySQLではSQLモードにONLY_FULL_GROUP_BYを指定することによって（SET sql_mode='ONLY_FULL_GROUP_BY';）、曖昧なクエリを無効にすることができます。http://dev.mysql.com/doc/refman/5.1/ja/group-by-hidden-fields.html

Oracle Database 10g Release 2

```
ORA-00979: not a GROUP BY expression
```

PostgreSQL 9.0

```
ERROR:  column "bugsproducts.bug_id" must appear in the GROUP BY clause or be used in an
aggregate function
```

SQLiteとMySQLでは、曖昧な列には予期しない値や信頼性の低い値が返される可能性があります。MySQLでは、グループの最初の行から値が返されます。この場合の「最初」は、物理的な格納場所を意味します。SQLiteでは、結果は反対になります。つまり、グループの最終行の値が返されます。両方とも、この振舞いは文書化されていません。また、これらのデータベースの将来のバージョンで、同じ振舞いが維持されることは保証されていません。これらのケースに気づき、曖昧さを避けるクエリを設計するのは、プログラマーの責任なのです。

GROUP BY と DISTINCT

クエリ修飾子にDISTINCTキーワード使うと、クエリ結果の行を減らして、すべての行を一意にすることができます。例えば、以下のクエリはバグの報告者と報告日を返しますが、返すのは日付と人ごとに1行のみです。

```
SELECT DISTINCT date_reported, reported_by FROM Bugs;
```

グループ化クエリでは、集約関数を省略することで同じ結果を得られます。クエリ結果は、GROUP BY句で指定された列内の個別の値の組み合わせごとに、1行に絞り込まれます。

```
SELECT date_reported, reported_by FROM Bugs
GROUP BY date_reported, reported_by;
```

両方のクエリは同じ結果を返し、同じように最適化され実行されます。このため、この例の違いは好みの問題に過ぎません。

14.4 アンチパターンを用いてもよい場合

これまでに見てきたように、MySQLとSQLiteでは、単一値の原則に違反する列に対する結果の信頼性を保証できません。ただし、「これらのデータベースでは他のデータベースほど単一値の原則に厳密に準拠していない」という事実を利用できる場合があります。

Groups/legit/functional.sql

```
SELECT b.reported_by, a.account_name
FROM Bugs b INNER JOIN Accounts a ON b.reported_by = a.account_id
GROUP BY b.reported_by;
```

上のクエリでは、account_name列は、GROUP BY句または集約関数で指定されていないために、厳密には単一値の原則に違反しています。しかし、それでも各グループでaccount_name列に入る可能性のある値は1つのみです。各グループは、Accountsテーブルの外部キーである Bugs.reported_byに基づいてグループ化されているため、各グループはAccountsテーブルの行と1対1に対応しているからです。

つまり、reported_byの値を知っていれば、account_nameの値を一意に特定できます。Accountsテーブルの主キーでクエリを実行する場合と同じというわけです。

この種の明確な関係性は、**関数従属性**（functional dependency）と呼ばれます。代表例は、テーブルの主キーとテーブルの属性との関係です。account_nameは、その主キーである account_idに関数従属性があります。テーブルの主キー列（1つまたは複数）でクエリをグループ化する場合、グループはそのテーブルの個別の行に対応し、同じテーブルの他のすべての列はグループごとに単一値を持つことになります。

Bugs.reported_byは Accountsテーブルの主キーを参照するので、Accountsテーブルの主キーに従属する属性に対して、主キーと同様の関係性を持ちます。クエリがreported_by列でグループ化されるとき、reported_by列が外部キーであるために Accountsテーブルの各属性が関数従属となり、クエリ結果に曖昧さは発生しません。

しかし、それでもほとんどのデータベースはエラーを返します。SQL標準で要求される振舞いであるだけではなく、関数従属を期待したクエリであることを即座に判断するのは、特に手間ではないからです[†2]。それでも、MySQLやSQLiteを使用している場合、関数従属性を持つ列のみにクエリを実行するようプログラマー自身が注意していれば、曖昧さの問題を避けつつ、この種のクエリのグループ化を使用できます。

14.5 解決策：曖昧でない列を使用する

以下の節では、このアンチパターンを解決し、曖昧でないクエリを書くための方法をいくつか紹介します。

14.5.1 関数従属性のある列のみにクエリを実行する

最も直接的な解決策は、クエリから曖昧な列を排除することです。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

このクエリは、製品ごとのバグの最新の日付を返します。最新のバグに対応するbug_idは返されませんが、それで十分な場合もあります。シンプルな解決策を見落とさないようにしましょう。

14.5.2 相関サブクエリを使用する

相関サブクエリ（correlated subquery）には外部クエリへの参照が含まれるため、外部クエリの各行に対応する結果が異なります。同じ製品でより新しい日付を持つバグを検索する相関サブクエリを実行するこ

†2 この章で紹介したクエリの例はどれもシンプルなのでプログラマーが関数従属性の有無を判断することができますが、任意のSQLクエリの場合は難しい場合もあるかもしれません。

とで、製品ごとの最新のバグを特定できます。サブクエリが結果を1つも検出しない場合に、外部クエリのバグが最新ということになるわけです。

Groups/soln/notexists.sql

```
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 INNER JOIN BugsProducts bp1 USING (bug_id)
WHERE NOT EXISTS
  (SELECT * FROM Bugs b2 INNER JOIN BugsProducts bp2 USING (bug_id)
   WHERE bp1.product_id = bp2.product_id
     AND b1.date_reported < b2.date_reported);
```

読みやすく、コードの記述も簡単な解決策です。ただしこの方法は、相関サブクエリが外部クエリの各行に対してそれぞれ実行されるため、最善のパフォーマンスは得られないことにも注意が必要です。

14.5.3 導出テーブルを使用する

サブクエリを**導出テーブル**（derived table）として使用すると、product_idと各製品に対応する最新のバグレポート日付のみを含む中間結果を取得できます。次に、この結果を用いてテーブルにJOINさせ、クエリの最終結果に製品ごとの最新日付を持つバグのみが含まれるようにします。

Groups/soln/derived-table.sql

```
SELECT m.product_id, m.latest, b1.bug_id
FROM Bugs b1 INNER JOIN BugsProducts bp1 USING (bug_id)
  INNER JOIN (SELECT bp2.product_id, MAX(b2.date_reported) AS latest
    FROM Bugs b2 INNER JOIN BugsProducts bp2 USING (bug_id)
    GROUP BY bp2.product_id) m
  ON bp1.product_id = m.product_id AND b1.date_reported = m.latest;
```

表14-3 サブクエリで返された日付が複数行と一致する場合の結果

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	3456
2	2010-02-16	5150
3	2010-01-01	5678

サブクエリで返されたlatestの日付が複数行と一致する場合、1つの製品に複数行が取得される可能性があることに注意しましょう（表14-3）。product_idごとに1行のみの取得を保証したい場合、外部クエリでもう1回グループ化関数を使えます。

Groups/soln/derived-table-no-duplicates.sql

```
SELECT m.product_id, m.latest, MAX(b1.bug_id) AS latest_bug_id
```

```
FROM Bugs b1 INNER JOIN
  (SELECT product_id, MAX(date_reported) AS latest
   FROM Bugs b2 INNER JOIN BugsProducts USING (bug_id)
   GROUP BY product_id) m
  ON b1.date_reported = m.latest
GROUP BY m.product_id, m.latest;
```

表 14-4 外部クエリで別のグループ化関数を使った結果

product_id	latest	latest_bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

導出テーブルによる解決策は、相関サブクエリに対する、よりパフォーマンスの高い選択肢として使用しましょう。導出テーブルには外部クエリとの相関がないため、ほとんどのデータベース製品では1回でサブクエリを実行できます。ただし、データベースは導出テーブルの中間結果をテンポラリテーブルに格納しなければならないため、これも最善のパフォーマンスをもたらす解決策ではありません。

14.5.4 JOINを使用する

結合（JOIN）を使用して、存在しないかもしれない行に対する突き合わせ処理を行うことができます。このタイプの結合は、**外部結合**（OUTER JOIN）と呼ばれます。一致する行が存在しない場合、存在しない行の列にはNULLが入ります。クエリで NULLを検出した場合、対象の行が存在しなかったものだとわかります。

Groups/soln/outer-join.sql

```
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1
INNER JOIN BugsProducts bp1
  ON b1.bug_id = bp1.bug_id
LEFT OUTER JOIN (
    Bugs AS b2 INNER JOIN BugsProducts AS bp2
      ON b2.bug_id = bp2.bug_id
  )
  ON (bp1.product_id = bp2.product_id AND (b1.date_reported < b2.date_reported
    OR b1.date_reported = b2.date_reported AND b1.bug_id < b2.bug_id))
WHERE b2.bug_id IS NULL;
```

表 14-5 外部結合を使った結果

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

この内容を理解するためには、しばらくコードを眺めて頭を捻ったり、メモを書いて頭を整理したりしなければならないかもしれません。しかし、ひとたび理解さえできれば、この技法は重要なツールになります。

結合を使用した解決策は、大量のデータに対するパフォーマンスが重要な場合に使用しましょう。この仕組みを理解するのは簡単ではなく、保守も難しくなりますが、概してサブクエリベースの解決策よりもパフォーマンスが優れています。とはいえ、ある方法が別のものよりも優れていると仮定するのではなく、複数種類のクエリのパフォーマンスを実際に測定することが大切です。

14.5.5 他の列に対しても集約関数を使用する

他の列にも集約関数を適用することによって、単一値の原則に従わせることができます。

Groups/soln/extra-aggregate.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  MAX(bug_id) AS latest_bug_id
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

この解決策は、最大の bug_id が最新日付を持つことを当てにできる場合のみに使用します。つまり、バグ報告が時系列順に行われることが保証されている場合です。

14.5.6 グループごとにすべての値を連結する

単一値の原則に準拠するために、bug_idに特殊な集約関数を使うこともできます。MySQL と SQLite は、各グループ内のすべての値を 1 つに連結する関数 GROUP_CONCATをサポートしています。デフォルトでは、この関数はカンマ区切りの文字列を返します（表 14-6）。

Groups/soln/group-concat-mysql.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  GROUP_CONCAT(bug_id) AS bug_id_list
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

表 14-6 GROUP_CONCAT 関数を使い、各グループ内のすべての値を 1 つに連結する

product_id	latest	bug_id_list
1	2010-06-01	1234,2248
2	2010-02-16	3456,4077,5150
3	2010-01-01	5678,8063

このクエリでは、最新日付に対応する bug_id は特定できません。bug_id_list には、各グループの bug_id がすべてが含まれます。

加えて、この解決策には、SQL 標準に準拠していないというデメリットがあります。他のデータベース製品はこの関数をサポートしていません。ただしデータベース製品によっては、カスタム関数とカスタム集約関数をサポートしているものもあります。例えば、PostgreSQL での解決策は、次のようになります。

Groups/soln/group-concat-pgsql.sql

```
CREATE AGGREGATE GROUP_ARRAY (
  BASETYPE = ANYELEMENT,
     SFUNC = ARRAY_APPEND,
     STYPE = ANYARRAY,
  INITCOND = '{}'
);

SELECT product_id, MAX(date_reported) AS latest,
  ARRAY_TO_STRING(GROUP_ARRAY(bug_id), ',' ) AS bug_id_list
FROM Bugs INNER JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

データベース製品のなかには、カスタム関数をサポートしていないものもあります。このため、この解決策を用いるには、非グループ化クエリの結果をループし、値を連結させるためのストアドプロシージャを記述することが必要になるかもしれません。

この解決策は、GROUP BY で指定されていない列について、グループごとに単一値を持つことが期待されてはいるが、制約によっては保証されていないような、単一値の原則に違反する可能性も残っている場合に使いましょう。

曖昧なクエリ結果を避けるために、単一値の原則に従いましょう。

15章
ランダムセレクション

乱数の生成は極めて重要な仕事なので、偶然には任せておけない。
——ロバート・コーブー（アメリカの数学者）

あなたは、広告を表示するウェブアプリケーションを開発しています。ウェブページでは、すべての広告主の広告が平等に表示され、かつ同じ広告を繰り返すことで読者がコンテンツに飽きてしまわないよう、ランダムに広告を表示しなければなりません。

アプリケーション公開後の数日間、広告はランダムに、問題なく表示されていました。しかし、次第に動作が重くなっていきました。数週間後、ウェブサイトがあまりにも遅いという苦情が読者から届くようになりました。実際にページのロード時間を測定したところ、はっきりと遅くなっていることがわかりました。不満を募らせた読者はウェブサイトへの興味を失い、トラフィックも下降してきています。

あなたは過去の経験から、テスト環境のデータベースでテストデータを使用し、プロファイリングツールでパフォーマンスのボトルネックを見つけようとしました。しかし、ウェブページの読み込み時間を測定したところ、ページの生成に使用するSQLクエリのどれにもパフォーマンスの問題は見つかりませんでした。それでも、本番環境のウェブサイトは日を追うごとに遅くなっています。

ようやく、本番環境のウェブサイトのデータベースのデータ量が、テスト環境のテストデータ量よりはるかに多いことに気づきました。本番環境と同規模のデータを用いてデータベースのテストを繰り返したところ、問題は広告選択用のクエリであることがわかりました。選択する広告の数が多くなるに従い、クエリのパフォーマンスが急激に低下していたのです。ボトルネックになっているクエリは特定できました。大きな一歩です。

ウェブサイトが読者とスポンサーを失う前に、ランダムに広告を選択するクエリを修正するには、どうすればよいのでしょうか。

15.1 目的：サンプル行をフェッチする

ランダムな結果を返すSQLクエリが必要になる場面は、驚くほど頻繁にやってきます。プログラミングにおける再現性と決定性の原則に反するように思われますが、大量のデータセットから標本（サンプル）を抽出するような場面では当たり前のことなのです。例えば、次のような場面です。

- ニュースのハイライトや広告のような、コンテンツのローテーション表示
- 一部のデータの監査
- コールセンターにかかってきた電話のオペレーターへの割り当て
- テストデータの生成

いったんアプリケーション側にすべてのデータを取得して、その中からサンプルを選ぶのではなく、直接データベースからクエリでサンプルを抽出する方が良い方法だと言えます。

この章の目的は、データのランダムサンプルのみを返す、効率の良いSQLクエリを書くことです[†1]。

15.2 アンチパターン：データをランダムにソートする

SQLでランダムな行を取得するための最も一般的な方法は、ランダムにソートを行い、最初の行を取得するというものです。この技法はわかりやすく、実装も簡単です[†2]。

Random/anti/orderby-rand.sql

```
SELECT * FROM Bugs ORDER BY RAND() LIMIT 1;
```

これはよく用いられる解決策ですが、時間の経過とともに弱点が現れます。この弱点を考えるために、まず、従来型のソートと比較してみましょう。従来型のソートでは、列の値を比較し、値の大きさによって降順または昇順に行を並び替えます。2回以上行っても同じ結果になるので、このソートには再現性があります。また、このソート方法ならば**インデックス**のメリットも得られます。インデックスとは本質的に、ある列の値を事前にソートした集合であるためです。

Random/anti/indexed-sort.sql

```
SELECT * FROM Bugs ORDER BY date_reported;
```

ソートの基準を行ごとにランダムな値を返す関数にすると、ある行の値が他の行の値より大きいか小さいかにかかわらず、行はランダムにソートされます。このため、ソート結果の順番は各行の値と無関係になり、順番はソートするたびに変わります。ここまでは問題ありません。この方法で、求めている結果が得られるからです。

非決定性を持つ式（RAND関数）によってソートを行うということは、インデックスからメリットを得られないことを意味します。ランダム関数から返された値を含むインデックスは存在しません。実行されるたびに異なり、予測できないからこそ、ランダムなのです。

これはクエリのパフォーマンスにとっては問題となります。インデックスの活用は、ソートを高速化するための最善策の1つだからです。インデックスを使えない場合には、データベースはクエリ結果を「手作業で」ソートしなければならなくなります。これは**テーブルスキャン**と呼ばれる手法で、多くの場合、すべ

†1 数学とコンピューターサイエンスでは、**乱数**と**疑似乱数**を区別します。厳密には、コンピューターが生成できるのは疑似乱数のみです。

†2 監訳注：RAND関数はMySQLの関数です。PostgreSQLでは RAMDOM関数を使います。

ての結果を一時的なテーブルとして保存し、物理的に行を入れ替える処理を行います。テーブルスキャンによるソートは、インデックスを用いたソートよりはるかに遅くなります。データセットのサイズが大きくなるにつれて、このパフォーマンスの違いはさらに大きくなります。

ランダムなソートのもう1つの弱点は、必要な行が最初の1行のみであるため、データセット全体をソートする労力のほとんどが無駄になってしまう点です。1行をランダムに取得するためだけに毎回大量の行をソートするのは、極めて非効率です。

これらの弱点は、少量のデータに対してクエリを実行するときには特に目立ちません。このため、開発期間やテスト期間では、特に問題があるとは見なされないことがあります。しかし、本番稼働後にデータベースのデータ量が増えていくにつれ、問題が現れ始めます。

15.3 アンチパターンの見つけ方

このアンチパターンで用いられる技法はシンプルです。多くのプログラマーは記事で読むか、あるいは自分で思いついて、この技法を使用します。次のような発言を耳にしたら、この「**ランダムセレクション**」アンチパターンが使われているかもしれません。

「SQLは、行をランダムに返そうとすると本当に遅くなるな」

ランダムにサンプル抽出を行うクエリは、開発やテストの期間で用いる少量のデータに対してはうまく機能しますが、データが増えるにつれて遅くなります。この方法を使用している限り、データベースサーバーのチューニングやインデックス、キャッシュなどの方法では、スケーラビリティの問題を根本的に改善することはできません。

「アプリケーションへのメモリ割り当てを増やしたいんだけど。いったんすべての行をアプリケーション上にフェッチしないと、ランダムに1行を選択できないんだ」

アプリケーションにすべての行を読み込むような非効率的な手法は、使用するべきではありません。また、いくらアプリケーションメモリを拡張したとしても、物理メモリサイズ以上にデータが増えてしまう可能性は残ります。

「一部のエントリは、他に比べて表示される頻度が高いように見える。この乱数発生器は平等に乱数を生成しているのだろうか？」

乱数が、データベースの主キー値の**欠番**をうまく扱えていません（欠番については解決策の「欠番の穴の後にあるキー値を選択する」の項を参照してください）。

15.4 アンチパターンを用いてもよい場合

ランダムなソートによって生じる非効率性の問題は、データセットが小さい場合には特に問題になりません。例えば、プログラマーにバグ修正担当を割り当てる場合などには、ランダムなソートを用いてもよいでしょう。特別にスケーラブルな方法を採用しなければならないほど、1つのプロジェクトに多くのプログラマーがいるとは、まず考えられないからです。

アメリカ50州のリストからランダムに州を選択したい場合も同じです。リストのサイズは特に大きくはなく、州の数が徐々に増えていくこともないでしょう（少なくとも私たちが生きている間には）。

15.5 解決策：特定の順番に依存しない

ランダムなソートを用いたクエリでは、テーブルスキャンが発生し、インデックスを使えない高コストなソートが行われてしまいます。アプリケーション側ではなくSQLでの解決策を検討する際には、こういった非効率なクエリに注意しましょう。最適化の効果がないクエリにいくら最適化を試みても、努力が無駄になるだけです。こういう場合は、アプローチを変えるべきです。以降の節では、別のやり方でクエリの結果からランダムに行を1つ選択する技法をいくつか紹介します。これらの解決策を用いることで、どのような状況下であっても、効率的にランダムな結果を得られます。

15.5.1 1と最大値の間のランダムなキー値を選択する

テーブル全体のソートを回避する1つの方法は、1から主キーの最大値までの間の値をランダムに選択することです。

Random/soln/rand-1-to-max.sql

```
SELECT b1.*
FROM Bugs AS b1
INNER JOIN (
  SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS rand_id
) AS b2 ON b1.bug_id = b2.rand_id;
```

この解決策は、主キー値が1から開始され、かつ連続していることを前提としています。すなわち、1から最大値までの間に、欠けている数値があってはいけません。欠番がある場合、ランダムに選ばれた値が行と一致しない可能性があります[†3]。

1から主キーの最大値までの間のすべての値が使用されていることが確実な場合にのみ、この解決策を使用しましょう。

15.5.2 欠番の穴の後にあるキー値を選択する

先ほどの解決策と似ていますが、1と最大キー値の間に**欠番**があり、かつ乱数によって欠番のキー値が算出されてしまう場合に、このクエリは欠番の穴の後にあるキー値を使います。

Random/soln/next-higher.sql

```
SELECT b1.*
FROM Bugs AS b1
INNER JOIN (
  SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS bug_id
) AS b2 ON b1.bug_id >= b2.bug_id
ORDER BY b1.bug_id
LIMIT 1;
```

†3 監訳注：ランダム値に一致する主キー値が欠番だった場合、結果がゼロ行になります。

この方法ならば、乱数に一致するキー値がない場合の問題を解決できます[†4]。ただし、欠番の1つ上のキー値が選択される頻度が高くなるという特性があります。ランダムな値はほぼ均等に生成されるべきですが、算出されるbug_id値は均等にはなりません。

この解決策は、欠番がめったにない場合や、キー値が均等に選ばれることが、さほど重要ではない場合に使用しましょう。

15.5.3 すべてのキー値のリストを受けとり、ランダムに1つを選択する

アプリケーションコードを用いて、結果セットの主キーから値を1つ選択する技法もあります。選択したら、その主キー値を用いてデータベースから該当する行全体を取得します。以下にPHPコードでこの技法を用いた場合のサンプルを示します。

Random/soln/rand-key-from-list.php

```
<?php
$bug_id_list =
    $pdo->query("SELECT bug_id FROM Bugs")->fetchAll(PDO::FETCH_ASSOC);

$rand = rand( 0, count($bug_id_list) - 1 );
$rand_bug_id = intval($bug_id_list[$rand]['bug_id']);

$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id = ?");
$stmt->bindValue(1, $rand_bug_id, PDO::PARAM_INT);
$stmt->execute();
$rand_bug = $stmt->fetch();
```

この技法はテーブル全体のソートを回避できますし、各キー値もほぼ平等に選択されます。ただし、この解決策には他の弱点があります。

- データベースからすべてのbug_id値を取得すると、リストのサイズが非常に大きくなってしまう可能性があります。アプリケーションメモリのリソースを越え、以下のようなエラーが返されることがあります。

  ```
  Fatal error: Allowed memory size of 16777216 bytes exhausted
  ```

- クエリを2回実行する必要があります。1回目は主キーのリストの取得、2回目はランダムに選択された主キーの行の取得です。クエリがあまりにも複雑で非効率な場合は問題になるでしょう。

この解決策は、単純なクエリを用いて、ほどほどのサイズの結果セットからランダムな行を選択するときに使用しましょう。特に、非連続なリストからランダムに行を選択したい場合に便利な技法です。

†4 監訳注：先ほどの例と異なり、導出テーブルb2と等値結合ではなく**非等値結合**を行っている点に注意してください。

15.5.4 オフセットを用いてランダムに行を選択する

ここまでに挙げた解決策の弱点を回避するために、データセットの行数をカウントし、0と行数までの間の乱数を返す技法もあります。データセットにクエリを実行する際、その数をオフセット値として使用します。

Random/soln/limit-offset.php

```
<?php
$rand_sql = "SELECT ROUND(
    RAND() * (SELECT COUNT(*) FROM Bugs)
  ) AS id_offset";
$result = $pdo->query($rand_sql)->fetch(PDO::FETCH_ASSOC);
$offset = intval($result['id_offset']);

$sql = "SELECT * FROM Bugs LIMIT 1 OFFSET :offset";
$stmt = $pdo->prepare($sql);
$stmt->bindValue(':offset', $offset, PDO::PARAM_INT);
$stmt->execute();
$rand_bug = $stmt->fetch();
```

この解決策では、SQL標準には無いLIMIT句を使用しています。LIMIT句は、MySQL、PostgreSQL、SQLiteでサポートされています。

Oracle、Microsoft SQL Server、IBM DB2では、ウィンドウ関数（window function）であるROW_NUMBER関数を用いて同等の処理を行えます。

Oracleを使う場合のコードは次のようになります。

Random/soln/row_number.php

```
<?php
$rand_sql = "SELECT 1 + MOD(ABS(dbms_random.random()),
  (SELECT COUNT(*) FROM Bugs)) AS id_offset FROM dual";
$result = $pdo->query($rand_sql)->fetch(PDO::FETCH_ASSOC);
$offset = intval($result['id_offset']);

$sql = "WITH NumberedBugs AS (
  SELECT b.*, ROW_NUMBER() OVER (ORDER BY bug_id) AS RN FROM Bugs b
) SELECT * FROM NumberedBugs WHERE RN = :offset";
$stmt = $pdo->prepare($sql);
$stmt->bindValue(':offset', $offset, PDO::PARAM_INT);
$stmt->execute();
$rand_bug = $stmt->fetch();
```

この解決策は、キー値が連続していることを前提にできず、かつ各行が平等に選択される必要がある場合

に使用しましょう。

15.5.5 ベンダー依存の解決策

ほとんどのデータベース製品には、この章の目的を実現するための独自の解決策が実装されています。例えば、Microsoft SQL Server 2005 では TABLESAMPLE句が使えます。

Random/soln/tablesample-sql2005.sql

```
SELECT * FROM Bugs TABLESAMPLE (1 ROWS);
```

Oracle では、TABLESAMPLE句とは若干異なる SAMPLE句が使用できます。例えば、テーブルの全行の1パーセントを返すクエリを書けます。

Random/soln/sample-oracle.sql

```
SELECT * FROM (SELECT * FROM Bugs SAMPLE (1)
ORDER BY dbms_random.value) WHERE ROWNUM = 1;
```

各データベース固有の解決策についての詳細は、各データベース製品のドキュメントを参照してください。それぞれに特有の制限やオプションがあるので、注意が必要です。

クエリには最適化できないものもあります。最適化できない場合は、別のアプローチを採用しましょう。

16章
プアマンズ・サーチエンジン（貧者のサーチエンジン）

問題にぶつかると「正規表現を使えばいい」と考える人がいる。そして、問題を2つ抱えることになる。
——ジェイミー・ザウィンスキー（XEmacsやNetscapeブラウザの開発で名を馳せたプログラマー）

私は1995年頃にテクニカルサポートの仕事をしていたことがあります。企業がウェブを使って顧客に情報を提供し始めようとしていた時代です。私たちも、よくある質問への回答をまとめた文書群を、ウェブのナレッジベースアプリケーションに掲載したいと考えました。

質問と回答が増えるにつれ、検索機能の必要性が明らかになってきました。利用者は、必要な回答を探すために、何百件もの記事に目を通すことを望まないはずです。記事をカテゴリー別に分類するという案もありましたが、分類したとしても記事数は膨大であり、また複数のカテゴリーにまたがる記事の分類が難しいという問題もありました。

やはり、利用者自身が条件に合った記事を絞り込む検索機能が必要でした。考えられる最も柔軟でシンプルなインタフェースは、ユーザーが1つ以上の語を入力し、その語が含まれる記事が表示されるというものでした。また、検索語との一致度が高い記事を上位に表示する必要もありました。加えて、語形変化への対応も検討しなければなりませんでした。例えば、crashという語で検索すると、crashed、crashes、crashingなどもヒットさせたいのです。当然ながら、記事が今後増加してもウェブアプリケーションの検索処理が重くならないことも、重要な要件でした。

今の話を、必要以上に細かいと感じた人がいるかもしれません。オンラインでのテキスト検索はあまりにも一般的になったため、利用できなかった時代のことを簡単には思い出せないほどです。しかし、SQLでキーワード検索を高速かつ正確に行うのは、簡単そうに見えて、実はそうでもないのです。

16.1 目的：全文検索を行う

テキストを格納するアプリケーションでは、語や句によるテキスト検索が必要になります。近年のデータベースには、ますます多くのテキストデータが格納されるようになってきており、合致するテキストを高速で取得することも求められています。特にウェブアプリケーションのテキスト検索では、パフォーマンスとスケーラビリティに優れたデータベース技術が必要です。

SQL（およびSQLの背後にあるリレーショナル理論）の原則の1つは、列の値がアトミックであること、

つまり値が別の値と比較可能であることです。ただし、比較の際には常に値全体が比較されます。部分文字列の比較は、SQLにおいては非効率性や不正確さにつながるのです。

しかし現実の世界では、短い文字列と長い文字列を比較し、長い文字列のなかに短い文字列が存在するかどうかを検出する方法が必要になります。SQLを用いて問題を切り抜けるにはどうすればよいのでしょうか。

16.2 アンチパターン：パターンマッチ述語を使用する

SQLには、文字列比較のための**パターンマッチ述語**があります。ほとんどのプログラマーは、キーワード検索を実装する際に、まずパターンマッチ述語を使うことを考えます。最も一般的なものは、LIKE述語です。

LIKE述語は、ゼロ個以上の文字と一致するワイルドカード（%）を扱うことができます。キーワードの前後にこのワイルドカードを指定すると、その語が含まれるすべての文字列がマッチします。キーワードの前後のワイルドカードは、それぞれキーワードの前後のあらゆるテキストとマッチします。

Search/anti/like.sql

```
SELECT * FROM Bugs WHERE description LIKE '%crash%';
```

多くのデータベース製品は、**正規表現**も独自の方法でサポートしています[†1]。正規表現を使うと、どのような部分文字列に対してもパターンマッチを行えるため、ワイルドカードは不要になります。MySQLの正規表現述語を使った例を見てみましょう。

Search/anti/regexp.sql

```
SELECT * FROM Bugs WHERE description REGEXP 'crash';
```

パターンマッチ述語の最も大きな問題点は、パフォーマンスの低下です。パターンマッチ述語は従来型の**インデックス**のメリットを得られないため、テーブルのすべての行をスキャンしなければなりません。文字列型の列に対するパターンマッチ処理は、例えば2つの整数の等価性比較よりもはるかに大きなコストがかかるため、検索時の**テーブルスキャン**には全体としてかなりの処理量が必要になります。

2番目の問題点は、LIKEや正規表現を用いた単純なパターンマッチでは、意図しないマッチが生じてしまうことです。

Search/anti/like-false-match.sql

```
SELECT * FROM Bugs WHERE description LIKE '%one%';
```

このクエリは単語「one」を含むテキストとマッチしますが、「money」、「prone」、「lonely」などもマッチしてしまいます。前後をスペースで区切ったキーワードで検索すると、今度は前後に句読点がある語や、

†1 SQL-99規格では正規表現のパターンマッチングのために SIMILAR TO述語を定義していますが、ほとんどのデータベース製品では独自の構文が使用されています。

文の冒頭または末尾にある語とマッチしなくなります。この問題を解決するために、単語境界のための特別なパターンをサポートしているデータベースもあります[†2]。

Search/anti/regexp-word.sql

```
SELECT * FROM Bugs WHERE description REGEXP '[[:<:]]one[[:>:]]';
```

このように、パフォーマンス、スケーラビリティ、処理の煩雑さなどの問題を考えると、キーワード検索のために単純なパターンマッチングを使用するのは、決して良い方法とは言えないのです。

16.3 アンチパターンの見つけ方

次のような質問を耳にすることがあれば、「**プアマンズ・サーチエンジン（貧者のサーチエンジン）**」アンチパターンが使われているかもしれません。

「LIKE述語の２つのワイルドカードの間に変数を挿入する方法を知らない？」

この問題は、プログラマーがユーザーからの入力内容を使用してパターンマッチ検索をしたいと考えているときに生じます。

「複数の語を含む文字列や、特定の語を含まない文字列、特定の語の語形のどれかを含む文字列などをチェックする正規表現はどうやって書けばいい？」

正規表現で解決するには難易度の高い複雑な問題がある場合、おそらくこのアンチパターンに陥っています。

「このウェブサイトの検索機能は、データベースにドキュメントデータが増えると、許容できないほど遅くなってしまう。何が原因だろう？」

データ量が増加すると、このアンチパターンのスケーラビリティ上の弱点が明らかになります。

16.4 アンチパターンを用いてもよい場合

アンチパターンの節で見たクエリは、れっきとしたSQLクエリです。シンプルな用途に対して正しく使用すれば、大きな価値を得ることができます。

パフォーマンスはデータベースにおいて常に悩ましい問題ですが、使用頻度が極めて低いので最適化に労力をかける意味がないクエリもあります。めったに使用しないクエリのためにインデックスを更新するのは、その非効率なクエリをインデックス無しで実行するのと同じくらいのコストがかかります。クエリがアドホックな性質を持つ場合、定義したインデックスがそのクエリにメリットをもたらすかどうかも保証できません。

複雑なクエリのためにパターンマッチ述語を使用すると困難に直面しますが、単純なケースのためにパターンマッチを行う場合には、弱点を最小限に抑えつつ、十分な結果を得ることができるでしょう。

†2　監訳注：この例ではMySQLのREGEXP述語を使用しています。

16.5 解決策：適切なツールを使用する

最善の方法は、SQLの代わりに専用の**全文検索**エンジンを使うことです。次善の策は、繰り返しのコストを減らすために検索結果を保存することです。

以降の節では、各種データベース製品に拡張機能として組み込まれている機能や、独立したプロジェクトとして提供されている技術の一部を紹介します。また最後には、SQL標準に準拠しつつも、部分文字列マッチングより効果的な解決策についても見ていきます。

16.5.1 ベンダー拡張

主要なデータベース製品は、全文検索というありふれた要件に対する独自の解決策を用意しています。ただしこれら独自機能は標準化されておらず、ベンダー間での互換性もありません。使用するデータベース製品が1つである（またはベンダー依存の機能を使いたい）場合には、これらの機能は、SQLクエリと親和性が高く、かつ高パフォーマンスなテキスト検索を行うための最善策だと言えるでしょう。

以下に、いくつかのデータベース製品における全文検索機能の概要を示します。細かい点は変更される場合がありますので、必ず各データベース製品の最新ドキュメントを確認するようにしてください。

16.5.2 MySQLのフルテキストインデックス

MySQLでは、MyISAMストレージエンジンのみがサポートする、シンプルなフルテキストインデックスが提供されています[†3]。フルテキストインデックスを定義できるのは、CHAR、VARCHAR、TEXT型の列です。Bugsテーブルのsummary列とdescription列の内容を含むフルテキストインデックスの定義を見てみましょう。

Search/soln/mysql/alter-table.sql

```
ALTER TABLE Bugs ADD FULLTEXT INDEX bugfts (summary, description);
```

インデックスに格納されたテキストからキーワードを検索するには、MATCH関数を用います。MATCH関数を使うときには、フルテキストインデックスに関連付けられた列名を指定する必要があります。列名は複数指定することができるので、同じテーブルの他の列に対して定義されたインデックスも同時に使えます。

Search/soln/mysql/match.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description) AGAINST ('crash');
```

MySQL4.1以降では、より詳細に結果を絞り込むために、パターンにシンプルな論理式記法を使えます。

Search/soln/mysql/match-boolean.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description)
  AGAINST ('+crash -save' IN BOOLEAN MODE);
```

†3 監訳注：MySQL5.6からはInnoDBストレージエンジンでもフルテキストインデックスが使えるようになりました。

16.5.3 Oracleでのテキストインデックス

Oracleは、1997年のOracle8 Server以降、テキストインデックスをサポートしています（当初は、ConTextと呼ばれるデータカートリッジの一部でした）。このテクノロジーは数度にわたってバージョンアップされ、現在はデータベースに統合されています。Oracleのテキストインデックス機能は、複雑で多機能なものです。ここでは簡単に概要を紹介します。

CONTEXT

このタイプのインデックスは単一のテキスト列に対して作成します。インデックスを用いた検索には、CONTAINS演算子を使用します。このインデックスはデータ変更に対する一貫性が維持されませんが、インデックス定義時にPARAMETERS ('SYNC (ON COMMIT)')を追加すると、自動的に内容が同期されるようになります[†4]。

Search/soln/oracle/create-index.sql

```
CREATE INDEX BugsText ON Bugs(summary) INDEXTYPE IS CTXSYS.CONTEXT;
SELECT * FROM Bugs WHERE CONTAINS(summary, 'crash') > 0;
```

CTXCAT

短いテキスト（オンラインカタログなどで使用されているもの）に特化したインデックスで、同じテーブルの他の列と組み合わせて構造化されます。インデックス化対象のデータがトランザクションによって更新された場合でも、インデックスの一貫性は維持されます。

Search/soln/oracle/ctxcat-create.sql

```
CTX_DDL.CREATE_INDEX_SET('BugsCatalogSet');
CTX_DDL.ADD_INDEX('BugsCatalogSet', 'status');
CTX_DDL.ADD_INDEX('BugsCatalogSet', 'priority');
CREATE INDEX BugsCatalog ON Bugs(summary) INDEXTYPE IS CTXSYS.CTXCAT
  PARAMETERS('BugsCatalogSet');
```

CATSEARCH演算子には、検索するテキスト列、テキスト検索条件、および構造化条件式を指定します。

Search/soln/oracle/ctxcat-search.sql

```
SELECT * FROM Bugs
WHERE CATSEARCH(summary, '(crash save)', 'status = "NEW"') > 0;
```

CTXXPATH

XMLドキュメントをexistsNode関数を用いて検索する用途に特化したインデックスです。

Search/soln/oracle/ctxxpath.sql

```
CREATE INDEX BugTestXml ON Bugs(testoutput) INDEXTYPE IS CTXSYS.CTXXPATH;
```

†4 監訳注：つまり自動的に同期を取るには、例えば次のように宣言します。CREATE INDEX BugsText ON Bugs(summary) INDEXTYPE IS CTXSYS.CONTEXT PARAMETERS ('SYNC (ON COMMIT)');

```
SELECT * FROM Bugs
WHERE testoutput.existsNode('/testsuite/test[@status="fail"]') > 0;
```

CTXRULE

データベースで大量の文書を扱い、それらの文書を内容に基づいて分類する必要があるとします。CTXRULE型のインデックスがあれば、大量の文書の解析ルールを設計し、分類結果を確認できます。または、分類ルールを示す文書のサンプルセットをOracleに渡し、残りの文書群に適用するルールをOracleに設計させることもできます。もしくは、これらの作業をすべて自動化し、Oracleに文書の分析を任せて、ルールの設定とそれに基づく分類を行わせることもできます。

CTXRULEインデックスの使用例については、他の書籍や記事を参照してください。

16.5.4 Microsoft SQL Serverでの全文検索

SQL Server 2000以降では、全文検索がサポートされています。言語、シソーラス、データ変更時の自動同期などに関する複雑な構成オプションを使用できます。SQL Serverでは、フルテキストインデックス作成用のストアドプロシージャや、フルテキストインデックスをクエリから使うためのCONTAINS述語などが提供されています。

例えば、crashという語を含むバグを検索するためには、まず最初に全文検索機能を有効化し、データベースにカタログを定義します。

Search/soln/microsoft/catalog.sql

```
EXEC sp_fulltext_database 'enable'
EXEC sp_fulltext_catalog 'BugsCatalog', 'create'
```

次に、Bugsテーブルでフルテキストインデックスを定義し、列を追加してから有効化します。

Search/soln/microsoft/create-index.sql

```
EXEC sp_fulltext_table 'Bugs', 'create', 'BugsCatalog', 'bug_id'
EXEC sp_fulltext_column 'Bugs', 'summary', 'add', '2057'
EXEC sp_fulltext_column 'Bugs', 'description', 'add', '2057'
EXEC sp_fulltext_table 'Bugs', 'activate'
```

フルテキストインデックスの自動同期を有効化し、インデックス列に対する変更がインデックスに反映されるようにします。次にインデックスの生成プロセスを開始します。生成はバックグラウンドで実行されるため、完了までしばらく時間がかかる場合があります。完了後は、クエリでインデックスのメリットが得られるようになります。

Search/soln/microsoft/start.sql

```
EXEC sp_fulltext_table 'Bugs', 'start_change_tracking'
EXEC sp_fulltext_table 'Bugs', 'start_background_updateindex'
EXEC sp_fulltext_table 'Bugs', 'start_full'
```

すべて終わると、CONTAINS述語を用いたクエリを実行できるようになります。

Search/soln/microsoft/search.sql

```
SELECT * FROM Bugs WHERE CONTAINS(summary, '"crash"');
```

16.5.5 PostgreSQLでのテキスト検索

PostgreSQL 8.3では、高度な設定が可能な洗練された全文検索機能によって、文書を検索可能な語要素に変換し、パターン検索を行うことができます。

パフォーマンスを最適化するには、オリジナルのテキスト形式に加えて、テキスト検索可能な特別なデータ型TSVECTORを用いてコンテンツを格納する必要があります。

Search/soln/postgresql/create-table.sql

```
CREATE TABLE Bugs (
  bug_id      SERIAL PRIMARY KEY,
  summary     VARCHAR(80),
  description TEXT,
  ts_bugtext  TSVECTOR
  -- 他の列 . . .
);
```

TSVECTOR列は、検索対象のテキスト列の内容と常に同期する必要があります。PostgreSQLでは、同期作業をより容易にするために、組込みのトリガーが提供されています。

Search/soln/postgresql/trigger.sql

```
CREATE TRIGGER ts_bugtext BEFORE INSERT OR UPDATE ON Bugs
FOR EACH ROW EXECUTE PROCEDURE
  tsvector_update_trigger(ts_bugtext, 'pg_catalog.english', summary, description);
```

さらに、TSVECTOR列に対して **GIN（汎用転置インデックス）** を作成しなければなりません。

Search/soln/postgresql/create-index.sql

```
CREATE INDEX bugs_ts ON Bugs USING GIN(ts_bugtext);
```

これで、PostgreSQLのテキスト検索演算子「@@」を用いて、フルテキストインデックスを活用した効果的な全文検索が行えるようになります。

Search/soln/postgresql/search.sql

```
SELECT * FROM Bugs WHERE ts_bugtext @@ to_tsquery('crash');
```

他にも、検索可能なコンテンツ、検索クエリ、検索結果をカスタマイズする多くのオプションがあります。

16.5.6 SQLiteでの全文検索（FTS）

SQLiteの標準的なテーブルは、効率的な全文検索をサポートしていません。しかし、SQLiteの拡張機能を使うことで、検索可能なテキストを専用の**仮想テーブル**に格納できます。検索可能なテキスト拡張には、FTS1、FTS2、FTS3の3つがあります。

FTS拡張は、SQLiteのデフォルトビルドでは有効化されていないため、ソースコードからSQLiteをビルドしてFTS拡張を有効化する必要があります。例えば、Makefile.inに以下のオプションを追加して、SQLiteをビルドします。

Search/soln/sqlite/makefile.in

```
TCC += -DSQLITE_CORE=1
TCC += -DSQLITE_ENABLE_FTS3=1
```

SQLiteでFTSを有効化した後は、テキスト検索を行うための仮想テーブルを作成できます。仮想テーブル作成の際には、データ型や制約などの列オプションは無視されます。

Search/soln/sqlite/create-table.sql

```
CREATE VIRTUAL TABLE BugsText USING fts3(summary, description);
```

この例のBugsテーブルのように、別のテーブルのテキストに対するインデックスを作成する場合、データを仮想テーブルにコピーしなければなりません。FTS仮想テーブルには常に主キーdocidが定義されているため、行をコピー元テーブルの行と関連付けることができます。

Search/soln/sqlite/insert.sql

```
INSERT INTO BugsText (docid, summary, description)
  SELECT bug_id, summary, description FROM Bugs;
```

これで、効率的な全文検索述語MATCHを用いてFTSの仮想テーブル BugsTextにクエリを実行できます。また、マッチした行を元テーブルであるBugsと結合できます。FTSテーブルの名前を疑似列のように使うと、FTSテーブル上のすべての列と一度にパターンマッチングを行うことも可能です。

Search/soln/sqlite/search.sql

```
SELECT b.* FROM BugsText t INNER JOIN Bugs b ON t.docid = b.bug_id
WHERE BugsText MATCH 'crash';
```

また、いくつかの論理式がパターンマッチングに使えます。

Search/soln/sqlite/search-boolean.sql

```
SELECT * FROM BugsText WHERE BugsText MATCH 'crash -save';
```

16.5.7 サードパーティーのサーチエンジン

データベース製品の独自機能に依存せずにテキストを全文検索したい場合、SQLデータベースから独立して動作する検索エンジンが必要になります。ここからは、データベースから独立した検索エンジン、Sphinx SearchとApache Luceneの2つを簡単に紹介します。

Sphinx Search

Sphinx Searchはオープンソースの検索エンジンで、MySQLやPostgreSQLとうまく連携できます[†5]。本書の執筆時点では、Sphinx Searchをオープンソースデータベースの Firebirdで使用するための非公式パッチも存在します。将来的には、他のデータベースでもサポートされることが考えられます。

Sphinx Searchはインデックスの作成と検索が高速で、分散クエリもサポートしています。高いスケーラビリティが要求される検索アプリケーションを、更新頻度が低いデータに対して構築する用途に適しています。

Sphinx SearchでMySQLデータベースに格納されたデータに対するインデックス作成を行うためには、設定ファイルsphinx.confの中のデータベース指定フィールドを設定します（サンプルコード内のtypeなど）。インデックス構築には、データを取得するSQLクエリも必要です（sql_query）。このクエリで指定する最初の列は整数型の主キーです。また、結果の絞り込みやソートのための属性列を宣言することもできます。残りの列は、フルテキストインデックスの対象となる列です。最後にもう1つSQLクエリを設定し（sql_query_info）、主キー値が$idとなる行全体をデータベースから取得できるようにします。

Search/soln/sphinx/sphinx.conf

```
source bugsrc
{
  type = mysql
  sql_user = bugsuser
  sql_pass = xyzzy
  sql_db = bugsdatabase
  sql_query = \
    SELECT bug_id, status, date_reported, summary, description \
    FROM Bugs
  sql_attr_timestamp = date_reported
  sql_attr_str2ordinal = status
  sql_query_info = SELECT * FROM Bugs WHERE bug_id = $id
}
```

†5 http://sphinxsearch.com/

```
index bugs
{
  source = bugsrc
  path = /opt/local/var/db/sphinx/bugs
}
```

sphinx.confの設定を終えると、indexerコマンドでインデックスを作成できます。

Search/soln/sphinx/indexer.sh

```
indexer -c sphinx.conf bugs
```

インデックス作成が終わると、searchコマンドでインデックスを検索できます。

Search/soln/sphinx/search.sh

```
search -b "crash -save"
```

Sphinx Searchにはデーモンプロセスと APIがあり、PHP、Perl、Rubyなどのスクリプト言語から検索を行えます。現行バージョンの主な短所は、このインデックスアルゴリズムが増分更新を効率的に行えない点です。このため、更新頻度の高いデータに対して Sphinx Searchを用いる場合には妥協が必要です。例えば、検索対象テーブルの分割です。片方のテーブルには、データ量が多いものの更新が不要な履歴データを格納し、もう一方のテーブルには、データ増加に伴うインデックス再構築が必要な少量の最新データを格納します。そして、この2つの Sphinx Searchインデックスに対してアプリケーションから検索を実行するのです。

Apache Lucene

Apache Luceneは、Javaアプリケーション向けの成熟した検索エンジンです[†6]。ほぼ同様の内容で C++、C#、Perl、Ruby、Python、PHPなどの言語向けのプロジェクトも存在します。

Luceneは、テキストドキュメントの集合に対して、独自形式でインデックスを構築します。Luceneインデックスは、インデックスの作成元のデータに対して同期を取らないため、列の挿入、削除、更新を行った場合、その変更を Luceneインデックスに適用しなければなりません。

Luceneの検索エンジンは車のエンジンに似ています。有効に使うためには、そのまわりの様々な技術が求められるのです。Luceneは SQLデータベースからデータを直接読み込まないため、Luceneインデックスにドキュメントを書き込む必要があります。例えば、SQLクエリを実行し、結果の各行に対して1つの Luceneドキュメントを作成し、それを Luceneインデックスに保存します。Luceneは、Java API経由で使うことができます。

幸い、Apacheには Solrと呼ばれる補完的なプロジェクトもあります[†7]。Solrは Luceneインデックスに

†6 http://lucene.apache.org/
†7 http://lucene.apache.org/solr/

ゲートウェイを提供するサーバーです。Solrの REST風インタフェースを通じてドキュメントの追加や検索を行えるため、様々なプログラミング言語からLuceneインデックスを使用できます。

また、Solrの DataImportHandlerツールを使用すると、Solr自身でデータベースに接続し、クエリを実行して結果をインデックスに取り込むことができます。

転置インデックスの自作

データベースベンダー依存の検索機能や、個別の検索エンジン製品のインストールを行わず、データベース中立かつ効率的なテキスト検索を行いたい場合もあります。ここからは、**転置インデックス**と呼ばれるインデックスの設計を行います。転置インデックスとは基本的に、検索対象になりる可能性があるすべての語のリストです。転置インデックスは語と語を含むテキストとの多対多関連を作ります。例えば、crashという語は複数のバグにマッチしますが、各バグは他の複数のキーワードともマッチします。では、転置インデックスの設計を実際に見ていきましょう。

最初に、ユーザーが検索するキーワードの一覧を格納するテーブル Keywordsを定義し、次に、多対多関連の構築のために交差テーブル BugsKeywordsを定義します。

Search/soln/inverted-index/create-table.sql

```
CREATE TABLE Keywords (
  keyword_id  SERIAL PRIMARY KEY,
  keyword     VARCHAR(40) NOT NULL,
  UNIQUE KEY (keyword)
);

CREATE TABLE BugsKeywords (
  keyword_id  BIGINT UNSIGNED NOT NULL,
  bug_id      BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (keyword_id, bug_id),
  FOREIGN KEY (keyword_id) REFERENCES Keywords(keyword_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

次に、あるバグの説明文に含まれるすべてのキーワードを、BugsKeywordsに行として加えます。キーワードの抽出には、LIKE述語や正規表現を用いた部分文字列マッチを使用できます。この抽出方法は「アンチパターン」の節で紹介した検索手法と同じく代償の多い手法ですが、検索を1回行えば良いだけなので、メリットの方が勝ります。交差テーブルに結果を保存すれば、以降の検索では同じキーワードを指定した検索を高速に行えるようになります。

次に、あるキーワードでの検索を簡単にするための**ストアドプロシージャ**を作成します[†8]。その語がすでに検索済みの場合、BugsKeywordsにそのキーワードが含まれるドキュメントのリストが格納されているので、クエリを高速に実行できます。そのキーワードを用いた検索が1回も実行されていなければ、労力を

†8 サンプルのストアドプロシージャはMySQLを使用しています。

かけてテキストエントリの集合を検索する必要があります。

Search/soln/inverted-index/search-proc.sql

```
CREATE PROCEDURE BugsSearch(keyword VARCHAR(40))
BEGIN
  SET @keyword = keyword;

①  PREPARE s1 FROM 'SELECT MAX(keyword_id) INTO @k FROM Keywords
     WHERE keyword = ?';
   EXECUTE s1 USING @keyword;
   DEALLOCATE PREPARE s1;

   IF (@k IS NULL) THEN
②    PREPARE s2 FROM 'INSERT INTO Keywords (keyword) VALUES (?)';
     EXECUTE s2 USING @keyword;
     DEALLOCATE PREPARE s2;
③    SELECT LAST_INSERT_ID() INTO @k;
④    PREPARE s3 FROM 'INSERT INTO BugsKeywords (bug_id, keyword_id)
       SELECT bug_id, ? FROM Bugs
       WHERE summary REGEXP CONCAT('' [[:<:]]'', ?, '' [[:>:]]'')
         OR description REGEXP CONCAT('' [[:<:]]'', ?, '' [[:>]]'')';
     EXECUTE s3 USING @k, @keyword, @keyword;
     DEALLOCATE PREPARE s3;
   END IF;

⑤  PREPARE s4 FROM 'SELECT b.* FROM Bugs b
     INNER JOIN BugsKeywords k USING (bug_id)
     WHERE k.keyword_id = ?';
   EXECUTE s4 USING @k;
   DEALLOCATE PREPARE s4;
END
```

①ユーザー指定のキーワードを検索します。Keywords.keyword_idから整数型の主キーが返されるか、語が1回も検索されていない場合はNULLが返されます。
②マッチしなかった語は、新しい語として挿入されます。
③Keywordsで生成された主キー値を検索します。
④新しいキーワードを含む行をBugsで検索して、交差テーブルに値を代入します。
⑤最後に、キーワードが既存の行にマッチしたか、新エントリとして挿入したかにかかわらず、keyword_idとマッチした行全体をBugsテーブルから取得します。

これで、ストアドプロシージャを呼び出し、任意のキーワードを渡せるようになりました。マッチするバ

グを検索して交差テーブルに新しいキーワードの値を代入しなければならない場合でも、過去の検索結果を利用できる場合でも、ストアドプロシージャはマッチしたバグの集合を返します。

Search/soln/inverted-index/search-proc.sql

```
CALL BugsSearch('crash');
```

ただしこの方法では、トリガーを定義して、新しいバグを挿入するたびに交差テーブルに値を代入する必要があります。バグ説明文の編集機能を提供する場合には、もう1つトリガーを記述して、更新されたテキストを再解析してBugsKeywordsテーブルから行を追加または削除しなければならないでしょう[†9]。

Search/soln/inverted-index/trigger.sql

```
CREATE TRIGGER Bugs_Insert AFTER INSERT ON Bugs
FOR EACH ROW
BEGIN
  INSERT INTO BugsKeywords (bug_id, keyword_id)
    SELECT NEW.bug_id, k.keyword_id FROM Keywords k
    WHERE NEW.description REGEXP CONCAT('[[:<:]]', k.keyword, '[[:>:]]')
        OR NEW.summary REGEXP CONCAT('[[:<:]]', k.keyword, '[[:>:]]');
END
```

このキーワードリストにはユーザーが検索を実行するたびに値が追加されていくので、ナレッジベースの記事にあるすべての語をキーワードリストに追加する必要はありません。また、検索されるキーワードが予期できる場合には、あらかじめ検索を実行しておくこともできます。検索を実行しておくことによって、ユーザーはキーワードの最初の検索実行者にならずに済みます。

私は、この章の冒頭で紹介したナレッジベースアプリケーションには、自作の転置インデックスを使いました。また、Keywordsテーブルには num_searches列を追加しました。この列には、ユーザーがキーワードを検索するたびに値を加算し、どのキーワードへの需要が最も高いかがわかるようにしたのです。

問題を解決するために、必ずしもSQLを使う必要はありません。

†9 サンプルのトリガーはMySQLを使用しています。

17章
スパゲッティクエリ

ある事柄を説明するために、必要以上に多くの実体を仮定するべきではない
——オッカムのウィリアム（イギリス・オッカム出身の神学者・哲学者）

電話で幹部と話をしている上司が、慌てた様子であなたを手招きしました。側に寄ると、上司は受話器を手で覆ってあなたにささやきました。

「いいか、いま幹部が予算会議を開いている。そして、うちの課が人数相当の仕事をしていることを示さないと、人員をカットすると言っているんだ。すぐに調べて欲しい。うちの課が取り扱っている製品の数、バグを修正した開発者の数、開発者1人当たりの平均バグ修正数、修正したバグの中で顧客から報告されたバグの数だ。今すぐ取りかかってくれ！」

あなたは机に飛び戻ると、SQLツールを立ち上げ、クエリを書き始めました。すべての答えを一度に得たいので、複雑なクエリを1つ書くことにしました。単純な繰り返し作業を行わない方が、すぐに結果を出せると考えたからです。

Spaghetti-Query/intro/report.sql

```
SELECT COUNT(bp.product_id) AS how_many_products,
  COUNT(dev.account_id) AS how_many_developers,
  COUNT(b.bug_id)/COUNT(dev.account_id) AS avg_bugs_per_developer,
  COUNT(cust.account_id) AS how_many_customers
FROM Bugs b INNER JOIN BugsProducts bp ON b.bug_id = bp.bug_id
INNER JOIN Accounts dev ON b.assigned_to = dev.account_id
INNER JOIN Accounts cust ON b.reported_by = cust.account_id
WHERE cust.email NOT LIKE '%@example.com'
GROUP BY bp.product_id;
```

しかし、表示された結果の数字は間違っているように見えます。製品数は実際の製品数をはるかに上回る数を返し、開発者1人当たりの平均バグ修正数も1.0という不自然な数字です。顧客数も違っていました。顧客数のところに出ていたのは、上司が必要としていた、顧客が報告したバグの修正数の方でした。なぜ、

ことごとくかけ離れた結果になってしまったのでしょうか。このクエリは思ったよりもはるかに複雑なようです。

上司が電話を切り、ため息をついて言いました。「もういい。幹部に数字を報告できなかった。スタッフを減らさなくてはならない」

17.1 目的：SQLクエリの数を減らす

SQLプログラマーが最も多く直面する問題の1つが、「どのようにして目の前の仕事を1つのクエリで実現するか」というものです。

この問題はあらゆる状況で発生します。プログラマーは、1つのSQLクエリですら、難しく、複雑で、処理に手間がかかるものだと教わります。であるならば、SQLクエリを2つ書くことは2倍の苦しみを伴うだろうと考えますし、3つ以上書くのは論外だと考えるでしょう。

プログラマーは、タスクの複雑性を減らすことはできませんが、解決策はシンプルにしたいと考えます。そこで、クエリを「優雅に」「効率的に」書くことを目標にします。そしてタスクを1つのクエリで解決することで、この目標を達成したと見なします。

17.2 アンチパターン：複雑な問題をワンステップで解決しようとする

SQLは非常に表現力に優れた言語であり、1つのクエリやステートメントで多くのことを実現できます。しかし、だからといってSQLは1つのクエリですべてのタスクを処理することを強制するものではありませんし、ときにはそれが良くないアイデアである場合もあります。おそらくあなたが他のプログラミング言語でコードを書くときには、1つのプログラムですべての仕事をこなすような習慣はないはずです。

17.2.1 意図に反した結果

1つのクエリですべてを処理しようとすると、しばしば**デカルト積**（Cartesian product）が生じてしまいます。デカルト積は、クエリで指定する2つのテーブルが関連（リレーションシップ）を制限する条件を持たないときに生まれます。この制限がないと、2つのテーブルを結合することによって、1つのテーブルの各行が、もう1つのテーブルの**すべて**の行とペアになってしまいます。各行のすべての組み合わせによって、結果は予期しているよりもはるかに多くの行数になってしまうのです。

例を見てみましょう。バグデータベースに、製品別の修正済みバグ数と未修正バグ数を問い合わせたいとします。多くのプログラマーは、以下のようなクエリでバグ数を取得しようと考えるでしょう。

Spaghetti-Query/anti/cartesian.sql

```
SELECT p.product_id,
  COUNT(f.bug_id) AS count_fixed,
  COUNT(o.bug_id) AS count_open
FROM BugsProducts p
INNER JOIN Bugs f ON p.bug_id = f.bug_id AND f.status = 'FIXED'
INNER JOIN BugsProducts p2 USING (product_id)
INNER JOIN Bugs o ON p2.bug_id = o.bug_id AND o.status = 'OPEN'
WHERE p.product_id = 1
GROUP BY p.product_id;
```

あなたはたまたま、その製品の修正済みのバグ数が11件、未修正のバグが7件であると知っていたとします。すると、このクエリの結果は頭を悩ませるものになります（表17-1）。

表17-1 おかしなクエリ結果

product_id	count_fixed	count_open
1	77	77

結果がこれほどまでに実際と異なってしまった原因は何なのでしょうか。77が11×7の答えなのは偶然ではありません。この例では、BugsProductsテーブルがBugsテーブルの2つの部分集合と結合され、結果としてこれらの2つの部分集合のデカルト積が生じました。11行の修正済み（FIXED）バグが、7行の未修正（OPEN）バグとすべてペアになっていたのです。

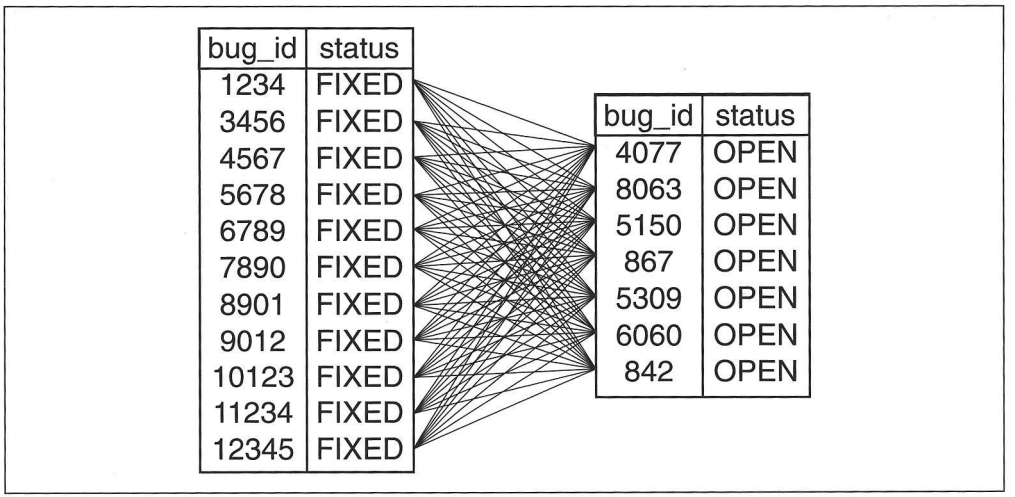


図17-1 修正済み（FIXED）バグと未修正（OPEN）バグ間のデカルト積

図17-1は、デカルト積を視覚的に示しています。修正済みと未修正のバグをつなぐそれぞれの線が、中間的な（グループ化が適用される前の）結果セットの行になっています。先ほどのクエリからGROUP BY句と集約関数を外すと、この中間的な結果セットを見ることができます。

Spaghetti-Query/anti/cartesian-no-group.sql

```
SELECT p.product_id, f.bug_id AS fixed, o.bug_id AS open
FROM BugsProducts p
INNER JOIN Bugs f ON p.bug_id = f.bug_id AND f.status = 'FIXED'
INNER JOIN BugsProducts p2 USING (product_id)
INNER JOIN Bugs o ON p2.bug_id = o.bug_id AND o.status = 'OPEN'
WHERE p.product_id = 1;
```

このクエリで示される唯一の関連（リレーションシップ）は、BugsProductsテーブルと、Bugsテーブルの各部分集合の間にあります。すべての修正済み（FIXED）バグとすべての未修正（OPEN）バグの組み合わせを制限する条件がないので、11 × 7の行が生じたのです。

このように、1つのクエリで2つのタスクを実現しようとした結果、意図しないデカルト積が頻繁に生まれます。1つのクエリでさらに多くのタスクを行おうとすると、さらに生じたデカルト積が掛けられた結果が導かれる可能性もあります。

17.2.2 さらなる弊害

1つのクエリで複数のタスクを行おうとすると、意図しない結果が導かれるだけでなく、クエリの記述や修正、デバッグが難しくなるという点も肝に銘じておくべきです。データベースアプリケーションでは、頻繁に機能追加が求められることを忘れてはいけません。上司から、さらに詳細なレポートの生成や、ユーザーインタフェースのフィールド追加を要求されることは十分に考えられます。複雑で巨大なSQLクエリを設計してしまうと、それらの修正に労力と時間がかかります。言うまでもなく、あなたにとってもプロジェクトにとっても、時間は極めて貴重なものであるはずです。

実行時のコストもあります。結合や相関サブクエリなどの多い手の込んだSQLクエリは、シンプルなクエリに比べて最適化処理や高速な実行が難しくなります。プログラマーは、実行するSQLクエリの数を減らせば、パフォーマンスが上がるという考えを抱いています。この考えは、SQLクエリの複雑性が同じであればその通りです。しかし、お化けのように複雑な1つのクエリは、パフォーマンスを低下させることがあります。それよりも、シンプルなクエリを複数実行する方が良い場合があるのです。

17.3 アンチパターンの見つけ方

プロジェクトのメンバーから以下のような発言を聞いたら、「**スパゲッティクエリ**」アンチパターンに注意すべきです。

「SUM関数や COUNT関数の結果があり得ないくらいに大きくなっているのはなぜ？」

意図しないデカルト積によって、結合された2つのデータセットが掛け合わされてしまったと考えられます。

「このお化けみたいに複雑なSQLクエリを書くのに、丸1日かかったよ！」

SQLは、化け物のように複雑なものにすべきではありません。1つのクエリを書くためにあまりに時間がかかるようなら、別のアプローチを考えてみるべきです。

「このレポート出力には、もう何も追加できない。このSQLクエリを書き直すのは手間がかかりすぎる」

クエリを書いたプログラマーは、たとえそのプロジェクトから離れても、そのコードのメンテナンスに責任を持たなくてはならない場合があります。あなたにも、そのような責任が生じる可能性は十分にあります。誰もメンテナンスできないほど複雑なSQLは書かないようにしましょう。

「このクエリに、もう1つDISTINCTを追加してみよう」

デカルト積で行が爆発的に増えたことを埋め合わせるために、プログラマーはクエリや集約関数の修飾子にDISTINCTキーワードを使って、行の重複を防ごうとします。DISTINCT修飾子によってクエリの

欠点は隠されますが、データベースはソートと重複排除のためだけに、内部で中間結果を生成しなくてはならなくなります。

実行時間が長くなりすぎる場合も、「スパゲッティクエリ」アンチパターンに陥っている可能性があります。もちろん他の原因によってパフォーマンスが低下したことも考えられますが、原因を調べる際には、1つのSQLであまりに多くのタスクをしようとしていないか、確認するように心がけましょう。

17.4 アンチパターンを用いてもよい場合

複雑なタスクを1つのクエリで実現しなければならない理由の1つは、単一のクエリをデータソースに結び付けてアプリケーションにデータを表示するような、プログラミングフレームワークやビジュアルコンポーネントライブラリを使っている場合です。単純なビジネスインテリジェンス（BI）ツールやレポートツールも、これに該当します。ただし、洗練されたBIツールには複数のデータソースの結果をマージできるものもあります。

データソースが単一のSQLクエリであることを前提とするコンポーネントやレポートツールはシンプルに使用できるかもしれませんが、すべてのデータをまとめて扱うためには、巨大で一枚岩なクエリ設計を行わなければなりません。このようなレポートアプリケーションを使う場合、結果セットを処理するために、自分でクエリを書く場合よりも複雑なSQLクエリを書かなければならない場合があります。

レポートの要件が1つのSQLクエリで実現するにはあまりにも複雑な場合は、レポートを複数作成する方が良いかもしれません。上司が複数のレポートを好まなければ、そのための処理がいかに複雑で時間を要するものであるかを説明して、理解を得るようにしましょう。

複数の結果を1つのソート順で表示させるために、1つのクエリから複雑な結果を得たい場合もあります。SQLクエリでは、ソート順を簡単に指定できます。複数のクエリの結果をソートするためには、アプリケーションコードを書くよりも、データベースでソートを行う方が効率的な場合が多いでしょう。

17.5 解決策：分割統治を行う

本章の冒頭で引用したオッカムのウィリアムの言葉は、**節約の原則**（the law of parsimony）としても知られています。

> まったく同じ予測をする2つの競合する理論があるときは、単純な方が優れている。
>
> ——節約の原則

この原則をSQLに当てはめると、まったく同じ結果セットを生む2つのクエリを選択できる場合は、単純なクエリを選ぶべきであると言えます。スパゲッティクエリの修正においても、この原則を念頭におくべきです。

17.5.1 ワンステップずつ

意図しないデカルト積が生じているテーブル間に論理的な結合条件が見つからない場合、単にそのような

条件がそもそも存在していないのかもしれません。デカルト積を避けるためには、スパゲッティクエリをいくつかのクエリに分割する必要があります。前述したシンプルな例では、必要なクエリはわずか2つです[†1]。

Spaghetti-Query/soln/split-query.sql

```
SELECT p.product_id, COUNT(f.bug_id) AS count_fixed
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON p.bug_id = f.bug_id AND f.status = 'FIXED'
WHERE p.product_id = 1
GROUP BY p.product_id;

SELECT p.product_id, COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs o ON p.bug_id = o.bug_id AND o.status = 'OPEN'
WHERE p.product_id = 1
GROUP BY p.product_id;
```

これらの2つのクエリの結果は、期待通り、11と7です。

クエリを複数に分割するのは「野暮な」解決策に思えるかもしれません。しかしクエリ分割は、開発、メンテナンス、パフォーマンスなどの面で様々なメリットをもたらしてくれるのです。

- この章の例で見てきたように、分割したクエリからは望まないデカルト積が生じません。このため、クエリが正確な結果をもたらしていることを簡単に確認できます。
- レポートに新たな要件が追加されたとき、すでに複雑なクエリをさらに複雑にするよりも、単純なクエリを新たに書く方がはるかに簡単です。
- 一般的に、SQLエンジンは複雑なクエリよりも単純なクエリの方がよりスムーズかつ確実に実行できます。クエリ分割によって処理が重複していると思えたとしても、それを補って余りあるメリットがあります。
- コードレビューやチームメンバーのトレーニングセッションでは、シンプルな複数のクエリを説明する方が、1つの複雑なクエリを説明するよりも簡単です。

17.5.2 UNIONを用いる

複数のクエリの結果は、UNIONによって1つの結果セットにまとめることができます。これは、どうしても1つのクエリを実行して1つの結果を得たい場合（結果をソートしなければならない場合など）に効果的です。

†1 監訳注：特定のステータスのバグ数がゼロ件の場合を考慮して、結合の種類が**外部結合**（LEFT OUTER JOIN）に変わっていることに注意してください。

Spaghetti-Query/soln/union.sql

```
(SELECT p.product_id, 'FIXED' AS status, COUNT(f.bug_id) AS bug_count
 FROM BugsProducts p
 LEFT OUTER JOIN Bugs f ON p.bug_id = f.bug_id AND f.status = 'FIXED'
 WHERE p.product_id = 1
 GROUP BY p.product_id)

UNION ALL

(SELECT p.product_id, 'OPEN' AS status, COUNT(o.bug_id) AS bug_count
 FROM BugsProducts p
 LEFT OUTER JOIN Bugs o ON p.bug_id = o.bug_id AND o.status = 'OPEN'
 WHERE p.product_id = 1
 GROUP BY p.product_id)

ORDER BY bug_count DESC;
```

このクエリの結果は、各サブクエリの結果を合わせたものになります。この例では、各サブクエリで1行ずつヒットし、全体として合計2行が結果として出力されます。サブクエリの結果を他のサブクエリの結果と区別するための列を追加することを忘れないようにしましょう。この例では、status列が区別のために使われます。

UNIONは、両方のサブクエリの列に互換性があるときにのみ使用できます。結果セットの途中から列の数や名前、データ型を変更することはできないため、すべての行において列が一貫性と正確性を保っていることを確認しましょう。bugcount_or_customerid_or_nullのような列エイリアスを定義しているのに気づいたら、おそらく互換性のないクエリ結果をまとめるためにUNIONを用いていると考えるべきです。

17.5.3 CASE式とSUM関数を組み合わせる[†2]

やや本文の趣旨と異なりますが、条件ごとの集約を1つのクエリでシンプルに行うために、CASE式とSUM関数を組み合わせる方法がよく使われます。

Spaghetti-Query/soln/union.sql

```
SELECT p.product_id,
  SUM(CASE b.status WHEN 'FIXED' THEN 1 ELSE 0 END) AS count_fixed,
  SUM(CASE b.status WHEN 'OPEN' THEN 1 ELSE 0 END) AS count_open
FROM BugsProducts p
INNER JOIN Bugs b USING (bug_id)
WHERE p.product_id = 1
GROUP BY p.product_id;
```

†2 監訳注：17.5.3項は監訳者による追記。

17.5.4 上司の問題を解決する

冒頭のエピソードのような、プロジェクトの統計値を出して欲しいという緊急のリクエストに応えるにはどうすればよいのでしょうか。上司からは、課が取り扱っている製品の数、バグを修正した開発者の数、開発者1人当たりの平均バグ修正数、修正したバグの中で顧客から報告されたバグの数を求められていました。

最善の解決策は、これらのタスクを分割して処理することです。

課が取り扱っている製品の数：

Spaghetti-Query/soln/count-products.sql

```
SELECT COUNT(*) AS how_many_products
FROM Products;
```

バグを修正した開発者の数：

Spaghetti-Query/soln/count-developers.sql

```
SELECT COUNT(DISTINCT assigned_to) AS how_many_developers
FROM Bugs
WHERE status = 'FIXED';
```

開発者1人当たりの平均バグ修正数：

Spaghetti-Query/soln/bugs-per-developer.sql

```
SELECT AVG(bugs_per_developer) AS average_bugs_per_developer
FROM (SELECT dev.account_id, COUNT(*) AS bugs_per_developer
      FROM Bugs b INNER JOIN Accounts dev
        ON b.assigned_to = dev.account_id
      WHERE b.status = 'FIXED'
      GROUP BY dev.account_id) t;
```

修正したバグの中で顧客から報告されたバグの数：

Spaghetti-Query/soln/bugs-by-customers.sql

```
SELECT COUNT(*) AS how_many_customer_bugs
FROM Bugs b INNER JOIN Accounts cust ON b.reported_by = cust.account_id
WHERE b.status = 'FIXED' AND cust.email NOT LIKE '%@example.com';
```

いま見てきたクエリのうちのいくつかは、それ自体で十分に手の込んだものであると言えます。これらをすべて1つのクエリで実現するのがいかに困難なものであるか、想像できるのではないでしょうか。

17.5.5 SQLを用いたSQLの自動的な記述

複雑なSQLクエリを分割すると、データの値によってわずかに異なる、似たようなクエリをいくつも作成することがあります。よく似たクエリを書くのは煩わしい仕事であるため、コード生成を行う良い機会となるでしょう。

「**コード生成**」は、コンパイルや実行が可能なコードをコードから出力する技法です。コード生成は、新しいコードを手で書くには非常に労力がかかるような場面で効果的です。この技法を使いこなすことによって、プログラマーは繰り返し同じようなコードを書かなくても済むようになります。

複数のUPDATEステートメント生成

コンサルティングを行っていたときの話です。別の部門のIT担当者から電話があり、SQLに関する緊急の問題の解決を依頼されました。

マネージャーのオフィスに行くと、担当者が憔悴しきった様子で問題の解決に取り組んでいました。挨拶もそこそこに、彼は嘆くように状況を説明し始めました。

「在庫マネジメントシステムが朝からオフラインのままなんです。すぐに問題を解決しなくてはいけません。力を貸してください」

彼はSQLの初心者ではありませんでした。それでも、大量の行の更新をするためのステートメントを1つ書くために、数時間も悪戦苦闘していたのです。

問題は、行の値を更新するためのUPDATEステートメントで、すべての行に対する一貫したSQL式を使えなかったことでした。設定対象の値が、行によって異なっていたのです。そのデータベースは、コンピューター室の各コンピューターの利用明細を記録するというものでした。担当者の彼が取り組んでいた仕事は、列last_usedの値を、各コンピューターが使用された最新の日付に設定することでした。

彼はこの複雑なタスクを、必死になって1つのSQLステートメントで解決しようとしていました。まさにスパゲッティクエリアンチパターンです。何時間もかけて、完璧なUPDATEステートメントを書くために苦しんでいたのです。

複雑な1つのSQLステートメントを書く代わりに、私はシンプルなSQLを複数生成するスクリプトを書きました。その結果、求めている結果を得ることができました。

Spaghetti-Query/soln/generate-update.sql

```
SELECT CONCAT('UPDATE Inventory '
  ' SET last_used = ''', MAX(u.usage_date), '''',
  ' WHERE inventory_id = ', u.inventory_id, ';') AS update_statement
FROM ComputerUsage u
GROUP BY u.inventory_id;
```

このクエリは、複数のUPDATEステートメントを出力します。各ステートメントはセミコロンで区切られ、SQLスクリプトとして実行できます（表17-2）。

表 17-2 生成される UPDATE 文

update_statement
UPDATE Inventory SET last_used = '2002-04-19' WHERE inventory_id = 1234;
UPDATE Inventory SET last_used = '2002-03-12' WHERE inventory_id = 2345;
UPDATE Inventory SET last_used = '2002-04-30' WHERE inventory_id = 3456;
UPDATE Inventory SET last_used = '2002-04-04' WHERE inventory_id = 4567;
..

このテクニックによって、その担当者が何時間も頭を悩ませていた問題を、数分のうちに解決できたのです。

複数の SQL クエリやステートメントを実行するのは、効率の面からは最善の方法ではない場合もあるかもしれません。しかし状況に応じて、効率とタスク実現のどちらを優先するかを的確に判断することが大切なのです。

SQL では、1 行のコードで複雑な問題を解決できると思える場合があります。しかし、状況に応じてクエリを分割することも検討するようにしましょう。

18章
インプリシットカラム（暗黙の列）

自分が何を考えているかは、実際に言葉にしてみるまでわからない
—— E・M・フォースター（イギリスの小説家）

あるPHPプログラマーから助けを求められました。図書館のデータベースで、シンプルなSQLクエリが返す結果に困惑しているというのです。

Implicit-Columns/intro/join-wildcard.sql

```
SELECT * FROM Books b INNER JOIN Authors a ON b.author_id = a.author_id;
```

このクエリは、すべての書籍の書名にNULLを返していました。さらに奇妙なのは、Authorsテーブルとの結合を行わないクエリを実行すると、結果には期待通り正しい書名が含まれることでした。

私は、彼と共に問題の原因を突き止めました。原因は、彼が利用していたPHPのDB拡張ライブラリが、このSQLクエリの実行結果の各行を連想配列として返すことにありました。例えば、Books.isbn列には、PHPからは$row["isbn"]でアクセスできます。このデータベースのBooksテーブルとAuthorsテーブルには、どちらにも、title列があります（前者は本のタイトル、後者はDr.やRev.など、著者の肩書を意味するタイトル）。しかし、結果セットのある一行に対して、連想配列の要素$row["title"]には、1つの値しか格納できません。今回の場合には、Authors.titleがこの配列要素に格納されていました。データベース内のほとんどの著者は、肩書き（title）を持たないため、結果として$row["title"]がNULLになってしまったのです。Authorsとの結合を行わなければ列名が衝突しないので、期待通りに書名が連想配列に格納されていたというわけです。

私は彼に、列に**エイリアス（別名）**を宣言してどちらかのtitle列に別の名前を与え、それぞれが別の名前で連想配列に入るようにすると良いとアドバイスしました。

Implicit-Columns/intro/join-alias.sql

```
SELECT b.title, a.title AS salutation
FROM Books b INNER JOIN Authors a ON b.author_id = a.author_id;
```

彼の2番目の質問は、「1つの列にはエイリアスを付けて、かつ他の列もすべて取得するにはどうすればいいですか？」でした。彼はワイルドカード（SELECT *）を使いつつ、ワイルドカードの対象となる列の1つにエイリアスを適用する方法が知りたかったのです。

18.1 目的：タイプ数を減らす

ソフトウェア開発者は概して、キーをたくさん打つことを好みません。「コンピューターのキーを打つ」ことを職業に選んだはずの開発者にとって、それは作家O. ヘンリーの短篇の結末のように皮肉なものだと言えるかもしれません。

プログラマーが入力を手間だと感じる例の1つが、SQLクエリですべての列名を指定することです。

Implicit-Columns/obj/select-explicit.sql

```
SELECT bug_id, date_reported, summary, description, resolution,
  reported_by, assigned_to, verified_by, status, priority, hours
FROM Bugs;
```

このサンプルを見れば、ソフトウェア開発者がSQLの**ワイルドカード**機能を頻繁に使うのも驚くに値しません。ワイルドカード記号のアスタリスク（*）はすべての列を意味するため、列名のリストは明示的ではなく暗黙的に指定されます。ワイルドカードを使うとクエリを簡潔に書くことができます。

Implicit-Columns/obj/select-implicit.sql

```
SELECT * FROM Bugs;
```

また、INSERT文を書く際にも、デフォルト順の恩恵を受けるのがスマートな方法のように見えます。VALUESに並べた値は、すべての列にテーブルで定義された順番で適用されます[†1]。

Implicit-Columns/obj/insert-explicit.sql

```
INSERT INTO Accounts (account_id, account_name, first_name, last_name, email,
  password_hash, portrait_image, hourly_rate) VALUES (DEFAULT,
  'bkarwin', 'Bill', 'Karwin', 'bill@example.com', SHA2('xyzzy', 256), NULL, 49.95);
```

列を明示的に指定せずに書くことで、ステートメントは短くなります。

Implicit-Columns/obj/insert-implicit.sql

```
INSERT INTO Accounts VALUES (DEFAULT,
  'bkarwin', 'Bill', 'Karwin', 'bill@example.com', SHA2('xyzzy', 256), NULL, 49.95);
```

18.2 アンチパターン：ショートカットの罠に陥る

ワイルドカードや暗黙的な列の指定によって、タイプ数は減らせます。しかし、この方法には様々な弊害

†1 監訳注：サンプルコードのSHA2関数はMySQLのハッシュ作成関数です。詳しくは19章「リーダブルパスワード（読み取り可能パスワード）」を参照してください。

があるのです。

18.2.1 リファクタリングにおける問題

例えばスケジュール管理のために、Bugsテーブルに date_due列を加える必要があるとします。

Implicit-Columns/anti/add-column.sql

```
ALTER TABLE Bugs ADD COLUMN date_due DATE;
```

すると、INSERTステートメントがエラーを返すようになります。テーブルには12の列があるのに、クエリでは11列しか値を指定していないからです。

Implicit-Columns/anti/insert-mismatched.sql

```
INSERT INTO Bugs VALUES (DEFAULT, CURRENT_DATE,
    '新規バグ', 'テスト[T987]が失敗します...',
    NULL, 123, NULL, NULL, DEFAULT, 'MEDIUM', NULL);

-- SQLSTATE 21S01: Column count doesn't match value count at row 1
```

暗黙的な列を使ったINSERTステートメントでは、テーブルに列が定義されている順番と同じ順番で、すべての列に値を与えなければなりません。列定義に変更があった場合には、ステートメントがエラーを返すか、さらに悪い場合には、誤った列に値が格納されてしまうことになります。

また、列名を知らずに SELECT * クエリを実行する場合、列は定義順に基づいて参照されます。

Implicit-Columns/anti/ordinal.php

```
<?php
$stmt = $pdo->query("SELECT * FROM Bugs WHERE bug_id = 1234");
$row = $stmt->fetch();
$hours = $row[10];
```

しかし、チームの誰かが、あなたの知らないうちに列を削除したらどうなるでしょうか。

Implicit-Columns/anti/drop-column.sql

```
ALTER TABLE Bugs DROP COLUMN verified_by;
```

hours列は、添字 10の位置には存在しなくなったため、アプリケーションは誤って別の列の値を使用するようになってしまいました[†2]。列の追加、削除、名前変更などを行うと、クエリ結果に生じた変化をコードがうまく扱えなくなる場合があります。ワイルドカードを使用している場合は、クエリがいくつの列を返すのか予測できないからです。

†2 監訳注：PHPのDB拡張（PDO）では、列番号は0から始まります。

このようなエラーはコード全体に伝播します。アプリケーションの出力に問題があることに気づいてから、原因を遡って追求するのは簡単ではありません。

18.2.2 隠れた代償

クエリでワイルドカードを使うのは便利ですが、パフォーマンスとスケーラビリティに悪い影響を及ぼす場合があります。クエリが多くの列をフェッチするようになるため、多くのデータがアプリケーションとデータベースサーバーの間を行き来しなくてはなりません。

本番稼働中のアプリケーションでは、多くのクエリを同時に実行することがあるはずです。これらは同じネットワーク帯域幅を利用します。ギガビットネットワークでさえも、数千行を返すクエリを100個も同時に実行すれば、帯域幅がいっぱいになってしまうことがあります。

オブジェクトリレーショナルマッピング（ORM）フレームワークでは、データベースの行を表すオブジェクトのフィールドに値を代入するために、ワイルドカードがデフォルトで使われていることがあります。ORMがこの振舞いを上書きする手段を提供している場合でも、ほとんどのプログラマーはこのデフォルトを変えようとはしません[†3]。

18.2.3 求めなければ得られない

SQLのワイルドカードについて、私がプログラマーから最も多く尋ねられる質問の1つは「指定した一部の列を除いたすべての列を求めるショートカットはありますか」というものです。たいていは、クエリの対象外であるデータ量の多いTEXT列は除きたいものの、ワイルドカードは使いたいという理由です。

この質問への答えは、ノーです。SQLでは「不要な列以外のすべての列」を意味する構文はサポートされていません。ワイルドカードで暗黙的にすべての列を対象にするか、明示的に列を指定するかのどちらかしかないのです。

18.3 アンチパターンの見つけ方

プロジェクトが以下のような状況になったときには、暗黙的な列指定を不適切に使用したために問題が発生する「**インプリシットカラム（暗黙の列）**」アンチパターンに陥っている可能性があります。

「アプリケーションに障害が発生した。データベースの結果セットを古い列名で列を参照していたことが原因だ。コードをすべて修正したつもりだったが、修正漏れがあるかもしれない」

列の追加、削除、名前変更、順番変更などによってテーブル定義を変更したものの、テーブルを参照するアプリケーションコードを修正していなかったことが原因です。参照しているアプリケーションコードをすべて修正するのは、たいへんな手間がかかります。

「何日もかけて、ようやくネットワークのボトルネックを突き止めた。原因は、データベースサーバーへのトラフィックが多いことだった。統計値によれば、クエリは平均で2MB以上のデータをフェッチしているが、実際に表示しているのはその10分の1以下だった」

不要なデータをフェッチしすぎています。

†3 監訳注：近年では、ワイルドカードを使うような素朴な実装のORMは少なくなってきました。

18.4 アンチパターンを用いてもよい場合

ワイルドカードの使用は、アドホックなSQLを素早く書きたい場合には妥当であると言えます。例えば、ある解決法を試してみたいときや、現行システムのデータを診断したいときなどです。1回しか使用しないクエリでは、保守性の低さはそれほど問題にはなりません。

本書のサンプルでもワイルドカードを使用していますが、それは紙面を節約したり、重要な部分を強調したりすることが目的です。私は実際のアプリケーションコードではめったにワイルドカードを使いません。

列の追加、削除、名前変更、順番変更を行ってもできるだけ修正せずに済むクエリを書きたい場合には、ワイルドカードの使用が最適だと考えるかもしれません。ただし、前述したリスクに対する対策を事前にしっかりと計画しておく必要があります。

また、結合を行うクエリでは、テーブルごとにワイルドカードを使えます。この場合、ワイルドカードの前にテーブル名やエイリアスを指定します。これによって、あるテーブルの部分的な列を指定すると同時に、ワイルドカードによってもう一方のテーブルからすべての列をフェッチできます。以下に例を示します。

Implicit-Columns/legit/wildcard-one-table.sql

```
SELECT b.*, a.first_name, a.email
FROM Bugs b INNER JOIN Accounts a
  ON b.reported_by = a.account_id;
```

多くの列名を入力するのには時間がかかります。実行時の効率よりも開発時の効率を重視するプログラマーがいます。クエリを短くすることによる可読性の向上を大切にするプログラマーもいます。ワイルドカードを使うことで、たしかにキー入力は減り、クエリも短くなります。開発時の効率、クエリの短さと可読性の方が重要であるなら、ワイルドカードを使いましょう。

SQLクエリが長くなることで、アプリケーションとデータベースの間のネットワークトラフィックが増えてしまうと言うプログラマーもいます。理論上、クエリが長ければトラフィックが増える可能性もあります。しかし一般的には、クエリが多くの行を返すことによって増加するデータの方が、はるかに多くの帯域幅を使用するものです。例外的なケースもあるので適切に判断することが大切になりますが、クエリ自体の長さは概して大きな問題ではありません。

18.5 解決策：列名を明示的に指定する

ワイルドカードや暗黙的な列指定を使わずに、**必要な列名は明示的に指定する**ようにしましょう。

Implicit-Columns/soln/select-explicit.sql

```
SELECT bug_id, date_reported, summary, description, resolution,
  reported_by, assigned_to, verified_by, status, priority, hours
FROM Bugs;
```

Implicit-Columns/soln/insert-explicit.sql

```
INSERT INTO Accounts (account_name, first_name, last_name, email,
```

```
  password_hash, portrait_image, hourly_rate)
VALUES ('bkarwin', 'Bill', 'Karwin', 'bill@example.com',
  SHA2('xyzzy', 256), NULL, 49.95);
```

列名をすべて入力するのは手間がかかると思うかもしれませんが、それだけの価値はあるのです。

18.5.1 誤りの防止

本書の4章で紹介した「**ポカヨケ**」を覚えているでしょうか[†4]。クエリの選択リストで列を明示的に指定することで、本章で見てきたエラーや混乱が発生しにくくなります。

- テーブル定義の列の順番が変更された場合でも、クエリ結果の列の位置は変わりません。
- テーブルに列が加えられた場合でも、クエリ結果に影響はありません。
- テーブルから列が削除された場合、クエリはエラーを返します。ただし、これは良いエラーです。修正すべきコードをすぐに特定できるからです。原因究明のためにコード全体を探し回る必要がなくなります。

INSERTステートメントでも列を明示的に指定すると、同じようなメリットを得られます。明示的に指定した列順はテーブル定義での列順よりも優先されるため、値は必ず意図した列に割り当てられます。新たに追加された列がステートメントで指定されていない場合は、値がデフォルト値またはNULLになります[†5]。削除された列が指定されているステートメントはエラーを返しますが、これも解決が容易な問題です。

これは「できるだけ早い段階で失敗すべし」という原則に従っている良い例だと言えるでしょう。

18.5.2 それは多分、必要ない（YAGNI：You Ain't Gonna Need It）

ソフトウェアのスケーラビリティとスループットを考慮すると、ネットワーク帯域幅はなるべく減らしたいと考えるのも当然です。SQLクエリの帯域幅は、開発期間やテスト期間では無害なものに思えます。しかし、本番環境で1秒に何千ものSQLクエリが実行されるようになると、大きな影響が生じます。

SQLでワイルドカードを使用しないようにすると、使うかどうか分からない列をなるべく作らないようにしようという意識が高まります。列が必要最小限であることは、タイプ量が減ることを意味するからです。結果として、帯域幅も効率的に使用できるというわけです。

Implicit-Columns/soln/yagni.sql

```
SELECT date_reported, summary, description, resolution, status, priority
FROM Bugs;
```

†4 ポカヨケは、日本で開発された製造現場でのミス防止のシステムのことです。

†5 監訳注：新たに追加された列にNOT NULL制約が定義されていて、かつ列にデフォルト値が定義されていない場合には、INSERTステートメント実行時にエラーが発生します。

18.5.3 ワイルドカードを使えない局面はいずれ訪れる

自動販売機で袋入りのキャンディを1袋買ったとします。袋があるからこそ、すべてのキャンディを机に持ち帰ることができます。いったん袋を開ければ、キャンディは1つずつ扱わなくてはなりません。袋を開けた弾みでキャンディが机の上に転がり、床に落ちてしまうかもしれません（アリが集まってくることもあるでしょう）。それでも、袋を開けなければキャンディは食べられません。

SQLクエリを書いていて、列に式を適用したり、列のエイリアスを宣言したり、効率のために列を除外するような必要が出てきた場合には、ワイルドカードという袋を開けなければなりません。列をひとまとまりのパッケージとして扱える便利さは失いますが、中身に個別にアクセスできるようになります。

列エイリアスや関数を利用したり、選択リストから特定の列を除外する場合など、クエリで各列を個別に扱わなければならないときが、いつか必ず訪れます。初めからワイルドカードを使っていなければ、後でクエリ変更が楽になるのです。

> 必要な列だけを指定するようにしましょう。

Ⅳ部
アプリケーション開発のアンチパターン

19章
リーダブルパスワード（読み取り可能パスワード）

敵はシステムを知っている
——クロード・シャノン（アメリカの電気工学者・数学者）

あなたがサポートを担当しているアプリケーションのユーザーから電話がかかってきました。うまくログインできないというのです。

「営業部門のPat Johnsonと言います。パスワードを忘れてしまったようです。調べてもらえませんか？」

男性は少しきまり悪そうに言いました。かなり急いでいるようでもあります。

「すみませんが、それはできないのです」あなたは答えます。「アカウントをリセットすることはできます。リセットすると、アカウントに登録したアドレスに電子メールが自動的に送信されるので、そのメールに記載された指示に従って新パスワードを設定してください」

男性は苛立った様子で、強く主張しました。「いますぐパスワードが必要なんです。以前務めていた会社では、サポートスタッフはパスワードを忘れたらすぐに調べてくれました。なぜ口頭で教えてもらえないのでしょうか？ どうしても駄目というのなら、話をするのであなたの上司につないでください」

ユーザーといざこざを起こしたくないあなたは、しかたなくSQLクエリを実行してPat Johnsonのアカウントの平文パスワードを取得し、電話で伝えました。

電話を切ったあなたは、隣にいた同僚にぼやきました。「まいったよ。営業部のPat Johnsonという人から電話があって、パスワードは口頭で伝えられないと言ったら、上司につなげと言うんだ。しょうがないから教えたよ。後で彼が文句を言わないといいんだけど」

同僚は驚いた様子で言いました。「彼？　営業部門にはPat Johnsonという人がいるけど、その人は女性だよ。騙されたんじゃない？」

19.1　目的：パスワードのリカバリーとリセットを行う

パスワードを使用するアプリケーションには、ユーザーがパスワードを忘れることがつきものです。こうしたユーザーのために、最近のアプリケーションの多くは、電子メールを通じてパスワードの再通知やリセットを行うことができます。再通知やリセットは、アプリケーションのユーザープロファイルに関連付けられた電子メールアドレスを、パスワードを忘れたユーザーが使用できることを前提としています。

19.2 アンチパターン：パスワードを平文で格納する

この種のパスワードリカバリー手法でよくある間違いは、**平文**のパスワードが含まれた電子メールをユーザーがリクエストできるようにしてしまうことです。これはデータベース設計における**重大なセキュリティ欠陥**であり、権限のない人に特権的アクセスを与えるセキュリティリスクを生じさせます。

以降の節では、これらのリスクを見ていきます。なお、この章のサンプルでは、バグ管理データベースのAccountsテーブルの各行に、ユーザーのアカウントが格納されているものとします。

19.2.1 パスワードの格納

パスワードは、よく文字列型の列としてAccountsテーブルに格納されます。

Passwords/anti/create-table.sql

```
CREATE TABLE Accounts (
  account_id    SERIAL PRIMARY KEY,
  account_name  VARCHAR(20) NOT NULL,
  email         VARCHAR(100) NOT NULL,
  password      VARCHAR(30) NOT NULL
);
```

新規アカウント作成のために行を挿入する際に、パスワードも文字列リテラルで記述します。

Passwords/anti/insert-plaintext.sql

```
INSERT INTO Accounts (account_id, account_name, email, password)
  VALUES (123, 'billkarwin', 'bill@example.com', 'xyzzy');
```

パスワードを平文で格納するのは安全ではありませんし、ネットワークの上で平文パスワードをやりとりするのは、なおさら危険です。パスワードの挿入に使うSQL文を攻撃者に読み取られてしまった場合、パスワードが簡単に判明してしまいます。これはパスワード変更のためのSQL文や、データベースに格納されたパスワードとユーザーの入力内容が一致するかどうかを確認するSQL文でも同様です。攻撃者には、パスワードを盗むチャンスがいくつもあります。以下に例を挙げましょう。

- アプリケーションクライアントからデータベースサーバーに送信されたSQL文のネットワークパケットを傍受することは、見かけほど難しいことではありません。傍受を可能にするWiresharkなどのフリーソフトウェアもあります[†1]。
- データベースサーバー上のSQLクエリログを探す方法もあります。データベースサーバーに侵入した攻撃者によって、データベースが実行したSQL文の記録を含むログファイルにアクセスされてしまう可能性があります。

†1 Wireshark（以前の名称はEthereal）はhttp://www.wireshark.org/からダウンロードできます。

- データベースのバックアップファイル、またはバックアップメディアからもデータを読み取れます。バックアップメディアが盗まれたり、バックアップメディアのリサイクルや破棄の前にデータを完全に消去していないと、パスワードが盗まれてしまうことがあります。

19.2.2 パスワードの認証

ユーザーがログインを行うとき、アプリケーションはユーザーの入力内容を、データベースに格納したパスワード文字列と比較します。パスワードが平文として格納されている設計では、この比較も平文として行われます。例えば以下のクエリは、ユーザー入力内容とデータベースのパスワードが一致しているかどうかによって、0（不一致）または1（一致）を返します。

Passwords/anti/auth-plaintext.sql

```
SELECT CASE WHEN password = 'opensesame' THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

この例では、ユーザーが入力したパスワード「opensesame」は正しくなかったため、0が返されます。

パスワードの格納と同様、ユーザーの入力した文字列を平文としてSQLクエリに挿入するのは、攻撃者にパスワードを晒してしまうことを意味します。

2つの条件をひとまとめにしない

認証クエリの多くでは、WHERE句に account_id列と password列の2つの条件が指定されています。

Passwords/anti/auth-lumping.sql

```
SELECT * FROM Accounts
WHERE account_name = 'bill' AND password = 'opensesame';
```

アカウントが存在しないか、ユーザーが間違ったパスワードを入力した場合、このクエリは空の結果セットを返します。アプリケーションは、認証が失敗した原因がアカウントなのかパスワードなのかを区別できません。2つの条件を区別するクエリを使えば、失敗の原因に適切に対処しやすくなります[†2]。

例えば、ログインが複数回連続で失敗したときは、第三者による侵入の可能性があるため、アカウントを一時的にロックしたい場合があります。しかし、ログインが失敗した原因がアカウント名なのかパスワードなのかを区別できなければ、このパターンを検出できません。

19.2.3 パスワードを電子メールで送信する

パスワードがデータベースに平文で格納されている設計では、アプリケーションはパスワードを簡単に取

†2 監訳注：ただし、ログインエラーメッセージとして「ユーザーが存在しない」のか「ユーザーは存在するが、パスワードが間違っている」のかは表示すべきではありません。攻撃者にヒントを与えてしまうからです。参考：http://www.ipa.go.jp/security/awareness/vendor/programmingv1/b09_03.html

得できます。

Passwords/anti/select-plaintext.sql

```
SELECT account_name, email, password
FROM Accounts
WHERE account_id = 123;
```

アプリケーションは、ユーザーのリクエストに応じて、パスワードをユーザーのメールアドレス宛に送信できます。ほとんどの人は、以下のようなパスワード再通知の電子メールを見たことがあるはずです。

パスワードを再通知する電子メールの例

```
From: daemon
To: bill@example.com
Subject: パスワードのリクエスト

アカウント「bill」のパスワードの再通知リクエストに回答します。
パスワードは「xyzzy」です。

アカウントにログインするには、以下のリンクをクリックしてください。
http://www.example.com/login
```

平文のパスワードを電子メールで送信するのは、**非常に深刻なセキュリティリスク**です。攻撃者は様々な方法で、電子メールの傍受、記録、保存を行えます。メールを取得する際にセキュアなプロトコルを使用したり、メールの送受信サーバーが信頼できるシステム管理者によって管理されていても、十分な対策にはなりません。電子メールはインターネットの様々なサーバーを経由するため、途中で傍受される可能性があります。電子メール向けのセキュアなプロトコルも、誰もが使用しているとは限りませんし、開発者が完全にコントロールできるものでもありません。

19.3 アンチパターンの見つけ方

パスワードのリカバリーを行ってユーザーへ送信できるアプリケーションは、パスワードを平文または復元可能な暗号化によって格納しています。それこそが「**リーダブルパスワード（読み取り可能パスワード）**」アンチパターンです。アプリケーションが正当な目的でパスワードを読み取れるということは、攻撃者が不当にパスワードを読み取れることも意味するのです。

19.4 アンチパターンを用いてもよい場合

アプリケーションは、外部のサービスにアクセスするためのパスワードを扱う場合があります。つまり、あなたのアプリケーションがクライアント側になる場合です。この場合は当然、アプリケーションは解読可能な形式でパスワードを格納しなければなりません。ただし、なるべく平文ではなく、アプリケーションで解読できる暗号化を用いるべきです。

本人識別（Identification）と**認証**（Authentication）を区別する考えも大切です。本人識別とは、自分が

誰であるかを申告することです。ユーザーは誰の名前であれ、好きに申告できます。これに対して認証とは、自らが名乗った人物であることを証明することです。パスワードは、認証を行う最も一般的な方法です。

スキルのある、しつこい攻撃者からの攻撃を防げないときは、認証のメカニズムではなく、本人識別のメカニズムを構築しているに過ぎない場合があります。しかし、それが常に重大な問題になるとは限りません。

すべてのアプリケーションが攻撃の危険にさらされているわけではありません。同じく、すべてのアプリケーションが重大な機密情報を含んでいるわけでもありません。例えば、イントラネット上のアプリケーションには、不正な利用をすることはまず考えられない少人数のメンバーのみで使用しているものがあります。この場合、アプリケーションを管理するうえでは、単純なログインのメカニズムのみで十分な場合もあります。こうした内部的なメンバーのみが使用する環境では、堅牢な認証システムを構築するために作業を増やすのが妥当でない場合もあります。

ただし、注意は必要です。アプリケーションは、元々の環境や役割を越えて成長する傾向があります。小さなイントラネットアプリケーションを社内のファイヤーウォールの外でも使えるようにする前に、優秀なセキュリティ専門家による評価を行うべきです。

ソフトウェア開発における倫理

パスワードを扱うアプリケーションの開発を手がけていて、ユーザーによるパスワードリカバリー機能の設計を求められた場合は、丁重に断りを入れ、プロジェクトの意志決定者にそのリスクを伝えて、別の安全な方法でパスワードをリカバリーできる代替案を提示しましょう。

電気技師が危険な配線設計がないかを確認し、見つけた場合はそれを修正するのと同じように、セキュリティの問題を常に意識し、安全なソフトウェアの開発につとめることは、**ソフトウェアエンジニアの責任**です。

このテーマについての良書には『19 Deadly Sins of Software Security』[HLV05]があります。また、Open Web Application Security Project †3 もよい情報源になるでしょう。

19.5 解決策：ソルトを付けてパスワードハッシュを格納する

このアンチパターンの大きな問題点は、パスワードのオリジナル形式が解読可能なかたちで格納されていることです。しかし、実は解読可能でなくとも、ユーザーの入力内容を使用したパスワード認証は行えるのです。この節では、このようなセキュアなパスワードの格納手法をSQLデータベースに実装する方法について説明します。

19.5.1 ハッシュ関数を理解する

パスワードを、一方向性の**暗号学的ハッシュ関数**（cryptographic hash function）を用いて暗号化しま

†3 http://owasp.org

す。この関数は、入力文字列を、入力内容が識別不可能な**ハッシュ**と呼ばれる文字列に変換します。ハッシュ関数で返されるハッシュ値は固定長の文字列であるため、ハッシュからオリジナルの文字列の長さを特定することもできません。例えば、SHA-256アルゴリズムは、先ほどのサンプルで用いたパスワードxyzzyを、256ビットの文字列に変換します。これは通常、16進数の文字列64文字で表現されます。

```
SHA2('xyzzy', 256) = '184858a00fd7971f810848266ebcecee5e8b69972c5ffaed622f5ee078671aed'
```

ハッシュには、**不可逆である**という特性もあります。ハッシュアルゴリズムは、入力情報を「失う」ように設計されているため、ハッシュ値から入力文字列を復元することはできません。優れたハッシュアルゴリズムであるほど、ハッシュ値を解読するには、単純なトライアンドエラーよりも多くの労力が必要になります。

従来一般的だったハッシュアルゴリズムはSHA-1です。しかし、最近の研究によって、この160ビットのハッシュアルゴリズムの暗号強度が不十分であることが証明されました。SHA-1のハッシュ値から入力を逆算する技法があり、時間がかかるものの、単純なトライアンドエラーよりも短い時間で入力内容を解読できるのです。アメリカ国立標準技術研究所（NIST）は、2010年以降、安全なハッシュアルゴリズムとして認定されたリストからSHA-1を段階的に削除する計画を発表しました。新たに標準の暗号として認められたものには、SHA-224、SHA-256、SHA-384、SHA-512などがあります[†4]。NIST標準に従う必要があるか否かにかかわらず、パスワードの格納には、少なくともSHA-256を使うことが望ましいと言えます。

MD5も広く普及しているハッシュ関数です。MD5は128ビットのハッシュ文字列を作り出します。しかしMD5にも、暗号学的な弱点があることが判明しているため、パスワードの暗号化にMD5を用いるべきではありません。弱点が明らかになったアルゴリズムにも依然として使い道はありますが、パスワードのような機密情報に使うべきではありません。

19.5.2 SQLでのハッシュの使用

以下に示す例は、Accountsテーブルを再定義したものです。SHA-256を用いてパスワードをハッシュ化すると、常に64文字になります。このため、列を固定長のCHAR列として定義します。

Passwords/soln/create-table.sql

```
CREATE TABLE Accounts (
  account_id     SERIAL PRIMARY KEY,
  account_name   VARCHAR(20),
  email          VARCHAR(100) NOT NULL,
  password_hash  CHAR(64) NOT NULL
);
```

ハッシュ関数はSQL標準では定義されていないので、データベース製品ごとに異なるハッシュ拡張を使わなければなりません。例えば、SSLサポートを有効にしたMySQLでは、SHA2関数が使えます。SHA2

†4 http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html

関数の第2引数にはビット長を渡します。

Passwords/soln/insert-hash.sql

```
INSERT INTO Accounts (account_id, account_name, email, password_hash)
  VALUES (123, 'billkarwin', 'bill@example.com', SHA2('xyzzy', 256));
```

ユーザー入力に同じハッシュ関数を適用し、その結果をデータベースに格納されたハッシュ値と比較することで、パスワードの妥当性を確認できます。

Passwords/soln/auth-hash.sql

```
SELECT CASE WHEN password_hash = SHA2('xyzzy', 256) THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

また、パスワードハッシュの値をハッシュ関数が返せない文字列に変更することで、アカウントを簡単にロックできます。例えば、文字列 noaccess は 16 進数ではない値を含んでいます。

19.5.3 ハッシュにソルトを加える

平文パスワードではなくハッシュを格納していても、攻撃者にデータベースへのアクセスを許してしまえば（例えば、ゴミ箱に捨てたバックアップ用の CD-ROM を拾われてしまった場合など）、攻撃者はトライアンドエラーでパスワードを解読しようとするでしょう。解読には長時間を要するかもしれませんが、攻撃者は良くあるパスワードとそのハッシュ値のリストを格納した自前のデータベースを準備しておくこともできます。辞書に載っているようなありふれた言葉がパスワードに使われていれば、攻撃者のデータベースで同じハッシュを見つけ出すことは難しくありません。このような解読手法は、SQL を使って行うこともできます。

Passwords/soln/dictionary-attack.sql

```
CREATE TABLE DictionaryHashes (
  password       VARCHAR(100),
  password_hash  CHAR(64)
);

SELECT a.account_name, h.password
FROM Accounts AS a INNER JOIN DictionaryHashes AS h
  ON a.password_hash = h.password_hash;
```

この種の**辞書攻撃**（dictionary attack）を防ぐ方法の1つは、暗号化前のパスワードへの「**ソルト**」の付加です。ソルトとは、ハッシュ関数に渡す前にパスワードに連結する無意味な文字列のことです。ユーザー

が辞書にある「password」のような語をパスワードに選んだとしても、ソルトを付加されたパスワードから生成されたハッシュは、攻撃者のハッシュデータベースのハッシュとは一致しにくくなります。例えば、パスワードに使う文字列が「password」である場合、これをそのままハッシュ化した場合と、ソルトを付けてハッシュ化した場合とではハッシュ値が異なります[†5]。

```
SHA2('password', 256)
  = '5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8'

SHA2('password' || 'G0y6cf3$.ydLVkx4I/50', 256)
  = '9cb669bbba0bfd55189f7b58c1d85014ec4438e815e2993847a289bb41c46de8'
```

それぞれのパスワードに異なるソルト値を付加することで、攻撃者はハッシュの辞書テーブルをパスワードごとに新たに生成しなければならなくなります。パスワードの解読にはトライアンドエラーと同じだけの時間がかかるため、攻撃者をふりだしに戻すことができます[†6]。

Passwords/soln/salt.sql

```
CREATE TABLE Accounts (
  account_id     SERIAL PRIMARY KEY,
  account_name   VARCHAR(20),
  email          VARCHAR(100) NOT NULL,
  password_hash  CHAR(64) NOT NULL,
  salt           BINARY(20) NOT NULL
);

INSERT INTO Accounts (account_id, account_name, email,
    password_hash, salt)
  VALUES (123, 'billkarwin', 'bill@example.com',
    SHA2('xyzzy' || 'G0y6cf3$.ydLVkx4I/50', 256), 'G0y6cf3$.ydLVkx4I/50');

SELECT (password_hash = SHA2('xyzzy' || salt, 256)) AS password_matches
FROM Accounts
WHERE account_id = 123;
```

レインボーテーブル対策を行うには、ソルトとパスワードをあわせた長さが最低でも 20 文字は必要です。また各パスワードごとに、異なるソルトを付加するべきです。また、前述のサンプルではソルトの文字列に印刷可能な値が含まれていますが、ソルトにはランダムな、印刷不可能な文字を用いることもできます[†7]。

†5 監訳注：この例の場合は'G0y6cf3$.ydLVkx4I/50'がソルトです。また、MySQLで文字列連結を行うには、文字列連結演算子||の代わりにCONCAT関数を使うか、または「set @@session.sql_mode='PIPES_AS_CONCAT';」を実行する必要があります。

†6 ハッシュからパスワードを復元するためのさらに洗練された技術は、**レインボーテーブル**と呼ばれます。適切な長さのソルトを用いることで、レインボーテーブルによる解読も困難になります。

†7 監訳注：ソルトの用件と実装方法に関しては『体系的に学ぶ安全なWebアプリケーションの作り方』[Tok11]に詳しく記述されています。

19.5.4 SQLからパスワードを隠す

パスワードをデータベースに格納する前に強力なハッシュ関数でパスワードを暗号化し、さらに暗号化の前にソルトをパスワードに付加することで、辞書攻撃に対抗することができるようになりました。これで、セキュリティを十分に確保できるようになったと思うかもしれません。しかしパスワードは、SQL文の中ではまだ平文として使用されています。つまり、攻撃者によってネットワークパケットが傍受された場合や、SQLクエリが記録されたログファイルが攻撃者の手に渡ってしまった場合には、パスワードを読み取られてしまうのです。

これを防ぐための方法は、SQLクエリでパスワードを平文として使わないことです。アプリケーションコードでハッシュを計算し、SQLクエリではハッシュのみを用いるようにします。攻撃者がこのハッシュを傍受したとしても、メリットはありません。ハッシュからパスワードを解読することは難しいからです。

ハッシュの計算前には、ソルトの付加を忘れてはいけません。

以下に、PHPでハッシュ計算を行う例を示します。PDO拡張（PHPのDB接続ライブラリ）を用いてソルトを取得し、ハッシュを計算し、データベースに格納されたソルト付のハッシュとの比較によってパスワードの妥当性を確認しています。

Passwords/soln/auth-salt.php

```
<?php

$password = 'xyzzy';

$stmt = $pdo->query(
   "SELECT salt
    FROM Accounts
    WHERE account_name = 'bill'");

$row = $stmt->fetch();
$salt = $row[0];

$hash = hash('sha256', $password . $salt);

$stmt = $pdo->query("
  SELECT (password_hash = '$hash') AS password_matches
  FROM Accounts AS a
  WHERE a.account_name = 'bill'");

$row = $stmt->fetch();
if ($row === false) {
  // アカウント 'bill' は存在しない
} else {
  $password_matches = $row[0];
  if (!$password_matches) {
```

```
    // パスワードが間違っている
  }
}
```

サンプルコードのhash関数は、16進数のみを返すことが保証されています。このため、SQLインジェクションの危険はありません（20章「SQLインジェクション」を参照）。

ウェブアプリケーションには、まだ攻撃者によってネットワーク上のデータが傍受される可能性がある場所があります。それは、ユーザーのブラウザとウェブアプリケーションサーバの間のネットワークです。ユーザーがログインフォームをサブミットすると、ブラウザは平文でユーザーのパスワードをサーバーに送ります。パスワードがハッシュ化されるのは、受信先のサーバーです。このため、ブラウザからサーバーまでは平文でパスワードが送信されることになります。パスワードが平文で送られることを防ぐには、フォームのデータを送る前に、ブラウザ上でハッシュ関数によるパスワード暗号化を行うことも可能です。しかし、ハッシュ値を計算するためには、パスワードと関連付けられたソルトをサーバーから取得する必要があるため、この方法はスマートなものだとは言えません。このため、ブラウザからアプリケーションにパスワードを送るときには、**セキュアHTTP（HTTPS）**を使用する方法がよく用いられています。

19.5.5 パスワードをリカバリーするのではなく、リセットする

ここまでの方法で、パスワードをより安全な方法で格納できるようになりました。しかし、まだ本書の冒頭で見た問題は解決できていません。パスワードを忘れたユーザーを助ける方法です。しかし、もうパスワードはリカバリーできません。データベースには、パスワードではなくパスワードのハッシュを格納しているからです。ハッシュからパスワードを解読するのは、あなたにとっても攻撃者にとっても同じくらい難しいものになります。しかし、パスワードを忘れたユーザーにアクセスを許可する方法はあります。以下に、これらの方法の実装例を2つ紹介します。

1番目は、パスワードを忘れたユーザーが助けを求めてきたとき、パスワードを電子メールで送るのではなく、アプリケーションが生成した**一時パスワード**を電子メールで送る方法です。セキュリティを強化するために、一時パスワードを短時間で無効にすることもできます。これにより、その電子メールが傍受されても、不正なアクセスを防ぎやすくなります。もちろん、ユーザーの初回ログイン時にパスワードの変更を強制するようにアプリケーションを設計すべきです。

システムで生成された一時パスワードを記載する電子メールの例

```
From: daemon
To: bill@example.com
件名：パスワードのリセット

アカウントのパスワードリセット依頼に回答します。

一時パスワードは「p0trz3b1e」です。
このパスワードは1時間後に無効化され、アクセスできなくなります。
```

以下のリンクをクリックして、アカウントにログインし、新しいパスワードを設定してください。
http://www.example.com/login

2番目の方法は、電子メールに新しいパスワードを記載する代わりに、リクエストをデータベーステーブルに記録し、一意の**トークン**を識別子として割り当てるというものです。

Passwords/soln/reset-request.sql

```
CREATE TABLE PasswordResetRequest (
  token       CHAR(32) PRIMARY KEY,
  account_id  BIGINT UNSIGNED NOT NULL,
  expiration  TIMESTAMP NOT NULL,
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

SET @token = MD5('billkarwin' || CURRENT_TIMESTAMP || RAND());

INSERT INTO PasswordResetRequest (token, account_id, expiration)
  VALUES (@token, 123, CURRENT_TIMESTAMP + INTERVAL 1 HOUR);
```

次に、電子メールにトークンを記載します。送信先アドレスが、パスワードのリセットを要求したアカウントとすでに関連付けられている場合には、トークンをSMSなどを使って送信することも可能です。この方法であれば、第三者が違法にパスワードリセットをリクエストしてきた場合でも、アカウントの実際の所有者のみにトークンを記載した電子メールが送信されます。

パスワードリセットページへの一時的なリンクを記載した電子メールの例

```
From: daemon
To: bill@example.com
件名：パスワードのリセット

アカウントのパスワードリセット依頼に回答します。

1時間以内に以下のリンクをクリックしてパスワードを変更してください。
1時間が経過するとリンク先のページにはアクセスできなくなり、パスワードも変更できません。

http://www.example.com/reset_password?token=f5cabff22532bd0025118905bdea50da
```

アプリケーションがパスワードリセットページへのリクエストを受信したとき、tokenパラメータの値はPasswordResetRequestテーブルの行と一致していなければならず、かつこの行のトークン失効期限（expiration）は過去ではなく未来時刻でなくてはなりません。またこの行のaccount_idはAccountsテーブルを参照しているので、トークンは特定の1つのアカウントのパスワードしかリセットできないように制限されます。

もちろん、第三者にこのページへアクセスされた場合には危険が生じます。ただし、単純な制限でこのリスクは低減できます。例えば、特殊画面の表示時間を短くしたり、パスワードの再設定対象のアカウントを画面に表示しないという方法があります。

暗号化技術は、攻撃者の技術に対抗するために、日々進化を続けています。この章で紹介した技法は多くのアプリケーションにとって有益なものであるはずですが、極めてセキュアなシステムを開発する必要がある場合は、以下のようなさらに高度な技法の採用を検討してみましょう[†8]。

- PBKDF2 - 広く普及している暗号化標準で、**鍵強化**（keystrengthening）に使用されています http://tools.ietf.org/html/rfc2898
- Bcrypt - **アダプティブハッシュ**（adaptivehashing）**関数**の実装です http://bcrypt.sourceforge.net/

あなたが読み取れるものは、攻撃者にも読み取れます。

†8 監訳注：暗号化およびその解読技術は日進月歩で進む世界です。近年は特にクラウドコンピューティングの登場によって計算量のハードルが圧倒的に下がり、この章の解決策として挙げられているハッシュ関数＋ソルトの解読もごく短時間で行われる可能性がある時代になりました。このため、読者の皆様はこの章の解法に安住することなく、常に最新の情報を調査することをお勧めします。また、パスワードの管理方法を含むウェブアプリケーションのセキュリティ全般に関しては『体系的に学ぶ安全なWebアプリケーションの作り方』[Tok11]に詳しく記されています。

20章
SQLインジェクション

「発言を誤解されて引用された」という私の言葉を、引用してほしい
——グルーチョ・マルクス（1930年代を中心に活躍したアメリカのコメディアン、マルクス兄弟の三男）

2010年3月、史上最大の個人情報連続窃盗事件の主犯として、連続犯クラッカー、アルバート・ゴンザレスに有罪判決が下されました。ゴンザレスは複数の大手小売店チェーンとそのクレジットカード処理業者のATMマシンおよび支払いシステムに不法アクセスし、推定1億3000万件のクレジットカード番号とデビットカード番号を盗みました。

ゴンザレスはこの窃盗で、自らが持つ過去の記録を塗り替えました。2006年に、ワイヤレスネットワークの脆弱性を攻撃し、4560万件のクレジットカード番号とデビットカード番号を窃盗した事件です。

4560万件というそれまでの「記録」の3倍近い1億3000万件という大量のデータを、ゴンザレスはどのようにして盗むことができたのでしょうか。思わず、007の映画に出てくるような黒ずくめの窃盗団が、エレベーターシャフトを伝ってコンピューター室に忍び込み、最新技術で暗号化されたパスワードをスーパーコンピューターを使ってクラックしたり、都市全体の電力を停電させたりするような光景を浮かべたくなります。

しかし現実は、至極ありきたりなものでした。ゴンザレスは、インターネットの最も一般的なセキュリティの弱点を突いただけだったのです。**SQLインジェクション**（SQL Injection）と呼ばれる攻撃技術を用いて被害者の企業のサーバーに特権的にアクセスし、窃盗のためのプログラムをアップロードしたのです。起訴状には、ゴンザレスとその共謀者がサーバーへのアクセス後に何をしたかについて、次のように記述しています[†1]。

攻撃の実行：マルウェア

……ゴンザレスらはプログラム「スニッファー（sniffer）」をインストールした。このプログラムは、被害者企業のネットワーク内を行き来するクレジットカード及びデビットカードの処理情報から、クレジットカード番号、デビットカード番号、対応するカードデータなどの情報をリアルタイムで取得し、これらの情報を定期的にゴンザレスらに送信するものであった。

†1　http://voices.washingtonpost.com/securityfix/heartlandIndictment.pdf

ゴンザレスに攻撃されたウェブサイトを運営する小売り業者は、その後セキュリティホールを修正したと述べています。しかし、セキュリティホールが修正されたのはこの企業のウェブサイトのみです。今日も世界のあちこちでウェブアプリケーションが開発されていますが、そのなかには脆弱性があるものも少なくありません。ソフトウェア開発者の多くがこの脆弱性の本質を理解しておらず、またSQLインジェクションを防ぐためのコードの書き方を知りません。このため、SQLインジェクション脆弱性は依然として攻撃者にとって格好の攻撃対象のままなのです。

20.1 目的：動的SQLを記述する

多くの場合、SQLはアプリケーションコードと連携して使われます。SQLクエリを文字列として作成し、文字列内にアプリケーション変数を挿入する方法は、一般的に**動的SQL**と呼ばれます[†2]。

SQL-Injection/obj/dynamic-sql.php

```
<?php
... 他のコード
$sql = "SELECT * FROM Bugs WHERE bug_id = $bug_id";
$stmt = $pdo->query($sql);
```

このシンプルな例では、文字列にPHPの変数を挿入しています。$bug_idが整数であり、データベースがクエリを受け付けるときには、$bug_idの値がクエリの一部になっていることを意図しています。

動的SQLクエリは、データベースを最大限に活用するための自然な利用法です。データベースへのクエリ内容をアプリケーションデータによって変化させるとき、SQLは対話的な言語としての側面を見せます。人間ではなくアプリケーションがデータベースと対話するのです[†3]。

望む仕事をソフトウェアに実行させることは、それほど難しくはありません。難しいのは、望まない操作を許可しないように、ソフトウェアをセキュアにすることです。後者を怠ると、ソフトウェアにはSQLインジェクションの攻撃対象となる脆弱性が生じてしまうことになります。

20.2 アンチパターン：未検証の入力をコードとして実行する

SQLインジェクションは、SQLクエリ文字列に動的に挿入された文字列が、開発者の意図していない方法でクエリの構文を改変することによって生じます。SQLインジェクションの典型例は、挿入された値によって、元のSQL文の実行に続けて別のSQL文も実行されるというものです。例えば、$bug_id変数の値が文字列「1234; DELETE FROM Bugs」である場合、先ほどのサンプルのSQLは次のようになります。

SQL-Injection/anti/delete.sql

```
SELECT * FROM Bugs WHERE bug_id = 1234; DELETE FROM Bugs
```

†2 正確には、実行時に解析されるクエリはすべて動的SQLです。しかし一般的には、変数データを含むSQLを指します。

†3 監訳注：「埋め込みSQLにある根本的な原理は、**双対原理**（dual-mode principle）とよぶもので、対話的に使われるあらゆるSQL文はアプリケーション・プログラムでも使えることを指す」『データベースシステム概論』[Dat94]より。

図 20-1の Randall Munroe による漫画[†4] が示すように、このタイプの SQL インジェクションは、甚大な被害をもたらす可能性があります。

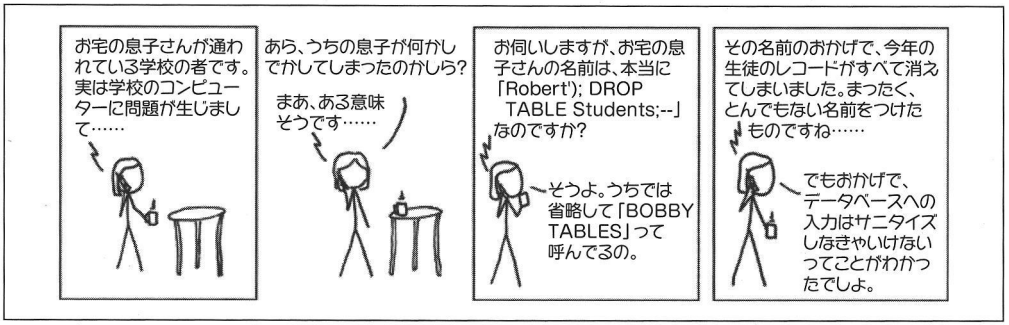


図 20-1　ママの SQL インジェクション

20.2.1 アクシデントは起きる

バグデータベースを閲覧するためのウェブインタフェースを作成しているとします。あるページでは、プロジェクトの名前を指定して画面に遷移します。

SQL-Injection/anti/ohare.php

```
<?php
$project_name = $_REQUEST["name"];
$sql = "SELECT * FROM Projects WHERE project_name = '$project_name'";
```

ところが、クライアントにシカゴの「O'Hare International Airport（オヘア国際空港）」が加わったとき、問題が生じます。プロジェクトには当然、「O'Hare」のような名前をつけます。この場合、ウェブアプリケーションでプロジェクトを閲覧するために、どのようなリクエストを実行するでしょうか。

```
http://bugs.example.com/project/view.php?name=O'Hare
```

PHP コードはリクエストパラメータの値を SQL クエリに挿入します。すると、開発者やユーザーが意図しないクエリを作り出してしまいます。

SQL-Injection/anti/ohare.sql

```
SELECT * FROM Projects WHERE project_name = 'O'Hare';
```

文字列は最初に検出された引用符文字で終了するため、結果として SQL には短い文字列「'O'」と、その後ろに、この文脈では意味をなさない文字「Hare'」が含まれることになり、データベースは構文エラーを

†4　http://xkcd.com/327/（使用許可取得済み）

返します。ただ、このエラーは特に有害なものではありません。構文エラーとなるステートメントは実行できないため、リスクは小さいからです。リスクが高まるのは、構文エラーにならずにステートメントが実行され、意図していない何かが行われてしまう場合です。

20.2.2 ウェブ最大のセキュリティ脅威

攻撃者がSQLインジェクションを使ってSQLステートメントを操作できる場合、**脅威は飛躍的に大きくなります**。例えば、ユーザーがパスワードを変更できるアプリケーションがあるとします[†5]。

SQL-Injection/anti/set-password.php

```
<?php
$password = $_REQUEST["password"];
$userid = $_REQUEST["userid"];
$sql = "UPDATE Accounts SET password_hash = SHA2('$password', 256)
    WHERE account_id = $userid";
```

賢い攻撃者は、リクエストパラメータがSQLステートメントでどのように使われるかを推測し、悪用するために巧妙な文字列を送信します。

```
http://bugs.example.com/setpass?password=xyzzy&userid=123 OR TRUE
```

useridパラメータからSQL式へ挿入された文字列は、ステートメントの構文を改変します。この改変によって、特定の1つのアカウントではなく、データベースの「すべての」アカウントのパスワードが変更されてしまいます。

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy', 256)
WHERE account_id = 123 OR TRUE;
```

ここが、SQLインジェクションを理解し、それを防御するための鍵です。つまり**SQLインジェクションは、SQLステートメントが解析される前に構文を改変する**のです。SQLステートメントの解析前に挿入される動的な内容を扱う限り、SQLインジェクションで攻撃されるリスクがあるということです。

悪意をもって選択された文字列は、実に様々な方法でSQLステートメントの振舞いを改変できます。限界があるとすれば、それは攻撃者の想像力や、開発者がSQLステートメントを守る能力の方なのです。

20.2.3 対処法の追求

SQLインジェクションの脅威について理解したら、次は当然、コードを保護するための方法を考えなく

†5 監訳注：この章のパスワードハッシュの扱いは、おそらくサンプルコードを短くするために単純に書かれています。ハッシュ関数を使用したパスワードハッシュ作成をより詳しく知るためには、19章「リーダブルパスワード（読み取り可能パスワード）」を参照してください。

てはなりません。ブログや雑誌ではよく、あらゆるSQLインジェクションの対処策となる1つの方法があると主張する記事を見かけます。しかし実際は、以下に述べる技法のどれか1つだけで、あらゆる形態のSQLインジェクションを防御することはできません。状況に応じて技法を適切に使い分けなければならないのです。

値のエスケープ

SQLのクエリで、意図せず引用符文字の不一致が発生することを避ける古典的な方法は、引用符をすべて**エスケープ**して、引用符によって引用符内の文字列が途中で終了してしまうことを阻止するという方法です。SQL標準では、引用符文字を2つ重ねることによって、1つの引用符文字を表せます。

SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O''Hare';
```

他の多くのプログラミング言語と同様に、各種データベース製品では、後続の文字をエスケープするためにバックスラッシュ（\）を使用できます。

SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O\'Hare';
```

ここまで述べてきたのは、SQL文字列内へ挿入する前にアプリケーションがデータを変更するという方法です。たいていのSQLプログラミングインタフェースでは、この用途のために便利な関数が提供されています。例えば、PHPのPDO拡張モジュールでは、quote関数を用いて、入力文字列を引用符文字で囲みつつ、入力文字列内のすべての引用符文字リテラルのエスケープを行えます。

SQL-Injection/anti/ohare-escape.php

```
<?php
$project_name = $pdo->quote($_REQUEST["name"]);
$sql = "SELECT * FROM Projects WHERE project_name = $project_name";
```

引用符文字リテラルのエスケープによって、動的コンテンツ内の引用符の不一致を原因とするSQLインジェクションのリスクを減らせます。ただし、この手法は文字列以外のデータには用いることができません。

SQL-Injection/anti/set-password-escape.php

```
<?php
$password = $pdo->quote($_REQUEST["password"]);
$userid = $pdo->quote($_REQUEST["userid"]);
$sql = "UPDATE Accounts SET password_hash = SHA2($password, 256)
    WHERE account_id = $userid";
```

SQL-Injection/anti/set-password-escape.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy', 256)
WHERE account_id = '123 OR TRUE'
```

データベースは基本的に、数値型の列を数字で構成された文字列と直接比較することはできません。一部のデータベースは暗黙的に文字列を等価な数値に変換できますが、SQL標準では、文字列を数値型に変換するにはCAST関数を意図的に用いなければなりません。

また、引用符文字をエスケープする関数を非ASCIIのキャラクタセットが通過してしまい、エスケープされなかった引用符文字がそのまま残ってしまう[†6]という、わかりにくく、まれなケースもあります[†7]。

プリペアドステートメント

SQLインジェクションの万能薬的な解決策として最も頻繁に語られるのが、**プリペアドステートメント**です。SQL文字列内に動的に値を挿入する代わりに、クエリの前処理（プリペア）時にパラメータのプレースホルダーを文字列内に残します。そして、準備したクエリを実行する前にクエリパラメータとして値を渡します。

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Projects WHERE project_name = ?");
$stmt->bindValue(1, $_REQUEST["name"], PDO::PARAM_STR);
$stmt->execute();
```

多くのプログラマーがこの解決策を推奨しています。動的コンテンツのエスケープが不要であり、エスケープ関数の欠陥も心配しなくて済むからです。実際、プリペアドステートメントはSQLインジェクションに対する**最も強力な防御手段**です。ただし、パラメータの値が常に1つのリテラル値と解釈されるため、万能に見えるプリペアドステートメントといえども、あらゆる状況に適用できる解決策ではないのです。以下に例を示します。

値のリストを1つのパラメータにすることはできない

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id IN ( ? )");
$stmt->bindValue(1, "1234,3456,5678", PDO::PARAM_STR);
$stmt->execute();
```

†6 http://bugs.mysql.com/bug.php?id=8378 の例を参照してください。

†7 監訳注：非ASCIIのキャラクタセットを主に扱う日本においては、キャラクタセットに関連した脆弱性は現実的な脅威となります。PDOの非ASCIIのキャラクタセットの扱いに起因する脆弱性、およびその後の文字コード指定機能の詳細については「PDOにおける一応の安全宣言と残る問題点」を参照してください。http://blog.tokumaru.org/2011/08/pdo.html

この場合、MySQLではカンマ区切りの最初の数値のみが文字列パラメータとして渡されたかのように処理されてしまうため、複数の整数値を渡したときと同じような振舞いにはなりません[†8]。

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs WHERE bug_id IN ( '1234' )
```

テーブル識別子もパラメータとして扱えない

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM ? WHERE bug_id = 1234");
$stmt->bindValue(1, "Bugs", PDO::PARAM_STR);
$stmt->execute();
```

この場合にはテーブル名の代わりに文字列リテラルを入力したかのように処理され、構文エラーが返されます。

SQL-Injection/anti/parameter.sql

```
SELECT * FROM 'Bugs' WHERE bug_id = 1234
```

列名もパラメータとして扱えない

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY ?");
$stmt->bindValue(1, "date_reported", PDO::PARAM_STR);
$stmt->execute();
```

この例では、式が文字列定数として解釈され、その値はすべての行で同じであるため、ソートしても何も起こりません。

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY 'date_reported' ;
```

†8 監訳注：本文の振舞いはMySQLのものです。PostgreSQLで上記プリペアドステートメントを実行すると、エラーが発生します。PHP Warning: PDOStatement::execute(): SQLSTATE[22P02]: Invalid text representation: 7 ERROR: invalid input syntax for integer: "1234,3456,5678"

SQL予約語もパラメータにならない

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY date_reported ?");
$stmt->bindValue(1, "DESC", PDO::PARAM_STR);
$stmt->execute();
```

パラメータは、SQL予約語ではなく、リテラル文字列と解釈されます。この例の結果は構文エラーになります。

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY date_reported 'DESC'
```

構築済みクエリはどんなものか

多くの人が、プリペアドステートメントとは、きちんとエスケープした値をSQLステートメント内に自動的に入れる方法であると考えています。厳密に言うと、これは間違っています。プリペアドステートメントに対するこのような理解は、振舞いの誤解につながります。

RDBMSサーバーは、クエリの前処理（プリペア）時にSQLを解析します。その後は、SQLクエリの構文は変更できません。

前処理を行ったクエリを実行するときに値が渡されます。渡された値は、それぞれ対応したプレースホルダーで使われます。

前処理が行われたクエリは、前回実行時のパラメータの代わりに新しいパラメータ値を用いて、再び実行できます。このためRDBMSはクエリとパラメータ値を分けて保存する必要があります。これはセキュリティ面で見ても良いことであるといえます。

つまり、準備されたSQLクエリ文字列を取得しても、そこにはパラメータ値が含まれていません。もしSQLステートメントにパラメータ値が含まれていれば、クエリのデバッグやロギングの際に便利になるでしょう。しかしパラメータ値は、人間が読むことができる形式のSQLでクエリと組み合わされることはないのです。

動的SQLステートメントをデバッグする最善策は、パラメータプレースホルダーを含むステートメントは作成時に、パラメータ値は実行時に、それぞれロギングすることなのです。

ストアドプロシージャ

ストアドプロシージャもまた、多くのソフトウェア開発者がSQLインジェクションに対して効果的だと主張する方法です。一般的なストアドプロシージャには、プロシージャの定義時に解析された静的なSQLステートメントが記述されています。

しかし、ストアドプロシージャにも安全性の低い動的SQLを使用することができます。以下の例では、

input_userid変数は動的にSQLクエリ内に挿入されます。もちろん、この行為は安全ではありません。

SQL-Injection/anti/procedure.sql

```
CREATE PROCEDURE UpdatePassword(input_password VARCHAR(20),
  input_userid VARCHAR(20))
BEGIN
  SET @sql = CONCAT('UPDATE Accounts
    SET password_hash = SHA2(', QUOTE(input_password), ', 256)
    WHERE account_id = ', input_userid);
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
END
```

ストアドプロシージャで動的SQLを扱うのは、アプリケーションコードで動的SQLを使用するのと同じくらい危険です。input_userid変数に危険な文字列を含めることができるため、次のような安全性の低いSQLステートメントになる可能性があるのです。

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy', 256)
WHERE account_id = 123 OR TRUE;
```

データアクセスフレームワーク

データアクセスフレームワークやオブジェクトリレーショナルマッピング（ORM）フレームワークによって、SQLインジェクションのリスクからコードを保護できると主張する人がいます。しかし、この主張は利用者がSQLステートメントを文字列として直接記述できるフレームワークには当てはまりません。

安全なSQLコードを保証するフレームワークはありません。フレームワークは便利な機能を提供してくれるかもしれませんが、その機能を使わずに、一般的な文字列操作でSQLステートメントを容易に構築できてしまいます。その結果、リスクが生じてしまうのです。

毎日、歯を磨くように

私が、自分が開発したPHP製のデータアクセスフレームワークについて講演したときのことです。講演後、聴講していた1人の若者が近付いてきて「そのフレームワークは、SQLインジェクションを防げますか？」と言いました。私は、フレームワークには文字列を引用符で囲む機能と、プリペアドステートメントを使う機能があると答えました。

若者は当惑した表情を浮かべて、先ほどと同じ質問を繰り返しました。「つまり、SQLインジェクションを防げるということですか？」彼は、自分が気付かないミスを自動的に防いでくれる方法を求めていたのです。

私は、「歯ブラシ」のようにこのフレームワークを使えば、SQLインジェクションを防げるだろうと彼に

伝えました。虫歯を防ぐためには毎日歯磨きをしなければならないように、フレーワークの機能も毎回忘れずに使わなければならないのです。

20.3 アンチパターンの見つけ方

実際のところ、**ほぼすべての**データベースアプリケーションは、動的にSQLステートメントを構築します。SQLステートメントの一部分を文字列連結や変数の挿入を用いて構築する場合、そのステートメントによって、アプリケーションはSQLインジェクション攻撃を受けるリスクにさらされます。

SQLインジェクション脆弱性は極めて一般的であるため、SQLインジェクション専門のコードレビューを実施して問題を修正していない限り、SQLを使用する**ほぼすべての**アプリケーションが、多かれ少なかれ、このアンチパターンに該当する可能性があると想定すべきです。

「ルール31」：後部座席に気をつけよう[†9]

ホラー映画には、自動車の後部座席に隠れていたゾンビのような恐ろしい怪物が、運転席に誰かが座ったとたんに襲いかかってくるというシーンがよくあります。ここから得られる教訓は、自動車のような慣れ親しんだスペースの中であっても、リスクは存在すると考えるべきだということです。

SQLインジェクションは、間接的な形式をとることができます。ユーザーから入力されたデータをプリペアドステートメントを用いて安全に格納したとしても、後でそのデータが他の動的SQLクエリのために使われるかもしれません。

```
<?php
$sql1 = "SELECT last_name FROM Accounts WHERE account_id = 123";
$row = $pdo->query($sql1)->fetch();
$sql2 = "SELECT * FROM Bugs WHERE MATCH(description) AGAINST ('"
  . $row["last_name"] . "')";
```

このクエリで、ユーザーが名前を「O'Hara」と記述したり、SQL構文を意図的に操作するための文字列を挿入した場合、どうなるでしょうか。

20.4 アンチパターンを用いてもよい場合

このアンチパターンの性質は、本書の他のアンチパターンとは異なります。つまり、SQLインジェクションのようなセキュリティ上の脆弱性を正当化する理由は**ありません**。ソフトウェア開発者には、SQLインジェクションを防ぐために防御的なプログラミングを進んで行い、同僚が防御的なプログラミングを行いやすいよう支援する責任があります。ソフトウェアの安全性のレベルは、最も脆弱な部分で判断されるということを忘れないようにしましょう。

†9 訳注：映画『ゾンビランド』で主人公が実践する「ゾンビだらけの世界で生き残るための32のルール」のうちの1つ。

20.5 解決策：誰も信用してはならない

SQLコードの安全性を保証するための唯一絶対の方法などというものは存在しません。以下の技法のすべてを習得し、状況に応じて適切に使い分けるようにしましょう[†10]。

20.5.1 入力のフィルタリング

ユーザーからの入力に危険な文字列が含まれていないかどうかを探るよりも、その入力にとって無効な文字を初めからすべて取り除くようにすべきです。すなわち、整数が必要な場合は、入力内容から整数で構成される部分のみを使います。実現方法はプログラミング言語によって異なります。例えばPHPでは、filter拡張を使います[†11]。

SQL-Injection/soln/filter.php

```
<?php
$bugid = filter_input(INPUT_GET, "bugid", FILTER_SANITIZE_NUMBER_INT);
$sql = "SELECT * FROM Bugs WHERE bug_id = $bugid";
$stmt = $pdo->query($sql);
```

数値のような単純なケースでは、型キャスト関数を使えます。

SQL-Injection/soln/casting.php

```
<?php
$bugid = intval($_GET["bugid"]);
$sql = "SELECT * FROM Bugs WHERE bug_id = $bugid";
$stmt = $pdo->query($sql);
```

正規表現を使って安全な部分文字列を取得することで、無効な内容をフィルタリングする方法もあります。

SQL-Injection/soln/regexp.php

```
<?php
$sortorder = "date_reported"; // default

if (preg_match("/([_[:alnum:]]+)/", $_GET["order"], $matches)) {
  $sortorder = $matches[1];
}

$sql = "SELECT * FROM Bugs ORDER BY $sortorder";
$stmt = $pdo->query($sql);
```

†10 監訳注：SQLの安全な扱い方に関しては、独立行政法人情報処理推進機構（IPA）の「安全なSQLの呼び出し方」[IPA10]に、またウェブアプリケーションのセキュリティ全般に関しては『体系的に学ぶ安全なWebアプリケーションの作り方』[Tok11]に詳しく記されています。

†11 監訳注：手作業による入力フィルタリングは誤りが混入しやすい手法なので、フィルタリングだけに頼るのは危険です。SQLインジェクション対策は、まず初めにプリペアドステートメントの使用を検討するべきです。プリペアドステートメントに関しては、以降の項で説明されます。

20.5.2 動的値のパラメータ化

動的な部分がシンプルな値から構成されているときには、SQLから分離させるために**プリペアドステートメント**を用いることができます。

SQL-Injection/soln/parameter.php

```
<?php
$sql = "UPDATE Accounts
  SET password_hash = SHA2(?, 256)
  WHERE account_id = ?";
$stmt = $pdo->prepare($sql);
$stmt->bindValue(1, $_REQUEST["password"], PDO::PARAM_STR);
$stmt->bindValue(2, $_REQUEST["userid"], PDO::PARAM_INT);
$stmt->execute();
```

先ほどの「アンチパターン」の節で説明した通り、パラメータには単一の値しか用いることができません。RDBMSがSQLステートメントを解析した後にパラメータの値を渡すので、SQLインジェクション攻撃は、パラメータ化されたクエリの構文を改変できなくなるのです。攻撃者が「123 OR TRUE」のような悪意ある値をパラメータに渡しても、RDBMSはそのパラメータを値であると解釈します。最悪の場合でも、クエリがどの行にも合致しないだけです。悪意がある値が入力された場合も、結果は以下のような安全なSQLステートメントとして解釈されます[†12]。

SQL-Injection/soln/parameter.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy', 256)
WHERE account_id = '123 OR TRUE'
```

つまり、アプリケーション変数をリテラル値としてSQL文字列と連結する必要がある場合は、プリペアドステートメントを使うべきです。

20.5.3 動的値を引用符で囲む

SQLインジェクション対策には、ほぼすべての場合において、プリペアドステートメントが最善策です。しかし、ごくまれに、パラメータプレースホルダーを持つクエリのインデックスに関して、クエリオプティマイザーがおかしな判断をしてしまうことがあります。

例えば、Accountsテーブルに、account_statusという名前の列があるとします。そしてこの列では、行の99パーセントに文字列「ACTIVE」が格納されているとしましょう。値の配分が不均等になっているため、「account_status = 'BANNED'」を求めるクエリではインデックスのメリットを得られますが、「account_status = 'ACTIVE'」を求めるクエリの場合、インデックスを読みに行くとむしろ非効率になってしまいます。しかし、プリペアドステートメントでaccount_status = ?という式を使う場合、オプティマイザーは

†12 監訳注：MySQLで動作確認を行ったところ、MySQLでは「123 OR TRUE」の最初の数値部分（123）のみが数値パラメータとして渡されたかのように処理されました。.

作成済みのクエリを実行する際にどちらの値が来るのかを判断できません。このため、不適切な最適化を行ってしまう場合があるのです。

このような非常にまれなケースでは、一般的に推奨されているプリペアドステートメントではなく、SQLステートメントの中に直接値を挿入する方がよい場合もあります。その際には、十分に注意して文字列を引用符で囲む必要があります[†13]。

SQL-Injection/soln/interpolate.php

```
<?php
$quoted_status = $pdo->quote($_REQUEST["status"]);
$sql = "SELECT * FROM Accounts WHERE account_status = $quoted_status";
$stmt = $pdo->query($sql);
```

こうした複雑でわかりにくいSQLのセキュリティ問題に対しては、成熟した、よくテストされた関数を使うようにしましょう。ほとんどのデータアクセスライブラリには、文字列を引用符で囲むための関数があります。例えばPHPではPDO::quoteです。**これらのセキュリティの問題を完全に理解するまでは、絶対に自分で実装しようとは思わないでください。**

IN述語のパラメータ化

前述したように、1つのパラメータにはカンマ区切りの文字列は渡せません。このため、IN述語をパラメータ化するためには、リストの要素数と同じ数のパラメータを用意しなければなりません。

例えば、6つのバグを主キーで検索するために、$bug_list変数にキーの配列を入れたとしましょう。

```
<?php
$bug_list = array(12,23,34,45,56,67);
$sql = "SELECT * FROM Bugs WHERE bug_id IN (?, ?, ?, ?, ?, ?)";
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

このクエリが実行できるのは、$bug_list内に、プレースホルダーの数と一致する6つの要素が入っている場合のみです。より汎用的にIN述語を扱うためには、$bug_list内の要素数と同じ数のプレースホルダーを用いて、SQLのIN述語を動的に構築しなければなりません。

以下のサンプルコードは、いくつかの組込み配列関数を用いて$bug_listと同じ数のプレースホルダーを生成し、次にこれらをカンマ区切りで連結して、SQLに挿入します。

```
<?php
$bug_list = array(12,23,34);
$sql = "SELECT * FROM Bugs WHERE bug_id IN ("
   . join(",", array_fill(0, count($bug_list), "?")) . ")";
```

†13 監訳注：このサンプルの場合にはstatusパラメータの有効値が特定の文字列に絞られるので、10章「サーティワンフレーバー（31のフレーバー）」アンチパターンに注意しつつ、後に出てくる「ユーザーの入力をコードから隔離する」の項で説明されているような、ユーザーの入力値をマッピングする手法を使う方が良いでしょう。

```
// $sql 変数には "SELECT * FROM Bugs WHERE bug_id IN (?,?,?)" が入る
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

この技法は、値のリストをパラメータ化するために使用できます。

20.5.4 ユーザーの入力をコードから隔離する

プリペアドステートメントやエスケープの技法は、SQL式内にリテラル値を組み込むためには役立ちますが、その他の部分、例えばテーブル識別子や列識別子、SQL予約語には適用できません。クエリのこれらの部分を動的にするには、別の解決策が必要です。

ユーザーが、ステータスや作成日などの条件でバグをソートしたいとします。また、ソートが昇順か降順かも選べるとしましょう。

SQL-Injection/soln/orderby.sql

```
SELECT * FROM Bugs ORDER BY status ASC
SELECT * FROM Bugs ORDER BY date_reported DESC
```

以下の（悪い）例では、PHPスクリプトがリクエストパラメータorderと dirを取り、コードがこれらのユーザー選択を列名及び予約語としてSQLクエリに挿入します。

SQL-Injection/soln/mapping.php

```
<?php
$sortorder = $_REQUEST["order"];
$direction = $_REQUEST["dir"];
$sql = "SELECT * FROM Bugs ORDER BY $sortorder $direction";
$stmt = $pdo->query($sql);
```

スクリプトは、orderに列名が、 dirにASCまたはDESCが含まれることを想定していますが、この想定は安全ではありません。ユーザーは、ウェブからのリクエストでどのようなパラメータ値でも送れるからです。

代替手段として、リクエストパラメータの値を用いて定義済みの値を参照し、SQLでは定義済みの値を使うという技法があります。

1. ユーザーの入力値をキー、SQL列名を値にマッピングする$sortorders配列を宣言します。また、ユーザーの入力値をキー、SQL予約語ASC、DESCを値にマッピングする $directions配列を宣言します。

SQL-Injection/soln/mapping.php

```
$sortorders = array( "status" => "status", "date" => "date_reported" );
```

```
$directions = array( "up" => "ASC", "down" => "DESC" );
```

2. ユーザーの入力値が配列内に存在しない場合のデフォルト値として、$sortorder変数と$direction変数を設定します。

SQL-Injection/soln/mapping.php

```
$sortorder = "bug_id";
$direction = "ASC";
```

3. ユーザーの入力値が、$sortordersと $directionsで宣言した配列キーと一致する場合、対応する値を使用します。

SQL-Injection/soln/mapping.php

```
if (array_key_exists($_REQUEST["order"], $sortorders)) {
  $sortorder = $sortorders[ $_REQUEST["order"] ];
}

if (array_key_exists($_REQUEST["dir"], $directions)) {
  $direction = $directions[ $_REQUEST["dir"] ];
}
```

4. これで、SQLクエリで$sortorder変数と $direction変数を安全に使用できます。これらの変数には、コード内で宣言した、安全性の保証された値のみが格納されているからです。

SQL-Injection/soln/mapping.php

```
$sql = "SELECT * FROM Bugs ORDER BY $sortorder $direction";
$stmt = $pdo->query($sql);
```

この技法には、以下のようなメリットがあります。

- ユーザーの入力内容とSQLクエリの連結が行われません。このため、SQLインジェクションのリスクが減ります。
- SQLステートメントのどの部分でも動的にできます（識別子やSQL予約語だけでなく、式全体さえも動的に構築できます）。
- ユーザー入力値の妥当性確認を、簡単かつ効率的に行えます。
- ユーザーインタフェースから、データベースクエリ内部の詳細を分離できます。

選択肢はアプリケーション内部にハードコードされていますが、これはテーブル名、列名、SQL予約語にとっては適切です。文字列型や数値型のデータ列に対しては定義域全体が選択肢となる可能性があります

が、識別子や構文はそうではないからです。

20.5.5 他の開発者にコードをレビューしてもらう

欠陥を見つけるための最善策は、他人の目でチェックをしてもらうことです。SQLインジェクションのリスクに精通しているチームメイトに、**コードレビュー**を行ってもらいましょう。プライドやエゴにこだわってはいけません。レビューを受けているときは少々恥ずかしい思いをするかもしれませんが、それを怠ったばかりに、攻撃者にウェブサイトを攻撃され、その責任を負わなくてはならなくなるかもしれないのです。

SQLインジェクション対策のコードレビューをする際には、以下のガイドラインに従いましょう。

1. アプリケーション変数や文字列連結、文字列置換によって構築されているSQLステートメントを特定します。
2. SQLステートメントで使われている、すべての動的コンテンツの起点を辿り、外部ソースから来るすべてのデータを特定します（ユーザー入力、外部ファイル、環境まわり、外部のウェブサービス、サードパーティーのコード、データベースから取得した文字列など）。
3. これらの外部コンテンツにはすべて、潜在的なリスクがあると想定します。フィルター、バリデーター、マッピング配列などを用いて、これら信頼性の低いコンテンツを変換します。
4. プリペアドステートメントまたは検証済みのエスケープ関数を用いて、SQLステートメントと外部データを組み合わせます。
5. 他にもストアドプロシージャなどの、動的SQLステートメントが隠れている可能性がある場所をチェックすることも忘れないようにしましょう。

コードレビューは、SQLインジェクションの欠陥を特定するための、最も正確かつ効率的な方法です。コードレビューを必須の作業とみなして、そのための時間を確保すべきです。そしてレビューをしてくれたチームメイトのコードに対して、今度はあなたがレビューをしましょう[†14]。

[ユーザーには、値の入力は許可しても、コードの入力を許可してはいけません。]

†14 監訳注：SQLの安全な扱い方に関しては、独立行政法人情報処理推進機構（IPA）の「安全なSQLの呼び出し方」[IPA10]に、またウェブアプリケーションのセキュリティ全般に関しては『体系的に学ぶ安全なWebアプリケーションの作り方』[Tok11]に詳しく記されています。

21章
シュードキー・ニートフリーク（疑似キー潔癖症）

重要な人は気にしない。どうでもいい人ほど気にするものだ。
（ディナーパーティーの座席位置を決めているときの言葉）
――バーナード・バルーク（アメリカの政治家）

2枚のレポートを手にした上司が、あなたのところにやってきました。「経理部門の担当者が、今期の四半期レポートと前期の四半期レポートの間に不一致があると言っている。調べてみたが、確かに不一致がある。うちの部門が管理するコンピューター機器のうち、新しく購入したものがほとんど消えている。どういうことだ？」

あなたはレポートを見て、不一致のパターンに気づきます。「いいえ、機器が消えたわけではありません。以前あなたは、経理部門から欠けている番号の機器はどうなっているのかと質問されると言って、データベースで欠けている行がないように整理することを私に命じましたよね。そこで私は、欠けていた行の番号を一部の行に割り当て直したのです。ですから、欠けている行はなく、1から最後尾の 12340までの番号はすべて連続しています。消えたように見える機器は、物理的には存在します。ただ、欠けていた行を埋めるために番号が繰り上っているのです。私は、指示通りにこの作業をしたつもりです。」

上司は首を横に振って言いました。「だがこの結果は、私が望んだものとは違うな。経理部門が減価償却を行うには、機器番号を管理する必要がある。機器番号は、各四半期レポートで同じでなくてはならない。それに、すべての機器にはIDのラベルを貼ってある。会社のすべての機器のラベルを貼り直そうとすれば、何週間もかかる。変更したID番号を元に戻してくれ」

あなたは上司の指示に従い、作業を開始するためにコンピューターの前に座りました。そのとき、ふと疑問が浮かびました。「今月購入した機器のIDはどうすればよいのだろう？ 番号振り直し後に購入されたこれらの機器には、振り直し前に使われていたIDも割り当てられている。IDを元に戻せば、以前の機器と今月購入した機器でIDが重複してしまう。どうすればいいんだ？」

21.1 目的：欠番を詰める

表21-1のように、番号が連番になっていないことを、特に気にしない人もいます。

表21-1 連番になっていない行

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider

一方、**欠番**を気にする人がいるのも理解できます。bug_idが 3の行がなぜ欠けているのかがわからないからです。「クエリで bug_id=3を検索しても結果が返されないのはなぜか？」「データベースはそのレコードを消してしまったのか？」「bug_id=3が存在しない理由はなぜか？」「そのバグが、重要な顧客から報告されたものだったら？」「自分は、このデータの損失について責任を取らされるのではないか？」

開発者は、こうした煩わしい問題を解決しようとして、「**シュードキー・ニートフリーク（疑似キー潔癖症）**」アンチパターンに陥ってしまいます。潔癖症患者はデータの整合性に責任を持つ人である場合が多いですが、ほとんどの場合、生成されたレポートの意味を自信を持って説明できるだけのデータベース技術の理解を持ち合わせていないのです。

21.2 アンチパターン：隙間を埋める

欠けている行を見つけたとき、多くの人は自然にそれを埋めたいと考えます。たいてい、次の2つの方法のうちのどちらかを用いるでしょう。

21.2.1 欠番を割り当てる

疑似キーの自動採番メカニズムを用いて新しい主キー値を割り当てる代わりに、新しい行に、検出した最も値の小さい欠番番号を割り当てる方法です。この方法では、データを挿入するたびに、1つずつ欠けている行は埋まっていきます（表21-2）。

表21-2 データを挿入するたびに、欠けている行が埋まる

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider
3	NEW	Visual TurboBuilder

しかし、最も小さな欠番の値を特定するためには、不要な自己結合クエリを実行しなければなりません。

Neat-Freak/anti/lowest-value.sql

```
SELECT b1.bug_id + 1 AS max_bug_id
FROM Bugs b1
LEFT OUTER JOIN Bugs AS b2 ON b1.bug_id + 1 = b2.bug_id
WHERE b2.bug_id IS NULL
ORDER BY b1.bug_id LIMIT 1;
```

本書の前半では、SELECT MAX(bug_id)+1 FROM Bugsなどのクエリを実行して一意の主キーを割り当てようとすると、並行処理の問題が生じることを説明しました[†1]。欠番の最小値を探す方法でも、2つのアプリケーションが同時に値を探そうとすると、同じ問題が生じます。2つが同じ主キー値を使おうとすると、片方は成功しますが、もう一方はエラーになります。非効率であり、エラーを招きやすい方法だといえます。

21.2.2 既存行に番号を振り直す

主キー値の欠番を、すぐにでも埋めたい場合があります。新しい行の挿入時に欠けている行を1つずつ埋めていく方法では、すべての欠番を埋めるには時間がかかります。そこで考えつく方法は、欠番を埋めてすべての値が連続するように、既存行のキーの値を更新するというものです。この方法は多くの場合、最大の主キー値を持つ行を特定し、そのキーを欠番の最小値に更新することを意味します。例えば、値4を、3に更新します（表21-3）。

Neat-Freak/anti/renumber.sql

```
UPDATE Bugs SET bug_id = 3 WHERE bug_id = 4;
```

表21-3 最大の主キー値を持つ行を特定し、そのキーを欠番の最小値に更新する

bug_id	status	product_name
1	NEW	Open RoundFile
2	FIXED	ReConsider
3	DUPLICATE	ReConsider

既存行に番号を振り直すためには、先ほどの例で新しい行を挿入するために行ったものと同様のやりかたで、欠番のキー値を特定する必要があります。また、主キー値を更新するためにUPDATEステートメントを実行する必要もあります。これらのステップはどちらも、**競合状態**を引き起こす可能性があります。しかも、欠番の数が多い場合は、ステップを何度も繰り返さなくてはなりません。

振り直したキーの値を、振り直す前にその行を参照していたすべての子レコードに反映する必要もあります。この作業は外部キー定義でON UPDATE CASCADEを宣言している場合は簡単ですが、宣言していない場合には、外部キー制約を一時的に無効にし、手作業ですべての子レコードを更新し、再び制約を有効化しなければなりません。データベースのサービスを中断しなければ作業を行えない可能性があり、大変な労力を伴う上に間違いの起こりやすいプロセスです。当然、このような作業は避けるべきです。

このように大変な作業を実施し、一時的に欠番がなくなったとしても、その状態は短命に終わります。疑似キージェネレーターが最後に生成した値を持つ行が削除あるいは変更されていたとしても、ジェネレーターが次に生成するのは、最後に生成した値より1つ大きな値です。プログラマーが想定しているような、現在のテーブルの最大値ではありません。bug_idの最大値 4を持つ行の bug_idを欠番の値に更新して欠番を埋めた場合、次にデフォルトの疑似キージェネレーターで生成される値は5であるため、4が新たな欠番になってしまうのです。

†1 3章「IDリクワイアド（とりあえずID）」のコラム「シーケンスの特殊なスコープ」を参照してください。

21.2.3 データ不一致の元

技術ジャーナリストのMitch Ratcliffeは、「コンピューターは、人類がこれまでに発明したもののなかで、最もミスを誘発しやすいものである——拳銃とテキーラを除いて」と述べています[†2]。

この章の冒頭の物語は、主キー値の再割り当てを行うリスクを表しています。もしも、別の外部システムが主キーの値でレコードを識別している場合、主キー値の再割り当てによってそのシステムはデータを参照できなくなってしまいます。

主キー値を再利用するのは、良い考えではありません。欠番は、正当な理由による行の削除やロールバックの結果だからです。例えば、account_idの789番を割り当てられたユーザーが、有害な電子メールを送信したためにシステムから追放されることになったとします。アプリケーションのポリシーに従い、この違反者のアカウントは削除されます。しかし、主キーを再利用する場合、ID789番は後に別のユーザーに割り当てられることになります。しかし、IDの再割り当てが行われた時点で、違反者が送信した有害なメールをシステムの全ユーザーが読み終えているとは限りません。このため、ID789番の再割り当て後に以前の違反者のメールを初めて読んだユーザーが、ID789番についての苦情を申し立ててくる可能性があります。その結果、新たにID789番を割り当てられたユーザーは、無実の罪で責められてしまうことになってしまいます。

使用されていないからといって、**疑似キーを再利用してはならない**のです。

21.3 アンチパターンの見つけ方

「**シュードキー・ニートフリーク（疑似キー潔癖症）**」アンチパターンに陥っている兆候を示す発言の例を以下に示します。

「挿入をロールバックした後で、自動生成されたID値を再利用するにはどうすればいい？」

疑似キーの割り当てはロールバックされません。もしロールバックされるとするならば、RDBMSは、トランザクション内で疑似キー値を割り当てなければならなくなります。そうすると、今度は複数のクライアントが同時にデータを挿入するときに、競合状態や待ち状態が発生する可能性が出てくるのです。

「bug_idの4番に何が起こったの？」

主キーの連続値に欠番があるのを見つけて、根拠のない不安を感じていると考えられます。

「使用されていない最初のIDを取得するクエリはどうやって書いたらいい？」

この処理を実行したい理由は、ほぼ間違いなく、IDを再割り当てしたいからです。

「番号が足らなくなったらどうしよう？」

使用されていないID値の再割り当てを正当化しようとしていると考えられます。

21.4 アンチパターンを用いてもよい場合

疑似キー値の変更を正当化する理由は**ありません**。なぜなら、疑似キーの値それ自体が重要な意味を持つ

†2 『MIT Technology Review』1992年4月より。

べきではないからです。主キー列の値が何らかの意味を持つ場合、その列は疑似キーではなく**自然キー（ナチュラルキー）**です。自然キーの値を変更することは珍しくありません。

21.5 解決策：疑似キーの欠番は埋めない

主キーの値は、一意で非NULLの値でなければなりません。各行を識別できなくてはならないからです。しかし、主キーのルールはそれだけです。行の識別のために連続している必要はないのです。

21.5.1 行のナンバリング

疑似キージェネレーターによって生成される値は、一般に単調増加する（前の値より1つ大きな値となる）ため、**行番号**と同等に見えます。両者は一意である点では似ていますが、疑似キージェネレーターにより生成される値には欠番が発生する可能性があり、行番号には欠番が発生しないという点で異なります。

行番号と主キーを混同してはいけません。主キーがテーブルにおける1行を識別するのに対し、行番号は結果セット（条件によりフィルタリングされたテーブルの一部の行）における行の順序番号を示します。クエリの結果セット内の行番号は、テーブルの主キー値に必ずしも一致しません。特に、JOIN、GROUP BY、ORDER BYなどを用いたクエリではそうです。

行番号の使用には、正当な理由があります。例えば、クエリ結果から行のサブセットを返したい場合などです。これは、しばしば「**ページネーション（Pagination）**」と呼ばれます（例：インターネット検索サイトにおける検索結果ページ）。サブセットを選択するには、クエリの形式にかかわらず、単調増加で連続した「真の行番号」を使う必要があります。

SQL:2003に定義されている**ウィンドウ関数**（window function）の中に、ROW_NUMBER関数があります。ROW_NUMBER関数は、クエリの結果セット固有の連番を返します。行のナンバリングは、クエリ結果を一定範囲の行に限定する場合によく使用されます。

Neat-Freak/soln/row_number.sql

```
SELECT t1.* FROM
  (SELECT a.account_name, b.bug_id, b.summary,
     ROW_NUMBER() OVER (ORDER BY a.account_name, b.date_reported) AS rn
   FROM Accounts a INNER JOIN Bugs b ON a.account_id = b.reported_by) AS t1
WHERE t1.rn BETWEEN 51 AND 100;
```

本書の執筆時点で、これらの関数は、Oracle、Microsoft SQL Server 2005、IBM DB2、PostgreSQL 8.4、Apache Derbyなど主要なデータベース製品の多くでサポートされています。またOracleは、ROW_NUMBER関数に加えて、行番号に相当するrownum疑似列をサポートしています。

MySQL、SQLite、Firebird、InformixはSQL:2003のウィンドウ関数をサポートしていませんが、行番号を用いたクエリ結果の限定に使える独自の構文を提供しています。MySQLとSQLiteは、LIMIT句をサポートしています。FirebirdとInformixは、予約語FIRSTとSKIPを使用できるクエリオプションをサポートしています。

整数は再生可能な資源か

シュードキー・ニートフリーク（疑似キー潔癖症）アンチパターンに関する誤解は他にもあります。それは「単調増加する疑似キージェネレーターは整数値を枯渇させてしまうかもしれないので、値を無駄にしてはいけない」という考えです。

一見、これは理にかなったもののように思えます。数学の世界では、整数の集合は「可算無限（数えることはできるが、無限）」です。しかしデータベースでは、どのようなデータ型でも値の数は有限です。32ビットの整数は、最大で2の32乗個の値を表せます。このため、主キーに値を割り当てるたびに、有限の値を1つ使用しているのは確かです。

これが何を意味するのか、計算をしてみましょう。例えば、新規行の挿入のために、一意の主キー値を毎秒1個、24時間休みなく生成していくとすると、32ビットの整数のすべての値を使うまでに136年かかります。

それでもまだ足りないというのであれば、64ビットの整数を使えます。64ビットの場合、毎秒100万個の値を生成したとしても、値を使い果たすまでに58万4542年かかります。

このように、整数値を枯渇させてしまうことは、ほとんど起こりえないのです。

21.5.2 GUIDの使用

同じ数を複数回使用しないために、ランダムな疑似キー値を生成する方法もあります。一部のデータベースは、この目的のために、**グローバル一意識別子**（Globally Unique IDentifier：GUID）をサポートしています[†3]。

GUIDは128ビットの疑似乱数です（通常、32個の16進数で表されます）。同じ識別子が生成される可能性が極めて低いために、GUIDは事実上、一意の値であると見なされます。このため、GUIDは疑似キーの生成に使用できます。

以下は、Microsoft SQL Server 2005の構文を使った例です。

Neat-Freak/soln/uniqueidentifier-sql2005.sql

```
CREATE TABLE Bugs (
bug_id UNIQUEIDENTIFIER DEFAULT NEWID(),
  -- 他の列. . .
);

INSERT INTO Bugs (bug_id, summary)
VALUES (DEFAULT, '保存時にクラッシュが発生');
```

この結果、表21-4のような行が作成されます。

†3 監訳注：一般にはUUID（Universally Unique IDentifier）という呼称の方が知られています。UUIDの定義はRFC4122で定義されています。http://www.ietf.org/rfc/rfc4122

表 21-4 GUID で採番された新規行

bug_id	summary
0xff19966f868b11d0b42d00c04fc964ff	保存時にクラッシュが発生

GUID は、従来型の疑似キージェネレーターと比べて、少なくとも以下の 2 つの利点をもたらします。

- 複数のデータベースサーバー間で、重複した値を生成することなく、並行して疑似キーを生成できます。
- 欠番に関する不満を誰も口にしなくなります（ただし、主キーの値を 32 個の 16 進数として入力しなければならないことをぼやく人は増えるでしょう）。

上記の 2 番目は、デメリットにもなります。

- 値が長いため、タイプしずらくなります。
- 値がランダムなので、値からパターンを推測したり、値の大小から生成された順番を推測したりすることができません。
- GUID の格納には 16 バイトが必要です。このため、一般的な 4 バイト整数の疑似キーと比べて多くのスペースが必要になり、実行時間も長くなります。

21.5.3 最も重要な問題

この章では、疑似キー列の番号を再割り当てすることの問題点と、それを回避するための解決策について学びました。しかし、もう 1 つ、解決すべき大きな問題が残っています。それは、疑似キーの欠番を埋めろという上司からの要求を、どうやって断るのかという問題です。これは技術ではなく、コミュニケーションの問題です。とはいえ、データベースのデータの整合性を守るために、あなたは「上司を管理する」必要があります。

技術について説明する

よく言われるように、正直は最善の策です。上司の要求に敬意を示し、なぜ上司がそれを要求したかを理解しましょう。そのうえで、上司には以下のように説明します。

「たしかに、欠番があると良い感じはしません。しかし、これらは無害なのです。行のスキップやロールバック、削除が時々生じるのは異常ではありません。新しい行には、安全に再利用できる欠番を探すのではなく、新しい番号を割り当てています。これによって開発コストを抑えつつ、処理は高速になり、エラーも削減できるのです」

コストの見積りを提示する

主キー値の変更は、些細なタスクだと見なされることがあります。しかし、そのコストを現実的に見積り、提示すべきです。コストには、新しい値の計算、重複値を処理するためのコードの記述とテスト、

データベース全体への変更の反映、他のシステムへの影響の調査、ユーザー及び管理者への新しい手順の教育などが含まれます。

管理職の多くは、タスクのコストに基づいて優先順位をつけます。この些細で最適化の効果も薄いタスクのために多くのコストがかかると知ったら、要求を撤回するでしょう。

自然キー（ナチュラルキー）を使用する

上司やデータベースのユーザーから、主キー値を意味のあるものにしてほしいとせがまれたら、その要求に応じましょう。この場合、疑似キーは用いず、識別しやすい意味を表す文字列や数値を使います。これらの自然キーの意味の文脈内であれば、欠番を説明しやすくなります。

疑似キー列とは別の属性列と疑似キー列の両方を、共に自然な識別子として使用できます。欠番が読み手に不安を与える場合、レポートには疑似キー列を表示しないようにしましょう。

> 疑似キーは、行を一意に識別するためにあります。行番号と混同しないようにしましょう。

22章
シー・ノー・エビル（臭いものに蓋）

すべての証拠を見つける前に理論を組み立てるのは、大きな誤ちだ。
——シャーロック・ホームズ

「おたくの製品に、またバグを見つけたぞ」と電話の相手が言いました。

1990年代、私はデータベース製品のテクニカルサポートエンジニアをしていました。電話の主は、私たちテクニカルサポートの間でいつも話題になっていたユーザーでした。彼の報告のほとんどは単純なミスが原因であり、私たちの製品のバグが原因ではなかったからです。

「おはようございます。ミスター・デービス。もちろん、弊社の製品に何か問題があれば修正いたします」私は答えました。「何が起きたかご説明いただけますでしょうか」

「データベースにクエリを実行しているのだが、何度実行しても何も返ってこない」デービス氏はきっぱりと言いました。「だが、データベースにデータがあることはわかっている。テストスクリプトでそれを検証できる」

「クエリには問題はないでしょうか？ APIはエラーを返しませんでしたか？」

「なぜ、私がAPI関数の戻り値を調べなきゃいけない？ 関数は私のSQLクエリを実行できるべきだ。エラーを返すのなら、それは製品にバグがあるということだ。製品にバグがなければ、エラーにはならないはずだろう。私に君のところの製品のバグを修正する義務などない」

私はこの一方的な主張にあきれながらも、気を取り直して事実を確かめようとしました。「わかりました。では、テストをしてみましょう。デービス様のコードから、SQLクエリを「正確に」コピーしてクエリツールに貼り付け、実行してみてください。何が表示されましたか？」そう言って、彼の回答を待ちました。

「SELCET文の構文エラー」しばらく間をおいた後、デービス氏は「この問題を解決してくれ」と言って、ガチャンと電話を切りました。

デービス氏は、とある航空交通管制会社の唯一のソフトウェア開発者で、国際便のフライトデータを記録するソフトウェアを開発していました。彼からは、毎週のようにこの手の電話がかかってきたものです。

22.1 目的：簡潔なコードを書く

開発者は誰でも、「エレガントな」コードを書きたいと考えています。すなわち、少ないコードで、「クー

ルな」仕事をしたいと望んでいるのです。仕事がクールであるということは、コードの量が少ないということであり、よりエレガントであるというわけです。仕事がクールではないのなら、それはエレガントさを改善できる余地があることになります。同じ仕事を少ないコードで実行することによって、少なくともコードのクールさを高めることができるはずだと考えるのです。

これは表層的な理由にすぎませんが、簡潔なコードには、より合理的ないくつかの理由があります。

- より短い時間でアプリケーションのコーディングを行えます。
- テスト、文書化、ピアレビューの対象となるコードの量が減ります。
- コードが少ないので、バグが混入する可能性も少なくなります。

このような理由があるため、開発者はほとんど本能的に、不要なコードをできるだけ削除しようとします（特に、それがクールではないと感じられる場合に顕著です）。

22.2 アンチパターン：肝心な部分を見逃す

開発者が「**シー・ノー・エビル（臭いものに蓋）**」アンチパターンに陥るときには、2つのパターンがあります。1つ目は、データベースAPIの戻り値を無視すること。もう1つは、アプリケーションコード内に点在するSQLしか読まないことです。両方のケースで、開発者は簡単に入手できる情報を見逃してしまうことになります。

22.2.1 診断せずに判断する

See-No-Evil/anti/no-check.php

```
    <?php
① $pdo = new PDO("mysql:dbname=test;host=db.example.com",
      "dbuser", "dbpassword");
  $sql = "SELECT bug_id, summary, date_reported FROM Bugs
      WHERE assigned_to = ? AND status = ?";
② $stmt = $pdo->prepare($sql);
③ $stmt->execute(array(1, "OPEN"));
④ $bug = $stmt->fetch();
```

このコードは一見簡潔ですが、関数の戻り値やステータスコードが問題を伝えてくれる箇所がいくつかあります。しかし、戻り値を無視する限り、その問題に気づくことはできません。

おそらく、データベースAPIがエラーを返すケースとして最も多いのは、データベース接続時です（例：①）。データベース名やサーバーのホスト名を誤って入力したり、ユーザー名やパスワードが間違っていたり、データベースサーバーのネットワークに障害があることなどが考えられます。PDO（PHPのDB拡張ライブラリ）コネクションのインスタンス化を行う際に問題があると例外が送出され、前述したサンプルスクリプトは終了してしまいます。

タイプミス、括弧の不一致、列名のスペルミスなどが原因の単純な構文エラーがある場合、②のprepareメソッド呼び出し時にfalseが返されます。この場合、③で$stmtのexecuteメソッドを呼ぶときに致命的エラーが発生します。PHPのfalseは、オブジェクトではないからです。

```
PHP Fatal error: Call to a member function execute() on a non-object
```

executeメソッドの呼び出しも失敗する可能性があります。例えば、SQLステートメントが制約に違反していたり、アクセス権限に違反している場合などです。executeメソッドも、エラー発生時にはfalseを返します。

④のfetchメソッドの呼び出しも、（その時点での）データベースへの接続失敗などのエラーが発生した場合にfalseを返します。

冒頭エピソードのデービス氏のような姿勢の開発者は珍しくはありません。彼らは、戻り値や例外のチェックのために自分のコードに手を加える必要はないと考える傾向があります。起こるはずがないと考えているのです。また、例外処理のために追加すべきコードはよく似た繰り返し作業が多く、アプリケーションを汚く、読みにくくするとも考えます。クールではないというわけです。

しかし、ユーザーが目にするのはアウトプットであり、コードではありません。図22-1に示すように、未処理の致命的エラーが生じるとき、ユーザーには真っ白なページや、理解不能な例外メッセージが表示されます。いくらアプリケーションコードが整然かつ簡潔に記述されていても、言い訳にはなりません。

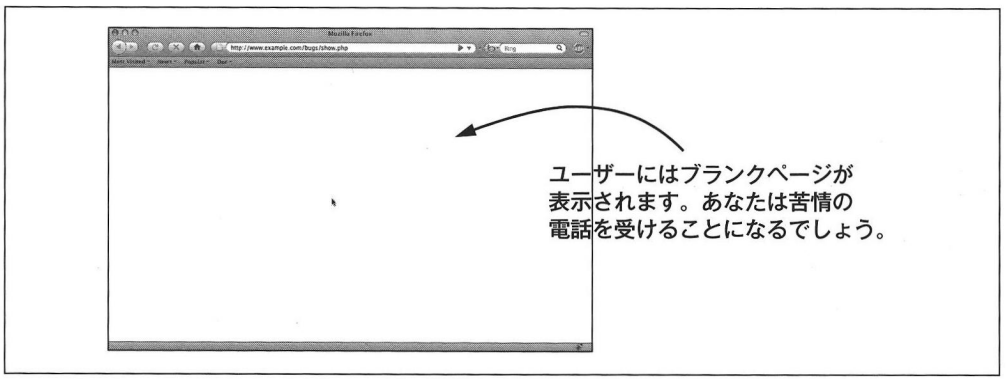


図22-1　PHPで致命的エラーが生じると、ブランクページが表示される。

22.2.2　見逃しがちなコード

「シー・ノー・エビル（臭いものに蓋）」アンチパターンの原因となる悪習は、SQL文字列を組み立てるアプリケーションコードの側からデバッグを開始することです。これは簡単ではありません。SQL文字列はアプリケーションロジックや文字列連結、アプリケーション変数の中身などを用いて構築されるため、最終的な構築結果を思い浮かべるのは難しいのです。

このようなデバッグのやりかたは、見本の写真を見ないでジグソーパズルを完成させようとするようなものです。

シンプルな例を用いて、私が開発者から良く受ける質問を見てみましょう。以下のコードは、バグ全件ではなく特定のバグを取得する必要がある場合、WHERE句を連結して条件付きのクエリを構築します。

See-No-Evil/anti/white-space.php

```
<?php
$sql = "SELECT * FROM Bugs";
if ($bug_id) {
    $sql .= "WHERE bug_id = " . intval($bug_id);
}
$stmt = $pdo->prepare($sql);
```

このクエリがエラーになる原因は何でしょうか。答えは、連結後の$sql文字列を見れば一目瞭然です。

See-No-Evil/anti/white-space.sql

```
SELECT * FROM BugsWHERE bug_id = 1234
```

そうです、Bugsと WHEREの間にスペースがありません。このクエリは BugsWHEREという名前のテーブルと、それに続く無効なSQL式として解釈され、クエリが構文エラーになっていたのです。このコードは、文字列の間にスペースを入れずに連結していたのです。

構築されたSQLそのものではなくSQLを構築するコードを調べることによって、デバッグの際に驚くほど多くの時間とエネルギーを浪費してしまうのです。

22.3 アンチパターンの見つけ方

「書かれていない」コードを見つけるのは難しいと思うかもしれません。しかし、最新のIDE（統合開発環境）製品の多くは、関数の戻り値を無視していたり、関数呼び出しの際にチェック例外の処理を怠っている場所などを強調してくれます[†1]。また次のような発言も、「シー・ノー・エビル（臭いものに蓋）」アンチパターンの兆候を示しています。

「データベースにクエリを発行した後で、プログラムがクラッシュする」

よくクラッシュが発生するのが、クエリが失敗したにもかかわらず、その結果が不正な方法で使用される場面です（例：非オブジェクトに対するメソッド呼び出しや、ヌルポインターの参照など）。

「SQLエラーの特定を手伝ってほしい。これがコードだ」

まず、SQLを構築するコードではなく、実際に構築されたSQLを調べることから始めましょう。

†1 チェック例外は関数のシグネチャの一部として宣言されます。その関数がどのような例外を送出する可能性があるかが一目でわかります。

「エラー処理でコードをゴチャゴチャさせたくないな」

一部のコンピューター科学者の調査によると、堅牢なアプリケーションのコードは、最大50パーセントがエラー処理のコードであるという結果が出ています。この割合は多いと感じるかもしれませんが、エラー処理に必要なエラーの検出、分類、レポート、回復などのいくつものステップを考えてみれば納得できるはずです。これらエラー処理のすべては、どのようなソフトウェアにおいても重要です。

22.4 アンチパターンを用いてもよい場合

エラーに対してまったく何もする必要がないときは、エラーチェックを省略できます。例えば、データベースコネクションのclose関数がステータスを返したものの、アプリケーション自身が終了の途中ならば、接続のためのリソースもクリーンアップされると考えてもよいでしょう。

また、オブジェクト指向言語の例外のしくみは、処理の責任のない例外を通過させることができます。例外処理の責任があるのは呼び出しを行ったコード側であると確信できる場合には、発生した例外をキャッチせずに呼び出し側まで通過させることができるのです。

22.5 解決策：エラーから優雅に回復する

ダンスの嗜みがある人は、ダンスにはミスステップが避けられないことを知っています。そのまま優雅に踊り続ける秘訣は、いかにミスから回復するかです。ミスの原因に気づくためのチャンスを自分に与えるのです。そうすれば、素早くスムーズに動いて、他の誰かに気づかれる前にリズムを取り戻せます。

22.5.1 リズムを維持する

データベースAPI呼び出しの**戻り値と例外のチェック**は、ステップにミスがないことを保証するための最善策です。以下に、エラーを起こす可能性があるメソッドの呼び出し後にステータスをチェックするコードの例を示します。

See-No-Evil/soln/check.php

```
    <?php
    try {
        $pdo = new PDO("mysql:dbname=test;host=localhost",
            "dbuser", "dbpassword");
①   } catch (PDOException $e) {
        report_error($e->getMessage());
        return;
    }

    $sql = "SELECT bug_id, summary, date_reported FROM Bugs
        WHERE assigned_to = ? AND status = ?";

②   if (($stmt = $pdo->prepare($sql)) === false) {
        $error = $pdo->errorInfo();
        report_error($error[2]);
```

```
        return;
    }

③  if ($stmt->execute(array(1, "OPEN")) === false) {
        $error = $stmt->errorInfo();
        report_error($error[2]);
        return;
    }

④  if (($bug = $stmt->fetch()) === false) {
        $error = $stmt->errorInfo();
        report_error($error[2]);
        return;
    }
```

①のコードは、データベース接続失敗時に送出される例外をキャッチします。また、このサンプルでは、PDOのコンストラクタ以外の関数は、問題があるときにfalseを返します。②、③、④のチェックで問題が検出できれば、データベースコネクションオブジェクトやステートメントオブジェクトから多くの情報を得ることができるでしょう。

22.5.2 ステップをたどり直す

デバッグには、SQLクエリを構築するコードではなく、**実際に構築されたSQLクエリを使用する**ことも重要です。綴り間違いや括弧の不一致など、コードではわかりにくく不可解になりがちな、多くの単純なミスを簡単に見つけられます。

- データベースAPIの引数に渡すその場で直接SQLを組み立てるのではなく、まず一時変数を使ってSQLクエリを構築してからAPIに渡しましょう。そうすることで、SQLの入った変数を使用前に調べられるようになります。
- アプリケーションとは別の出力先にSQLを出力するようにしましょう。例えば、ログファイル、IDEのデバッグコンソール、デバッグ情報を表示するブラウザ拡張[†2]などです。
- ウェブアプリケーション出力のHTMLコメント内にSQLクエリを表示しないようにしましょう。ユーザーは誰でもページのソースを見ることができます。SQLクエリを読んだ攻撃者に、データベースの構造について多くの情報を与えてしまいます。

SQLの構築と実行を透過的に行うオブジェクトリレーショナルマッピング（ORM）フレームワークを用いると、デバッグ作業が複雑になる場合があります。SQLクエリの中身にアクセスできなければ、どのようにデバッグすればよいのでしょうか。一部のORMフレームワークは、生成したSQLをログに出力することで、この問題を解決しています。

†2 Firebug（http://getfirebug.com/）など。

また、ほとんどのデータベース製品は、アプリケーション側のコードではなく、データベースサーバ自身のロギングメカニズムを提供しています。アプリケーションでSQLログを取得できない場合は、データベースサーバー側で実行時にクエリを監視できます。

> コードのトラブルシューティングは、それだけで十分に大変な作業です。闇雲に進めても、作業を遅らせるだけです。

23章 ディプロマティック・イミュニティ（外交特権）

人間は変化を嫌います。誰もが、「これまでそうしてきたから」と言って方法を変えません。
私はこうした考えに抗いたいのです。反時計周りの壁時計を使っているのもそのためです。
——グレース・ホッパー（アメリカの計算機科学者。プログラミング言語 COBOL の母）

若い頃に経験した1つの仕事が、ソフトウェアエンジニアリングのベストプラクティスを用いることの重要性を教えてくれました。それは、ある悲しい事故の後に任された、重要なデータベースアプリケーションの仕事でした。

私は、ヒューレットパッカードでの請負契約の面接を受けました。仕事の内容はUNIX上で動作するアプリケーションの開発と保守です。開発言語はCで、HP ALLBASE/SQLを使用していました。面接官のマネージャーやスタッフは、悲しそうな表情を浮かべながら、今回開発者を募集したのは、これまでそのアプリケーションを担当していたプログラマーが、交通事故で他界したからだと教えてくれました。その部門では他にUNIXを扱える人がおらず、アプリケーションを理解する人もいませんでした。

さっそく仕事を開始しましたが、すぐに重大なことに気づきました。前任者は、アプリケーションについての文書やテストを何も残していなかったのです。バージョン管理システムが使われていないだけでなく、ソースコードにはコメントさえありませんでした。すべてのコードは1つのディレクトリに格納されていました。そこには、稼働中のシステムコードも、開発途中のコードも、既に使われなくなっていたコードもすべて含まれていたのです。

ベストプラクティスに従わず、安易な方法を選んだことで、このプロジェクトは大きな**技術的負債**（Technical Debt）を抱えていました[†1]。技術的負債はプロジェクトのリスクを高め、リファクタリング、テスト、文書化などの追加作業という大きな代償を支払わなくてなりませんでした。

特に複雑でも大規模でもないアプリケーションであったにもかかわらず、このアプリケーションのコード整理と文書化に、私は半年を費やしました。アプリケーション利用者のサポートと継続的な開発にも、多くの時間を必要としたためです。

もっと短期間でプロジェクトを進めたかったのですが、当然ながら、前任者の手を借りるわけにはいきませんでした。この経験は、技術的負債が手に負えなくなったとき、いかに大きな影響が生じるかを雄弁に物

†1　技術的負債という比喩は、1992年のOOPSLA（Object-Oriented Programming, Systems, Languages & Applications）での事例報告の際にウォード・カニンガムが生みだした比喩です。（http://c2.com/doc/oopsla92.html）

語っていました。

23.1 目的：ベストプラクティスを採用する

プロの開発者は、以下のようなソフトウェアエンジニアリングのベストプラクティスに努めて従おうとします。

- Subversion や Git などのツールを用いて、ソースコードのバージョン管理を行う。
- ユニットテストや機能テストを自動化し、実行する。
- ドキュメント、仕様書、コードコメントを書き、アプリケーションの要件や実装戦略を記録する。

ベストプラクティスの実践に投じた時間は、極めて大きな見返りをもたらします。多くの不要な仕事や繰り返し作業を減らしてくれるからです。経験ある開発者は、このベストプラクティスを怠るとプロジェクトが失敗に向かって進み出すことを知っています。

23.2 アンチパターン：SQL を特別扱いする

アプリケーションコードの開発ではベストプラクティスを受け入れる開発者であっても、データベースコードではこれらの慣行が免除されると考える傾向があります。「アプリケーション開発のルールはデータベース開発には当てはまらない」――このような、データベースには特権的な何かがあるという考えを、私は「**ディプロマティック・イミュニティ（外交特権）**」アンチパターンと名付けました。

なぜこのように考えるのでしょうか。考えられる理由を挙げてみます。

- 一部の企業では、ソフトウェアエンジニアの役割とデータベース管理者（DataBase Administrator：DBA）の役割がはっきりと区別されています。データベース管理者はいくつかの開発チームを掛け持ちすることが多いため、どこかのチームのフルタイムメンバーではないと見なされる傾向があります。データベース管理者はどのチームでも「お客さん」のように扱われるため、他のメンバーと同じようなソフトウェアエンジニアの責任に従わなくてもよいという暗黙の了解が生まれるのです。
- リレーショナルデータベースで用いられる SQL 言語は、従来型のプログラミング言語とは性質が異なります。SQL ステートメントがアプリケーションコード内で特殊な言語として扱われる様子も、データベース管理者が「お客さん」のように扱われる状況を象徴しているかのようです。
- アプリケーションコードを記述する言語では、高度な IDE ツールが一般的に用いられ、ソースコードの編集、テスティング、バージョン管理などを素早く容易に行えます。しかしデータベース開発向けのツールにはこのような高度な機能はなく、あったとしても広く普及してはいないのが現状です。アプリケーションコードの開発では簡単にベストプラクティスを適用できても、SQL では勝手が違う面があるため、なかなかうまく適用できません。
- インフラ部門では、データベースに関する知識や運用権限が、1 人のデータベース管理者に集中してしまう傾向があります。なぜなら、データベース管理者は、データベースサーバーへの唯一のア

クセス権を持っているからです。このため、データベース管理者本人が、ナレッジベースやバージョン管理システムの役割を果たすようになってしまいます。

データベースはアプリケーションの屋台骨であり、その品質が重要です。高品質のアプリケーションコードを開発する方法を知っていたとしても、そのアプリケーションが依存するデータベースは、プロジェクトのニーズを満たしておらず、誰も理解していない代物の可能性もあるのです。せっかく良いアプリケーションを開発しても、データベースのためにプロジェクトは失敗してしまうかもしれません。

23.3 アンチパターンの見つけ方

「何かを行っていないこと」の証拠を見つけるのは難しいと思われるかもしれませんが、必ずしもそうではありません。以下の発言は、ベストプラクティスの実践を怠っている兆候を示しています。

「これからは、新しいエンジニアリングプロセスを採用するつもりだ。それは言わば、『軽量な』エンジニアリングプロセスだ」

ここでの「軽量な」とは、エンジニアリングプロセスに必要なタスクを、チームが省略しようとしていることを意味しています。タスクの一部には、省略が妥当なものもあるでしょう。しかしこの言葉は、重要なベストプラクティスに従わないことの、婉曲的な表現であるとも考えられます。

「データベース管理者には、バージョン管理システムのトレーニングに参加してもらわなくてもいいだろう。トレーニングを受けても、データベース管理者はこのシステムを使わないだろうから」

トレーニングに参加しない開発メンバーが、そのツールを使うようになることは**まずありません。**

「データベースのテーブルや列が使われているかどうかをどうやって調べたらいい？ 目的がわからないテーブルや列は、もう使われていないなら削除したいんだ」

プロジェクトのデータベース設計ドキュメントが使われていません。ドキュメントは古くなっているか、アクセスできないか、そもそも存在しないと考えられます。テーブルや列の目的がわからないとしても、他の誰かにとっては重要であるかもしれないため、削除はできません。

「2つのデータベーススキーマを比較し、差分をレポートし、一致させるために片方を修正するスクリプトを生成するツールはある？」

データベーススキーマ変更の適用プロセスに従っていなければ、スキーマは同期しなくなります。同期していないスキーマを直す仕事は、複雑なタスクになるでしょう。

23.4 アンチパターンを用いてもよい場合

私は、何度も使うコードに対しては、ドキュメントやテストを書き、バージョン管理などのベストプラクティスを適用しています。しかし、その場限りのコードもよく書いています。API の使い方を思い出すための1回限りのテストや、ユーザーの質問に答えるための SQL クエリなどです。

自分の書いたコードが、一時的なものかどうかを判断するよい指針があります。使い終わったコードをすぐに削除するのです。もし気が進まないのなら、おそらく保存しておく価値があります。そのコードをバー

ジョン管理システムに保存し、目的と使い方について簡単な記述をしておくことを忘れないようにしましょう。

23.5 解決策：包括的に品質問題に取り組む

ソフトウェア開発者の多くにとって、品質とは単にテストを行うことを意味します。しかし、これは**品質管理**（Quality Control：QC）に過ぎません。大きなストーリーの一部でしかないのです。ソフトウェアエンジニアリングのライフサイクルは**品質保証**（Quality Assurance：QA）を伴います。品質保証は、3つの部分から成ります。

1. プロジェクト要件の明確な定義・文書化
2. 要件に対する解決策の設計・構築
3. 解決策が要件を満たしていることの確認・テスト

適切な品質保証のためには、これらの3つをすべて行う必要があります。ただし、ソフトウェア開発プロセスによっては、必ずしもこの順番に従う必要はありません。

データベース開発における品質保証は、**文書化**、**バージョン管理**、**テスティング**のベストプラクティスに従うことで達成できます。

23.5.1 文書化

コードが自分で文書を書いてくれることはありません。たしかに、熟練のプログラマーなら、注意深くコードを分析し、実際に調査を行うことで、たいていのコードの意味を解読できます。しかしこれは大きな労力を伴います[†2]。またコードは、不足している機能や未解決の問題を私たちに教えてくれるわけでもありません。

データベースの要件と実装は、アプリケーションコードと同じように文書化すべきです。データベースのオリジナルの設計者でも、誰かが設計したデータベースを引き継いだ場合であっても、以下のチェックリストを使ってデータベースを文書化しましょう。

ER図

データベースドキュメントのなかで最も重要なものが、テーブルとその関連（リレーションシップ）を表すER図（Entity-Relationship diagram）です。この本のいくつかの章でも、簡単なER図を用いています。より詳細なER図には、列、キー、インデックスなどのデータベースオブジェクトを記すための表記法があります。

図表作成用のソフトウェアパッケージ製品のなかには、ER図の表記法要素を含むものがあります。また、SQLスクリプトや稼働中のデータベースをリバースエンジニアリングして、ER図を生成するツールもあります。

注意しなければならないのは、非常に多くのテーブルを持つ複雑なデータベースを、1つの図で表すの

†2 コードが文章のように簡単に読めるものであるのなら、なぜ私たちはそれをコード（暗号）と呼ぶのでしょうか。

は現実的ではないことです。このような場合は、複数のER図に分割します。読み手を圧倒してしまわないよう、各ER図をテーブルの適切なサブグループに分割しましょう。

テーブル、列、ビュー

データベースを説明する文書も必要です。ER図は、テーブルや列などのオブジェクトの目的や利用法の記述には適してしないからです。

テーブルには、テーブルが表現しているエンティティの分類についての説明が必要です。例えば、Bugs、Products、Accountsなどは、名前からそのエンティティを特定しやすいと言えます。しかし、BugStatusのような**参照テーブル**や、BugsProductsのような**交差テーブル**、Commentsのような**従属テーブル**には、説明が必要です。また、各テーブルで想定している行数や、テーブルに対して実行されるクエリ、テーブルに構築するインデックスについても記述します。

列にはそれぞれ名前とデータ型があります。しかし、それだけでは、読み手には列の値の意味がわかりません。その列にとって妥当な値は何か（データ型の全範囲が妥当であることはめったにありません）、定量的な値を格納する列で用いる単位や、NULLを許容するかしないか、**一意性制約**があるか（ある場合その理由）などを記述しましょう。

ビューには、1つ以上のテーブルに対して頻繁に使われるクエリが格納されています。そのビューの目的、ビューの使用が想定されるアプリケーションやユーザー、ビューにテーブル間の複雑な関連（リレーションシップ）を要約する意図があるか、特権アクセスのないユーザーが特権的なテーブルの行や列のサブセットを参照できるようにするか、ビューは更新可能か、などを記述します。

関連（リレーションシップ）

参照整合性制約はテーブル間の依存関係を実現しますが、これは制約のモデリングについての開発者の意図をすべて表しているとは限りません。例えば、Bugs.reported_byは NULLを許容する列ではありませんが、Bugs.assigned_toは NULLを許容します。これは、バグは割り当てられる前に、修正可能であることを意味しているのでしょうか。もしそうではなく、バグの割当てが必須なのであれば、その根拠となるビジネスルールは何でしょうか。

明示的に制約が定義されていないものの、暗黙的な関連がある場合もあります。ドキュメントがなければ、暗黙的な関連が存在している箇所を知るのは困難になります。

トリガー

トリガーの仕事には、データのバリデーションや変換、データベース変更のロギングなどがあります。トリガーに実装しているビジネスルールなどを記述しましょう。

ストアドプロシージャ

APIの文章化と同じように、**ストアドプロシージャ**を文書化しましょう。どのような問題を解決するためのものか、データに対する変更を行なうかどうか、入出力パラメータのデータ型と意味、パフォーマンスボトルネックを回避するために特定のクエリを置換することを意図しているか、特権テーブルに特権アクセスのないユーザーのアクセスを許可することを意図しているか、などを記述しましょう。

SQL セキュリティ

アプリケーションから使用するデータベースユーザーの定義、各ユーザーのアクセス権、提供するロー

ルと対象ユーザー、バックアップやレポートなどの特定タスクの実行を許可されているユーザーの有無、システムレベルのセキュリティ対策（クライアントはSSL経由でRDBMSサーバーにアクセスしなければならないなど）、不正アクセスの検出・防御のための処置、SQLインジェクション脆弱性対策のための徹底したコードレビューの有無などを記述します。

データベースインフラストラクチャ

この情報は主にインフラ技術者とデータベース管理者によって使われますが、開発者もその一部を知っておく必要があります。データベース製品の種類とバージョン、データベースサーバーのホスト名、データベースサーバの冗長化（レプリケーション、クラスタリング、プロキシなど）、データベースサーバーで使用するネットワーク構成とポート番号、クライアントアプリケーションが使用すべき接続オプション、データベースユーザーのパスワード、データベースのバックアップポリシーなどを記述します。

オブジェクトリレーショナルマッピング（ORM）

プロジェクトでは、オブジェクトリレーショナルマッピング（ORM）ライブラリを使ったクラス群を通して、アプリケーションコードにデータベース操作ロジックを実装するかもしれません。実装されるビジネスルールは何かを記述しましょう。またデータの妥当性確認、データ変換、ロギング、キャッシュの扱い、プロファイルの取り方などについても記述しましょう。

一般的に、開発者はエンジニアリング文書のメンテナンスを好みません。文書を書くことは手間がかかり、最新状態に維持することも簡単ではありません。また、文書を読む人がごく一部しかいない場合もあり、書く意欲を削がれます。しかし、百戦錬磨のプログラマーでさえ、ソフトウェアの他の部分は文書化しない場合でも、データベースを文書化する必要があることを知っています[†3]。

23.5.2 バージョン管理

データベースサーバーが故障によって完全に駄目になってしまった場合、どのようにデータベースを再構築すればよいでしょうか。データベースの複雑な設計変更を記録するための最善策は何でしょうか。変更を取りやめた場合、どのように元に戻せばよいでしょうか。

私たちは、バージョン管理システムを用いて、アプリケーションコードの管理や、ソフトウェア開発における複雑な履歴管理の問題を解決する方法を知っています。バージョン管理下にあるプロジェクトには、直近のリリースに大きな障害が生じた場合に、プロジェクトを再構築・再配備するために必要な**すべて**が格納されています。また、バージョン管理システムは、変更履歴管理や増分バックアップの機能を提供し、プロジェクトに発生した変更の巻き戻しも可能にします。

データベースを扱うコードに対してもバージョン管理を用いることで、開発において同様のメリットを得られます。

以下のようなデータベース開発関連のファイルを、バージョン管理システムの管理下に入れましょう。

---

†3 例えば、Jeff AtwoodとJoel Spolskyは、StackOverflowポッドキャストNo.80にて、アプリケーションコードの文書化の価値は低いが、データベースの文書化には価値があると述べています。（http://blog.stackoverflow.com/2010/01/podcast-80/）

データ定義スクリプト

各種データベース製品が提供する、CREATE TABLEなどのステートメントでデータベースオブジェクトを定義するSQLスクリプト群です。

トリガーとプロシージャ

多くのプロジェクトでは、データベースに格納したトリガーやプロシージャでアプリケーションコードを補完しています。アプリケーションは、これらのトリガーやプロシージャなしではおそらく機能しないので、プロジェクトコードの一部だと言えるでしょう。

ブートストラップデータ

ユーザーが使い始める前に、データベースの初期状態として参照テーブルにデータを格納しておくことがあります。プロジェクトのソースからデータベースを復元する必要がある場合には、このような**ブートストラップデータ**を管理しておくべきです。ブートストラップデータは、**シードデータ**（seed data）と呼ばれることもあります。

ER図とドキュメント

これらのファイルはコードではありませんが、コードと密接に結びついているものであり、データベースの要件、実装、アプリケーションとの統合について記述されています。プロジェクトが進むにつれ、データベースとアプリケーションの両方に変更が必要になった場合は、これらのファイルを最新にしておくべきです。文書が最新の設計を記述していることを確認しましょう。

データベース管理スクリプト

プロジェクトの多くには、アプリケーションの外部で実行する、データ処理のジョブがあります。ジョブには、インポート／エクスポート、同期、レポート、バックアップ、バリデーション、テスティングなどのタスクが含まれます。タスクは従来型のアプリケーション開発言語ではなく、SQLスクリプトとして書かれる場合があります。

データベース側コードのファイルが、そのデータベースを使うアプリケーションコードと対応付いていることを確認しましょう。バージョン管理システムを使うメリットの1つは、バージョン管理下にある特定のバージョン、日付、マイルストーンを用いてプロジェクトをチェックアウトすると、チェックアウトされたファイル群がきちんと動作することが期待できる状態になっていることです。このため、**アプリケーションコードとデータベースコードは、同じバージョン管理リポジトリを用いる**ようにしましょう。

スキーマ進化ツール

コードはバージョン管理システムの対象になっていても、データベースはそうではありません。Ruby on Railsは、バージョン管理下のデータベースインスタンスに対する変更を管理する**マイグレーション**（migrations）と呼ばれる手法を普及させました。マイグレーションによる変更例を簡単に見てみましょう。

データベースの変更を行うためのRailsの抽象クラス（ActiveRecord::Migration）を継承して、デー

タベース定義をワンステップずつ変更するコード（upメソッド）を作成します。また、変更を元に戻す機能（downメソッド）も記述します。

```
class AddHoursToBugs < ActiveRecord::Migration
  def self.up
    add_column :bugs, :hours, :decimal
  end

  def self.down
    remove_column :bugs, :hours
  end
end
```

マイグレーションを実行するRailsツールは、リビジョン管理用のテーブルを自動的に作成し、その管理テーブルに現在のデータベースに適用する1つ以上のリビジョン識別子を記録します。Rails 2.1では、このシステムをさらに柔軟にする変更が導入され、さらに後続のバージョンでも、マイグレーションの方法が変更されることが見込まれています。

データベースに対するスキーマ変更のたびに新しいマイグレーションスクリプトを作成することで、データベーススキーマをワンステップでアップグレード/ダウングレードするマイグレーションスクリプト群を蓄積していきます。もしデータベーススキーマ定義をバージョン5に変更したい場合は、マイグレーションツールに引数を指定します[†4]。

```
$ rake db:migrate VERSION=5
```

マイグレーションについての詳細は、『RailsによるアジャイルWebアプリケーション開発 第4版』[RTH11]やRailsGuides [†5]で読むことができます。

PHPのDoctrine [†6]、Pythonの Django [†7]、Microsoft の ASP.NET [†8] などウェブ開発フレームワークの多くも、フレームワークの一部またはコミュニティプロジェクトとして、Railsのマイグレーションと似た機能を提供しています。

マイグレーションによって面倒な作業の多くが自動化され、バージョン管理下にあるプロジェクトの対象リビジョンの構造と、データベースインスタンスを同期させることができます。しかし、まだ完ぺきではありません。マイグレーションは、シンプルなスキーマ変更のごく一部を管理し、従来型のバージョン管理システムの上にデータベーススキーマの版管理システムを実装しているものだといえるでしょう。

23.5.3 テスティング

品質保証の最終パートは、**品質管理**（Quality Control）です。すなわち、アプリケーションが設計通りに動作することの検証です。プロの開発者は、自動テストを記述してアプリケーションコードの振舞いを検証

†4 監訳注：原著が書かれた時はマイグレーションのリビジョン識別子が整数の連番でしたが、その後複数ブランチ等での並行開発を可能にするためにリビジョン識別子がタイムスタンプベースの文字列になりました。

†5 http://guides.rubyonrails.org/migrations.html

†6 http://www.doctrine-project.org/

†7 https://www.djangoproject.com/

†8 http://www.asp.net/

する技法に精通しています。テスティングの重要な原則の1つは、**独立**（isolation）です。テストごとにシステムの1部分のみを検証することで、欠陥が存在する場合に、可能な限り正確に、欠陥の箇所を絞り込むことができます。

アイソレーションテスト（isolation testing）は、データベースにも適用できます。データベースの構造と振舞いの妥当性確認を、アプリケーションコードとは独立させて行うことができます[†9]。

以下に、PHPUnitテストフレームワークを用いたユニットテストの例を示します[†10]。

Diplomatic_immunity/DatabaseTest.php

```
<?php

class DatabaseTest extends \PHPUnit_Framework_TestCase
{
    protected $pdo;

    public function setUp()
    {
        $this->pdo = new PDO("mysql:dbname=bugs", "testuser", "xxxxxx");
    }

    public function testTableBugsExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(*) FROM Bugs");
        $err = $this->pdo->errorInfo();
        $this->assertInternalType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt));
    }

    public function testTableBugsColumnBugIdExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(bug_id) FROM Bugs");
        $err = $this->pdo->errorInfo();
        $this->assertInternalType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt));
    }
}
```

データベースの妥当性を検証するテストのためのチェックリストを以下に示します。

---

†9 監訳注：「アイソレーションテスト（isolation testing）とは、周辺のコンポーネントとは独立に、個々のコンポーネントをテストすること。必要に応じて、スタブやドライバで、周辺コンポーネントをシミュレートする」『ソフトウェアテスト標準用語集（日本語版）Version 2.1.J01』[JST12]より。

†10 実は、正確にはデータベースの機能に対するテストは、ユニットテストではありません。しかし、PHPUnitによって、テストの組織化と自動化を行うことはできます。http://www.phpunit.de/

テーブル、列、ビュー

データベースに存在すべきテーブルやビューなどが、実際に存在することをテストで確認しましょう。テーブル、ビュー、列の追加によってデータベースを拡張する場合は、テストを新規作成してオブジェクトが存在していることを確認しましょう。また、**否定テスト**（negative test）を作成して、プロジェクトの現在のリビジョンから削除したテーブルや列が、本当に存在していないことを確認できます[†11]。

制約

制約に対しても、否定テストを行うことができます。制約に違反してエラーになるはずのINSERT、UPDATE、DELETEステートメントをテストで実行して、実際にエラーが生じることを確認します。例えば、NOT NULL制約、一意性制約、外部キー制約などに違反するようなステートメントを実行します。ステートメントがエラーを返さなければ、制約は機能していません。これらの障害を初期段階で特定することで、多くのバグを予防できます。

トリガー

トリガーも制約を強制します。トリガーは、カスケード更新、値の変換、変更のロギングなどを実行できます。トリガーが発動するステートメントを実行し、次にトリガーが意図した処理を行ったかどうかを確認するためのクエリを実行して、これらのシナリオをテストしましょう。

ストアドプロシージャ

ストアドプロシージャのテストは、従来型のアプリケーションコードのユニットテストと似ています。ストアドプロシージャには入力パラメータがあり、有効な入力範囲外の値を渡そうとすると、エラーを送出します。プロシージャの本体内のロジックには、複数の分岐パスを書けます。プロシージャは、入力およびデータベース内のデータの条件に応じて、単一の値またはクエリ結果セットを返せます。また、プロシージャには、データベースを更新する**副作用**を持つものもあります。プロシージャのこれらのすべての側面をテストします。

ブートストラップデータ

空の状態であることが想定されるデータベースであっても、通常は参照テーブルなどに初期データが必要です。初期データの存在を確認するためのクエリを実行してテストを行います。

クエリ

アプリケーションコードは、SQLクエリと組み合わせて使用されます。構文と結果の確認のために、テスト環境でクエリを実行できます。テーブルやビューのテストと同じように、結果セットに適切な列名とデータ型が含まれていることを確認します。

オブジェクトリレーショナルマッピング（ORM）を使用したクラス

トリガーと同様、オブジェクトリレーショナルマッピング（ORM）を使用したクラスには、バリデーション、変換、モニタリングなどのロジックが含まれます。他のアプリケーションコードと同様、ORMでデータベースを抽象化したコードもテストすべきです。これらのクラスが、入力に対して予期された振舞いをすることや、無効な入力を拒絶することを確認しましょう。

---

†11 監訳注：「否定テスト（negative testing）とは、コンポーネントやシステムが、正しく機能しないことを証明するためのテスト。否定テストは、特定のテストアプローチやテスト設計技法ではなく、テスト担当者の意向を反映したもの。例えば、無効入力値や例外値を用いたテスト」『ソフトウェアテスト標準用語集（日本語版）Version 2.1.J01』[JST12]より。

テストのいずれかが失敗する場合、アプリケーションが間違ったデータベースインスタンスを使用している可能性もあります。正しいデータベースに接続していることを常にダブルチェックしましょう。間違ったインスタンスに接続していたためにテストが失敗するというのが、本当によくあるミスなのです。必要に応じて設定を修正し、再試行しましょう。接続は適切なものの、データベース設計が異なるために変更しなければならない場合は、**マイグレーションスクリプト**（本章のコラム「スキーマ進化ツール」を参照）を実行し、データベースインスタンスをアプリケーションで必要としている設計に同期させることができます。

23.5.4 複数のブランチを扱う

アプリケーションの開発では、複数の**ブランチ**のコードを扱う場合があります。同じ日に、複数のブランチに関わることもあるかもしれません。例えば、現在本番環境に配備されているブランチで緊急のバグを修正したすぐ後で、長期的に取り組んでいるメインブランチの開発を再開するといった具合です。

しかしアプリケーションが使用するデータベースそのものは、バージョン管理の対象外です。データベース製品によっては軽くて使いやすいものがあるとはいえ、データベースを作成して設定し、すぐに破棄するのは現実的ではありません。

理想は、開発・テスト・ステージング・デプロイを行うアプリケーションの各ブランチ向けにそれぞれデータベースインスタンスを構築することです。また、プロジェクトチームの各開発者にも、他のメンバーの仕事を妨げないよう、それぞれデータベースインスタンスが必要です。

データベース接続パラメータが設定できるようにアプリケーションを設計しましょう。これにより、どのブランチのアプリケーションを使用していても、コードを上書きせずに使用するデータベースを指定できます。

現在では、商用製品であれオープンソースであれ、あらゆるデータベース製品は、開発とテストのための無料のソリューションを提供しています。またVMware Workstation、Xen、VirtualBoxなどのプラットフォーム仮想化技術によって、わずかな費用でサーバー環境のクローンを作成できます。これらを用いれば、ソフトウェア開発者は、本番環境と同一の機能を持った環境を使って、開発とテストを行えるのです。

> アプリケーションと同じく、データベースに対しても、ソフトウェア開発のベストプラクティスを適用し、文書化、テスト、バージョン管理を行いましょう。

24章
マジックビーンズ（魔法の豆）

言い訳は太古の昔から存在する。それは、あらゆる人間にとっての問題解決の常套手段だ――
巧妙で、まことしやかで、そして、間違っている。
―― H.L. メンケン（アメリカのジャーナリスト）

「小さな機能を1つ追加するだけなのに、なぜこんなに時間がかかっているんだ？」上司が言いました。

あなたのチームは、上司からバグ管理アプリケーションへの機能追加を命じられていました。バグごとのコメント数を表示する機能です。チームが作業に着手してから、すでに4週間が経過しています。

会議室は重苦しい雰囲気につつまれています。他の開発メンバーは、誰も上司の質問に答えようとしません。あなたはリーダーとしての責任から、口を開きました。「実は、出だしでいくつかのミスがあったのです。一見、簡単なタスクに思えたのですが、しばらくして、アプリケーションの他のいくつかの画面にもコメント数を表示しなければならないことに気づきました」

「で、その画面の設計に4週間もかかったと言うのか？」上司が尋ねます。

「いいえ、これは単純なHTMLですし、私たちはコードとプレゼンテーションを分離できるフレームワークを使っています。しかし、画面にこの要素を加えるたびに、画面のバックエンド側からデータを取得するコードをまた書かなければなりません。つまり、各バックエンドクラスには、新たなテスト群が必要になります」

「テスティングフレームワークを使っているんだろう？」上司が尋ねました。「テストをいくつか追加するのに、そんなに時間がかかるものなのか？」

「テストは、コードほど単純には書けません」他のメンバーがためらいがちに答えました。「テストデータ作成のためのスクリプトも作成しました。テストごとにテスト用データベースのデータをリロードする必要もあります。また、フロントエンドもテストしなければならないのですが、その際には新機能と古いシナリオのすべての組み合わせに対するテストが必要なのです」

上司はよく理解できていないようでしたが、同僚はかまわず話を続けました。「要するに、いまフロントエンド向けのテストは600もあり、それぞれがブラウザエミュレータのインスタンスを立ち上げます。ですから、テストの実行に非常に時間がかかってしまうのです」同僚は肩をすくめて言いました。「でも、もうこのやり方で進めるしかありません」

上司が大きくため息をつきました。「すべて理解できたわけじゃないが、言いたいことはわかった。だが私が訊きたかったのは、シンプルな機能を追加するはずが、なぜそれほど複雑になってしまったのかということだ。君たちが使っているオブジェクト指向フレームワークは、簡単かつ素早く機能を追加できるんじゃなかったのか？」

なかなか鋭い問いです。

24.1 目的：MVCのM（モデル）を単純化する

ウェブアプリケーションフレームワークを使うと、迅速かつ簡単にコードを書いて、アプリケーションに機能を追加できます。ソフトウェアプロジェクトでの最大のコスト要因は開発工数です。工数を減らせば、開発コストも減らせます。ロバート・L・グラスは「ソフトウェア開発の80パーセントは知的作業であり、そのほとんどが創造的なものである。事務的な作業はわずかしかない」と述べています[†1]。

この知的作業を支援してくれるのが、**デザインパターン**などの語彙や規約です。“Singleton”、“Façade”、“Factory”などのパターン名を使うことによって、チームのメンバーに意図を伝えやすくなり、多くの時間を節約できます。

どんなアプリケーションでも、コードの大部分は重複しています。それらは実質的にボイラープレート（定型的なコード）です。フレームワークが提供する再利用可能なコンポーネントやコード生成ツールによって、コードの生産性を改善できます。新規のコードを大量に書かずとも、アプリケーションを開発できるのです。

モデル・ビュー・コントローラ（MVC）アーキテクチャは、デザインパターンとソフトウェアフレームワークの両方が関わります。図24-1に示すように、MVCはアプリケーションを3つの関心事に分ける技法です。

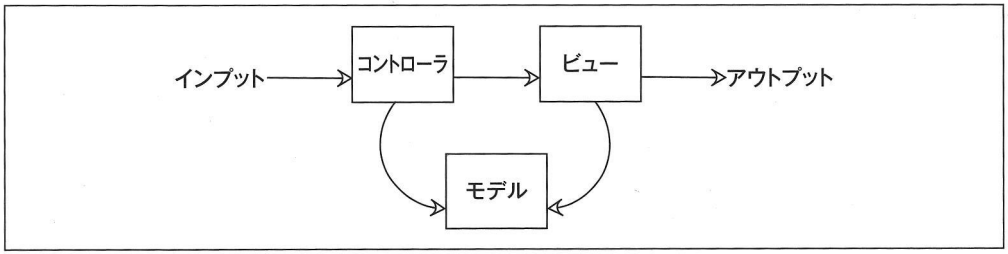


図24-1 モデル・ビュー・コントローラ

コントローラ

ユーザーの入力を受け入れ、入力に対してアプリケーションが何を返すべきかを定義し、関連するモデルに仕事を委譲し、結果をビューに送ります。

ビュー

ユーザーインタフェースに情報を表示します。

†1 『ソフトウェア開発55の真実と10のウソ』[Gla92]より抜粋。

モデル

他のすべてを担います。モデルはアプリケーションの中核であり、入力のバリデーション、ビジネスロジック、データベースとのやりとりなどが含まれます。

コントローラとビューが何を意味するかは明白ですが、モデルの目的は曖昧です。ソフトウェア開発者のコミュニティには、ソフトウェア設計の複雑さを低減するために、モデルの概念を整理し、一般化を行いたいという強い思いがあります。しかしそのあまりに、モデルを過度に単純化し、モデルは単なるデータアクセスオブジェクトに過ぎないと見なしてしまうことがあるのです。

24.2 アンチパターン：モデルがアクティブレコードそのもの

シンプルなアプリケーションでは、モデルに多くのカスタムロジックは不要です。モデルオブジェクトのフィールドを、テーブルの列と対応づけることは特に難しくありません。これは一種のオブジェクトリレーショナルマッピング（ORM）です。オブジェクトに必要な操作は、テーブルでの行の作成、行の読み込み、更新、削除です。これらは基本的な **CRUD 操作**[†2] と呼ばれるものです。

マーティン・ファウラーは、このようなマッピングをサポートするデザインパターンを「**アクティブレコード（Active Record）**」と名付けました[†3]。アクティブレコードは、データアクセスパターンの1つです。まず、データベースのテーブルやビューに対応するクラスを定義します。クラスメソッド findを呼び出すと、そのテーブルやビューの各行に対応したインスタンスが返されます。また、クラスのコンストラクタを用いて新しい行を作成できます。オブジェクトの saveメソッドを呼ぶことで、行の挿入や既存行の更新も行えます。

Magic-Beans/anti/doctrine.php

```
<?php
$bugsTable = Doctrine_Core::getTable('Bugs');
$bugsTable->find(1234);

$bug = new Bugs();
$bug->summary = " 保存時にクラッシュが発生 ";
$bug->save();
```

アクティブレコードパターンは、2004年に Ruby on Rails の登場によってウェブアプリケーションフレームワークで広く用いられるようになりました。現在では、ほとんどのウェブ開発フレームワークが、アクティブレコードパターンを事実上の**データアクセスオブジェクト（DAO）**設計として採用しています。もちろん、アクティブレコードそれ自体に問題があるわけではありません。アクティブレコードは、単一テーブルの各行に対するシンプルなインタフェースを提供してくれる、優れたパターンです。アンチパターンになってしまうのは、MVC アプリケーションのすべてのモデルクラスがアクティブレコードの基底クラスを

†2 監訳注：Create、Read（Retrieve とも）、Update、Delete の各頭文字から CRUD と呼ばれます。

†3 『エンタープライズ アプリケーションアーキテクチャパターン』[Fow03] の「アクティブレコード」の章を参照してください。

継承するという規約です。これは、「**黄金のハンマー（Golden Hammer）**」アンチパターン [BMMM98]の例だと言えます。道具がハンマーしかないときには、すべてが釘に見えるのです。

ソフトウェアの設計を単純化してくれる規約には、つい従ってみたくなります。柔軟性を犠牲にすれば、仕事は単純化できます。もともと柔軟性を求められていない仕事なのであれば、規約がより魅力的に見えてきます。

しかしこれは、お伽噺にすぎません。『ジャックと豆の木』では、ジャックは一夜明けたら豆が巨木になることを信じて眠りにつきます。しかし私たちは、ジャックのような魔法の豆を持ってはいません。では、「**マジックビーンズ（魔法の豆）**」アンチパターンがどのような結果をもたらすかを見ていきましょう。

24.2.1 アクティブレコードはモデルをデータベーススキーマに強く依存させてしまう

アクティブレコードはシンプルなパターンです。簡素なアクティブレコードクラスは、1つのテーブルまたはビューを表現するからです。各アクティブレコードオブジェクトのフィールドは、対応する1つのテーブルの列と一致します。16のテーブルがある場合、16のモデル（アクティブレコード基底クラスのサブクラス）を定義することになります。

これはつまり、データベースをリファクタリングして新たなデータ構造を表す場合には、モデルクラスだけでなく、そのモデルクラスを使うアプリケーションのコードも変更する必要があることを意味します。また、アプリケーションで新たな画面を処理するコントローラを加える場合、モデルとのやりとりを行うために、既存のコードによく似たコードをコントローラに書く必要が出てきます。

24.2.2 アクティブレコードはCRUD機能を公開してしまう

他に起こる可能性がある問題としては、モデルクラスを使用する他のプログラマーが、意図された用法を無視して、CRUD操作メソッドを通じて直接データを更新してしまうことが挙げられます。

例えば、あなたがassignUserメソッドをバグのモデルに追加したとしましょう。バグの修正担当エンジニアを変更する場合に、そのエンジニアに電子メールを送る必要があるからです。

Magic-Beans/anti/crud.php

```
<?php
class CustomBugs extends BaseBugs
{
  public function assignUser(Accounts $a)
  {
    $this->assigned_to = $a->account_id;
    $this->save();
    mail($a->email, "バグ担当に任命されました",
      "あなたはバグ #{$this->bug_id} の修正担当に任命されました。");
  }
}
```

しかし、バグアプリケーションに取り組んでいる別のプログラマーが、このメソッドを迂回し、電子メールを送信せずにバグ担当者を割り当てるコードを書いてしまいます。

Magic-Beans/anti/crud.php

```
$bugsTable = Doctrine_Core::getTable('Bugs');
$bug = $bugsTable->find(1234);
$bug->assigned_to = $user->account_id;
$bug->save();
```

あなたが手がけていた機能の要件は、バグ担当者が変更された場合に、常に電子メールを送信することでした。しかし、このステップは迂回されてしまいました。アクティブレコードの基底クラスから派生したモデルクラスが、基底クラスにあるCRUDメソッドを公開することは、はたして妥当だと言えるでしょうか。他のプログラマーがCRUDメソッドを不適切に使うことのないようにするには、どうすればよいでしょう。アクティブレコードの基底クラスのインタフェースを、モデルクラスから生成されるドキュメントや、プログラミングエディタのコード補完機能から除外するにはどうすればよいでしょうか。

24.2.3 アクティブレコードはドメインモデル貧血症をもたらす

先ほどの問題と密接に関連するのが、多くの場合、モデルが基本的なCRUDメソッド**以外**の振舞いを持たないという問題です。開発者の多くは、モデルがすべき仕事を表現する新たなメソッドを追加せずに、アクティブレコードの基底クラスを単に継承します。

モデルをシンプルなデータアクセスオブジェクトとして扱うと、モデルの外部でビジネスロジックのコーディングが必要になります。結果として複数のコントローラクラスにロジックが書かれ、モデルの振舞いの**凝集度**[†4]が低下します。マーティン・ファウラーはブログのなかで、このアンチパターンを「**ドメインモデル貧血症**（Anemic Domain Model）」と呼んでいます[†5]。例えば、Bugs、Accounts、Productsテーブルに対応するアクティブレコードクラスをそれぞれ作成しますが、アプリケーションタスクの多くでは、これら3つのテーブルすべてのデータが必要なのです。

バグ管理アプリケーションのサンプルコードを見てみましょう。バグ管理アプリケーションには、バグの割当て、バグ情報入力、バグ詳細表示、バグ検索のタスクが実装されています。シンプルなアクティブレコードインタフェースを提供するために、PHPのフレームワークDoctrineを用いています[†6]。また、MVCアーキテクチャを実現するためにZend Frameworkを使用しています。

Magic-Beans/anti/anemic.php

```
<?php
class AdminController extends Zend_Controller_Action
```

†4 監訳注：凝集度とは、クラスがどれだけ自身の責務に集中しているかを示す尺度です。

†5 監訳注：ファウラーのブログエントリは http://www.martinfowler.com/bliki/AnemicDomainModel.html 、翻訳は http://capsctrl.que.jp/kdmsnr/wiki/bliki/?AnemicDomainModel で読めます。

†6 監訳注：このサンプルではDoctrineバージョン1が使われています。Doctrineはバージョン2でアーキテクチャが大きく変更され、Active Recordパターンの代わりにData Mapperパターンの実装を提供しています。http://www.doctrine-project.org/

```
{
  public function assignAction()
  {
    $bugsTable = Doctrine_Core::getTable("Bugs");
    $bug = $bugsTable->find($_POST["bug_id"]);
    $bug->Products[] = $_POST["product_id"];
    $bug->assigned_to = $_POST["user_assigned_to"];
    $bug->save();
  }
}

class BugController extends Zend_Controller_Action
{
  public function enterAction()
  {
    $bug = new Bugs();
    $bug->summary = $_POST["summary"];
    $bug->description = $_POST["summary"];
    $bug->status = "NEW";

    $accountsTable = Doctrine_Core::getTable("Accounts");
    $auth = Zend_Auth::getInstance();
    if ($auth && $auth->hasIdentity()) {
      $bug->reported_by = $auth->getIdentity();
    }
    $bug->save();
  }

  public function displayAction()
  {
    $bugsTable = Doctrine_Core::getTable("Bugs");
    $this->view->bug = $bugsTable->find($_GET["bug_id"]);

    $accountsTable = Doctrine_Core::getTable("Accounts");
    $this->view->reportedBy = $accountsTable->find($bug->reported_by);
    $this->view->assignedTo = $accountsTable->find($bug->assigned_to);
    $this->view->verifiedBy = $accountsTable->find($bug->verified_by);

    $productsTable = Doctrine_Core::getTable("Products");
    $this->view->products = $bug->Products;
  }
}

class SearchController extends Zend_Controller_Action
```

```
{
  public function bugsAction()
  {
    $q = Doctrine_Query::create()
      ->from("Bugs b")
      ->join("b.Products p")
      ->where("b.status = ?", $_GET["status"])
      ->andWhere("MATCH(b.summary, b.description) AGAINST (?)", $_GET["search"]);
    $this->view->searchResults = $q->fetchArray();
  }
}
```

アクティブレコードを扱うコントローラクラスのコードは、アプリケーションロジックを手続き型アプローチで書くことになりがちです。データベーススキーマの変更やアプリケーションの仕様変更がある場合、コードの多くの場所を変更しなければなりません。また、新たなコントローラを加える場合、たとえモデルに対するクエリが他のコントローラ向けのものとよく似ていても、新しいコードを書く必要があります。

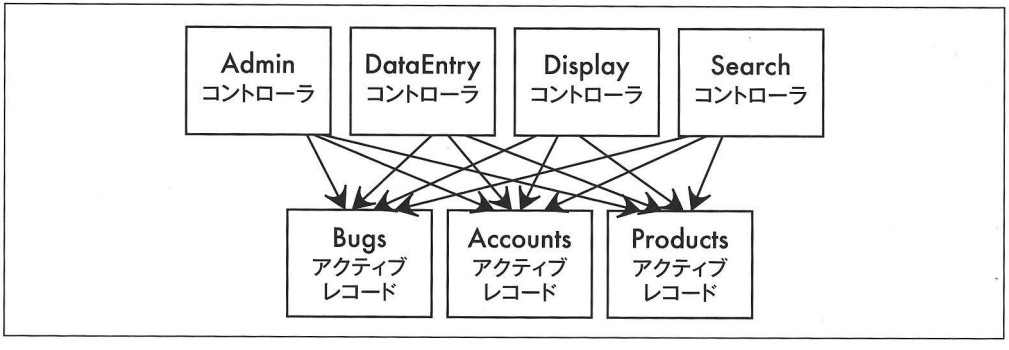


図 24-2 魔法の豆を使うと、つるのように絡まってしまう

これらのクラスのインタラクション図（図 24-2）は、複雑で非常に読みづらいものです。コントローラや DAO クラスが増えるにつれ、さらに酷くなります。この状況は、複数のモデルをまとめて使用するコードが、複数のコントローラにわたって重複していることの手がかりになります。別のアプローチを用いて単純化し、アプリケーションの一部を**カプセル化**しなければなりません。

24.2.4 マジックビーンズのユニットテストは困難

「マジックビーンズ（魔法の豆）」アンチパターンに陥っているときには、MVC の各レイヤーのユニットテストが難しくなります。

モデルのテスト

モデルをアクティブレコードと同じクラスにしているため、モデル自身の振舞いのテストを、データベースアクセスから分離して行うことができません。モデルをテストするために、本物のデータベースに対してクエリを実行しなければならないのです。

多くの開発者は**データベースフィクスチャ**（database fixture）を使用します。データベースフィクスチャは、テストデータベースへデータを読み込み、各テストが毎回同じ状態のデータベースに対して実行されることを保証する仕組みです。しかしデータベースの準備と後始末の作業は複雑であり、モデルのテスト時間を長引かせ、エラーを起こしやすくします。そしてもちろん、テストのためのデータベースが必要になってしまいます。

ビューのテスト

ビューをHTMLとしてレンダリングした結果を解析して、モデルが関係する動的なHTML要素が出力されていることを検証します。フレームワークによって結果の検証方法が単純化されていても、特定のHTML要素のレンダリングと解析のために、フレームワークは複雑で時間のかかるコードを実行しなければなりません。

コントローラのテスト

単なるデータアクセスオブジェクトでしかない設計のモデルは、複数のコントローラにおけるコードの重複を招くため、コントローラのテストも複雑になります。重複したコードも、すべてテストする必要があります。

コントローラのテストでは、偽物のHTTPリクエストを作成する必要があります。ウェブアプリケーションの出力はHTTPレスポンスのヘッダーとボディです。テストでコントローラの検証を行うには、対象のコントローラが返すHTTPレスポンスの一部分だけを切り出さなくてはなりません。このため、ビジネスロジックのテストを行いたいだけなのにも関わらず、多くの前準備のコードが必要となり、テストの実行にも時間がかかります。

データベースアクセスからビジネスロジックを、プレゼンテーションからビジネスロジックをそれぞれ分離できれば、MVCの目的の達成に役立ち、テストを簡易化できます。

漏れのある抽象化

ジョエル・スポルスキーは、2002年に「leaky abstracsions（漏れのある抽象化）」という用語を生み出しています[†7]。抽象化は、テクノロジーの内部の仕組みを隠蔽し、テクノロジーを使い易くします。しかし、生産性を上げるために内部の仕組みを知らなければならないとき、抽象化は「漏れやすく」なってしまうのです。

MVCのモデルとしてのアクティブレコードパターンは、漏れのある抽象化の典型例です。非常にシンプルなケースでは、アクティブレコードは魔法のように機能します。しかしデータベースアクセスすべてに対して使おうとすると、`JOIN`や`GROUP BY`などの多くの操作が、SQLのときのようには簡単に表現できなくなります。

†7 漏れのある抽象化の法則『Joel on Software』[Spo05]を参照してください。

一部のフレームワークは、アクティブレコードで様々なSQL句のサポートを試みています。しかし、クラスで内部的にSQLを扱えば扱うほど、SQLを直接使った方が良いのではないかという印象が強まります。

抽象化は、秘密を隠しておくことに失敗したのです。『オズの魔法使い』で、犬のトトがカーテンの端をくわえて、隠れていた魔法使いの正体を明らかにするように。

24.3 アンチパターンの見つけ方

「**マジックビーンズ（魔法の豆）**」アンチパターンの兆候を示す発言の例を見てみましょう。

「モデルにカスタムSQLクエリを渡すにはどうすればいい？」

データベースアクセスクラスをモデルクラスとして使用していることを示唆しています。SQLクエリをモデルに渡さなければならない状況は避けるべきです。モデルクラスは必要なクエリをすべて**カプセル化**すべきなのです。

「複雑なモデル操作をすべてのコントローラにコピーすればいいのだろうか、それとも、親の抽象コントローラクラスに1回だけコーディングすればいいのだろうか」

どちらの方法からも、求めている安定性やシンプルさを得られません。モデル内で複雑な問い合わせ操作をコーディングし、モデルのインタフェースの一部として公開すべきです。そうすることで、「**DRY原則（Don't Repeat Yourself：同じことを繰り返さない）**」[†8]を満たし、モデルの使用法をシンプルにできます。

「モデルのユニットテストを行うために、データベースフィクスチャをもっと書かなければならない」

データベースフィクスチャを使用している場合、あなたはビジネスロジックではなくデータベースアクセスのテストをしていることになります。データベースから独立してモデルをユニットテストできなければなりません。

24.4 アンチパターンを用いてもよい場合

アクティブレコードパターンそれ自体には、何も悪い点はありません。アクティブレコードパターンは、シンプルなCRUD操作にとって便利なパターンです。ほとんどのアプリケーションでは、テーブルの各行に対して単純な操作を行う、シンプルなデータアクセスオブジェクト（DAO）だけで十分な場合も多いのです。そのような場合には、モデルとDAOを同一のものとして定義することで、単純化を行うことができます。

アクティブレコードは、**プロトタイプ作成**においても効果を発揮します。素早くコードを書くことが重要なときは、テスト可能性や保守性よりも、作業の省力化の方が重要になります。動作するプロトタイプを初期段階で頻繁に見せることで、生きたフィードバックを得やすくなり、プロジェクトを早期に洗練させることができます。このような状況では、プロトタイプの開発を速めることで様々なメリットが得られるため、

†8 DRY原則は、『達人プログラマー—システム開発の職人から名匠への道』[HT00]でアンディ・ハントとデイブ・トーマスが提唱した原則です。

シンプルなアプリケーションフレームワークが役立つのです。

ただし、プロトタイプ作成時のコーディングによって**技術的負債**が生じます。負債を減らすために、コードのリファクタリングを行う時間を、あらかじめスケジュールに組み込んでおくことが大切です。

24.5 解決策：モデルがアクティブレコードを「持つ」ようにする

コントローラはアプリケーションの入力を処理し、ビューはアプリケーションの出力を処理します。どちらもシンプルであり、やるべきことは明確に定義されています。フレームワークは、コントローラとビューを素早く作成するためには最適ですが、モデルについての万能な解決策を提供することは困難です。なぜならモデルは、アプリケーションの残りの部分のオブジェクト指向設計を構成する要素だからです。

アプリケーションにおけるオブジェクトとは何か、そのオブジェクトが持つデータや振舞いとは何かについて、熟考しなければなりません。ソフトウェア開発の多くの部分は、知的で独創的な活動であるというロバート・L・グラスの言葉 [Gla92]を思い出しましょう。

24.5.1 モデルを理解する

幸い、オブジェクト指向設計の分野には、開発者のガイドラインとなる多くの優れた考えがあります。例えばクレーグ・ラーマンは自著『実践 UML—パターンによる統一プロセスガイド』[Lar04]で、GRASP（General= 汎用、Responsibility= 責任、Assignment= 割当て、Software= ソフトウェア、Patterns= パターン）と呼ばれるガイドラインを提示しています。以下に示すガイドラインは、特にモデルをデータアクセスオブジェクトから分離する際の設計判断において参考になります。

情報エキスパート（Information Expert）

操作の責任を持つオブジェクトは、その操作を果たすために必要な**すべての**データを持つべきです。操作の一部は複数のテーブルに関連しますし、どのテーブルとも関連しない操作もあるでしょう。アクティブレコードは1回に1つのテーブルを扱うことに特化しているため、複数のデータベースアクセスオブジェクトを集約して、複合的な操作を行うクラスを別に作成する必要があります。

アクティブレコードのような DAO とモデルとの間の関係は、is-a（**継承**）ではなく has-a（**集約**）であるべきです。アクティブレコードを使用するほとんどのフレームワークは、is-a 型を想定しています。しかし、モデルが DAO クラスからの継承ではなく DAO を使う関係にある場合、対象となるドメインのすべてのデータとコードを含むようなモデルを設計できます。モデルは、ドメインを表現するために複数のテーブルを参照することもできます。

生成者（Creator）

モデルがデータベース内のデータを扱う方法は、外部に公開されない、内部実装の詳細であるべきです。DAO を集約するドメインモデルが、これらのデータオブジェクトを生成する責任を持つべきです。

コントローラとビューは、ドメインモデルのインタフェースを使用すべきです。モデルのデータ取得や格納のためにデータベースとどのようなやりとりが必要なのか、コントローラとビューが知っている必要はありません。このような設計判断を行うことにより、一カ所変更するだけで、後からでもデータベースクエリ

の変更を容易に行えます。

疎結合性（Low Coupling）

論理的に独立しているコードは、分離して**疎結合化**を行うことが重要です。疎結合化を行うことにより、コードの利用者に影響を与えずに、クラスの実装を柔軟に変更できるようになります。アプリケーションの要件は単純化できません。コードのどこかには、複雑な部分が存在します。しかし、その複雑性をどこに実装するかは、自分で選択できるのです。

高凝集性（High Cohesion）

ドメインモデルクラスのインタフェースは、物理的なデータベース構造やCRUD操作ではなく、**意図**を示すべきです。アクティブレコードインタフェースの一般的なメソッドであるfind、first、insert、saveなどからは、どうやってアプリケーションの要件を満たせば良いのかが伝わりません。assignUserのようなメソッドはより説明的で、コントローラのコードも理解しやすくなります。

モデルクラスを、モデルが使用するDAOから分離すると、同じDAOを使う複数のモデルクラスを設計できるようにもなります。こうすることで、テーブルに対する操作すべてをアクティブレコードのサブクラスに詰め込むよりも、良い凝集性を得ることができます。

24.5.2 ドメインモデルの使用

エリック・エヴァンスは、『エリック・エヴァンスのドメイン駆動設計』[Eva03]のなかで、**ドメインモデル**と呼ばれる優れた解決策を提唱しています。

一部のソフトウェアの偏った意味合いではなく、MVC本来の意味合いにおけるモデルとは、オブジェクト指向によって対象ドメインをアプリケーションの中に表現することです。それはすなわち、アプリケーションのビジネスルールと、ビジネスルールのためのデータです。モデルとは、アプリケーションのビジネスロジックを実装する場所です。データベースとのやりとりは、モデルの内部的な実装の詳細なのです。

データベースの構造ではなく、アプリケーションの概念に基づいてモデルを設計すると、データベースの操作をモデルクラスに完全に隠蔽して実装できるようになります。前述したサンプルコードをどのようにリファクタリングできるか、例を見てみましょう。

Magic-Beans/soln/domainmodel.php

```
<?php

class BugReport
{
  protected $bugsTable;
  protected $accountsTable;
  protected $productsTable;

  public function __construct()
  {
```

```
    $this->bugsTable = Doctrine_Core::getTable("Bugs");
    $this->accountsTable = Doctrine_Core::getTable("Accounts");
    $this->productsTable = Doctrine_Core::getTable("Products");
  }

  public function create($summary, $description, $reportedBy)
  {
    $bug = new Bugs();
    $bug->summary = $summary;
    $bug->description = $description;
    $bug->status = "NEW";
    $bug->reported_by = $reportedBy;
    $bug->save();
  }

  public function assignUser($bugId, $assignedTo)
  {
    $bug = $bugsTable->find($bugId);
    $bug->assigned_to = $assignedTo;
    $bug->save();
  }

  public function get($bugId)
  {
    return $bugsTable->find($bugId);
  }

  public function search($status, $searchString)
  {
    $q = Doctrine_Query::create()
      ->from("Bugs b")
      ->join("b.Products p")
      ->where("b.status = ?", $status)
      ->andWhere("MATCH(b.summary, b.description) AGAINST (?)", $searchString);
    return $q->fetchArray();
  }
}


class AdminController extends Zend_Controller_Action
{
  public function assignAction()
  {
    $this->bugReport->assignUser(
```

```
      $this->_getParam("bug"),
      $this->_getParam("user"));
  }
}


class BugController extends Zend_Controller_Action
{
  public function enterAction()
  {
    $auth = Zend_Auth::getInstance();
    if ($auth && $auth->hasIdentity()) {
      $identity = $auth->getIdentity();
    }
    $this->bugReport->create(
      $this->_getParam("summary"),
      $this->_getParam("description"),
      $identity);
  }


  public function displayAction()
  {
    $this->view->bug = $this->bugReport->get(
      $this->_getParam("bug"));
  }
}

class SearchController extends Zend_Controller_Action
{
  public function bugsAction()
  {
    $this->view->searchResults = $this->bugReport->search(
      $this->_getParam("status", "OPEN"),
      $this->_getParam("search"));
  }
}
```

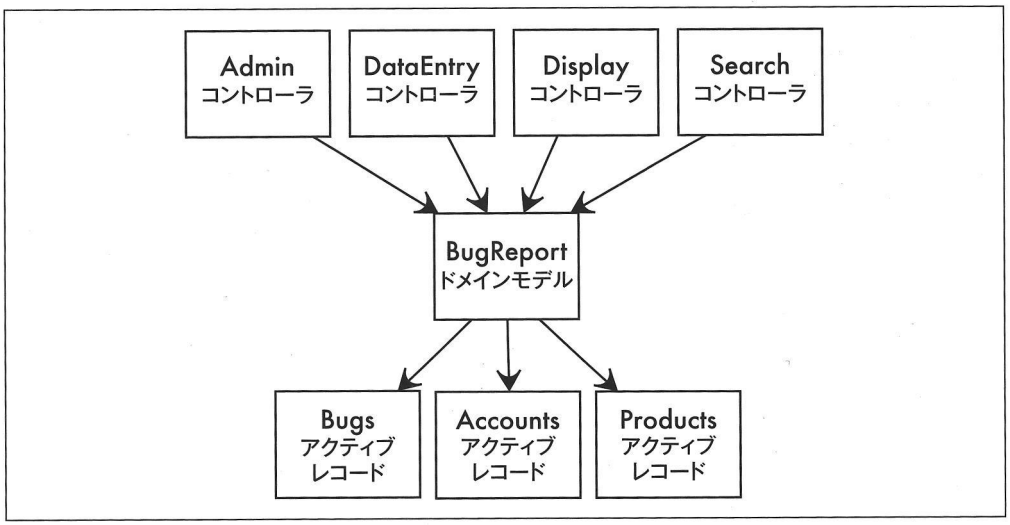


図24-3 疎結合化によって、絡まったつるをほどく

リファクタリング後のコードを読むと、いくつかの改善点に気づくはずです。

- クラスインタラクション図（図24-3）がシンプルになり、ぐっと読みやすくなりました。クラスの関係が**疎結合化**されたことがわかります。
- データベースの構造からモデルのインタフェースを分離することで、コントローラのコード量も減り、シンプルになりました。
- 各モデルクラスが、1つ以上のテーブルと相互作用するためのオブジェクトを作るようになりました。コントローラは、どのテーブルを操作する必要があるかを知らなくてもよくなります。
- モデルクラスはデータベースクエリを**カプセル化**し、隠蔽します。コントローラが関与するのは、ユーザーの入力内容の取得と、モデルのAPIを経由した高レベルのタスクのみです。
- 場合によっては、クエリが複雑なためにDAOでは簡単に処理できなくなり、カスタムSQLを書く必要が生じます。DAOへのアクセスがモデルクラス内に安全にカプセル化されていれば、データベースアクセスに生SQLを使用する影響範囲も狭くなり、リスクも減ります。

24.5.3 プレーンなオブジェクトのテスト

理想的なのは、本物のデータベースに接続することなくモデルをテストできることです。モデルからDAOを分離させると、DAOの**スタブ**や**モック**を作成できるようになり、モデルのユニットテストをデータベースから独立して行うことができます[†9]。

同様に、ドメインモデルのインタフェースも、通常のオブジェクト指向コードのユニットテストと同じよ

†9 監訳注：スタブやモックは、環境や他のクラスの実装から独立したテストを行うために作成する偽物のオブジェクトです。

うにテストできます。シンプルにオブジェクトのメソッドを呼び出し、メソッドの戻り値を検証すれば良いのです。偽物のHTTPリクエストを作成してコントローラにアクセスし、結果のHTTPレスポンスを解析するよりも、素早く簡単に実行できます。

コントローラのテストも引き続き偽物のHTTPリクエストを活用して行いますが、コントローラのコードがシンプルになったため、多くの分岐をテストする必要がなくなります。

モデルとコントローラを切り離し、モデルからデータアクセス機能を分離すると、これらすべてのクラスのユニットテストを、しっかりと分離したうえで、シンプルに実行できるようになります。すると、欠陥箇所の特定が容易になります。それこそがユニットテストに重要な点なのです。

24.5.4 現実的に考える

データアクセスオブジェクトは、どのようなソフトウェア開発フレームワークにおいても、たとえそのフレームワークが「マジックビーンズ（魔法の豆）」アンチパターンを招きやすいものであっても、効果的に使用できます。ただし、オブジェクト指向の設計原則をよく理解していない開発者は、スパゲッティコードを書いてしまうことになるでしょう。

この章で見てきたドメインモデリングの基本は、テストとコードの保守性を高め、継続的開発を行うために最適な設計の選択に役立つはずです。データベースが関わるアプリケーション開発の生産性を、大いに高めることができるでしょう。

[モデルはテーブルから分離させましょう。]

25章
砂の城

奥野 幹也

地震・津波による被災の可能性、自然現象を起因とするシビアアクシデント（過酷事故）への対策、大量の放射能の放出が考えられる場合の住民の安全保護など、事業者である東京電力（以下「東電」という）及び規制当局である内閣府原子力安全委員会（以下「安全委員会」という）、経済産業省原子力安全・保安院（以下「保安院」という）また原子力推進行政当局である経済産業省、（以下「経産省」という）が、それまでに当然備えておくべきこと、実施すべきことをしていなかった。
——国会事故調調査報告書（http://naiic.go.jp/report/）より

2011年3月11日、東日本を大地震が襲いました。震災による被害は甚大なものでしたが、それに拍車をかけて事態を悪化させたのが過去最悪の原発事故です。この事故により、福島には広大な立ち入り禁止区域ができてしまいましたし、破損した原発の撤去（修復？）は未だに終わる気配がありません。原発周辺での地震の規模も確かに大きかったのですが、本来はこの程度の地震で倒壊すべきではありませんでした。国会事故調調査報告書によれば、原発事故は「人災」、すなわちヒューマンエラーです。ヒューマンエラーが、ここまでも大きな被害をもたらす結果になってしまいました。

ヒューマンエラーによって大きな被害がもたらされるといえば、ITの分野も負けてはいません。データ消失やデータ漏洩によって甚大な被害がもたらされるインシデントが度々発生しています。皆さんの中にも痛い経験をされたことのある方がいらっしゃるのではないでしょうか。周りを見渡しても、たびたびサービス停止やデータ消失などのアクシデントが報じられています。明日は我が身です。油断しているといつなんどきそのような被害の渦中に巻き込まれることになるかも知れません。

幸いにして、ITにはそのようなインシデントからシステムを守るためのテクニックが多数存在します。

25.1 目的：サービスの安定稼働

今日では、様々なサービスが24時間365日の連続稼働を前提で提供されています。基幹系業務は無停止が基本ですが、ユーザーが無料で利用できるウェブサービスですら無停止が当たり前のようになっています。サービスを提供する側にとって、サービスの停止は収益の減少に直結するからです。たとえ無料ブログのようなサービスであっても、データ消失などが起きれば信頼が大きく失墜することでしょう。また、サービス停止中にユーザーが離れてしまうといった懸念もあります。

収益の減少を極力減らすには、様々なテクニックを駆使して停止時間を極限まで抑えることです。手法に

よってコストは様々ですので、停止によって失われると予測される収益の規模によって手法を使い分けます。

25.2 アンチパターン：想定不足

問題は、どのようなことが起きるかという想定と、それぞれの事象への対策が十分に検討されていないことです。コンピュータシステムにおいて安全神話はありません。サービスを安定稼働させるには、トラブルは当然起きるものとして想定しておく必要があります。想定しただけでは不十分で、実際に運用時に事象が起きたときどのような対応をするべきかといった検討も必要です。どのような種類の問題が起きるかという点については、マシンの停止やトランザクションの失敗といった定石を抑えるのはもとより、事象を網羅的に想定するには過去に起きた事例が大いに参考になるでしょう。事象に応じて自動的にサービスを継続させる仕組みや対応手順を決めておくことで、被害が最小限に抑えられます。

意外となされていないのが、性能問題や障害が起きた時の対処をどうするかというポリシーの策定です。そういった問題に対処するには種々のデータが必要となりますが、データの採取をおこなうと、運用中のシステムに対してディスクスペースやオーバーヘッドといった負担がかかることがしばしばです。あらかじめ、どこまでのデータ採取なら許容できるかといったことを決めておけば、問題の対処もスムーズに行えます。反対に、採取しないことになっているデータが必要な場合には、原因の究明を諦めて回避策に注力すると言った対応が求められます。

データ採取やリカバリ処理などは、日頃から訓練しておくと良いでしょう。実際にやってみると手順を間違ったり想定以上の時間がかかったりするものですが、実際にやってみればそのような問題も洗い出すことができます。被害の拡大を防ぐためにもそういった訓練は無駄にはなりません。備えあれば憂いなしです。

25.3 アンチパターンの見つけ方

次のようなセリフを耳にしたら、想定すべき状況を見落としているかも知れません。

「データサイズが一ヶ月で３倍になった」

想定以上の早さでデータが増加するのはよくあることですが、それは想定外の事象が起きやすい状況です。想定を超えたデータサイズでの運用は既に未知の世界です。いつなんどきサービスへ深刻な影響を及ぼす事象が起きても不思議ではありません。想定以上のアクセスやユーザー数でも状況は同じです。

「データベースへの更新でデッドロックが起きる。製品のバグじゃないか？」

知識不足は大敵です。製品や技術への理解なくして適切な想定はできません。ちなみに、デッドロックは行単位で排他制御を行うRDBMSなら必ず発生し得る状況であり、適切な例外処理（主にトランザクションのリトライ）を実装しておく必要があります。

「問題を解決するためにベンダーが要求しているデータは本番環境の負荷が高すぎるので採取できない」

サービスがカットオーバーしてから常に順風満帆で何事もなく運用できるなどと考えてはいけません。問題は当然起きるものであり、問題が起きたらデータの採取が必要となります。そして、データ採取にはオーバーヘッドが付きものです。欲しいデータが詳細であればあるほど、オーバーヘッドは大きくなります。使用しているRDBMS製品ではどのようなデータを採取することができ、それぞれどの程度

のオーバーヘッドがあるかを事前に調べておくべきです。採取時にサービスへの影響が出るようなデータが必要になったときにどうするべきか。いざ必要となったときに慌てないよう、事前に想定しておくことが必要です。

「最近マシンをアップグレードした。だから性能の問題とは無縁だろう」

マシンを新しくしたからと言って、性能の問題から逃れられるわけではありません。マシンを最新のモデルのものへと交換したところで、その効果はたかが知れています。CPU 速度やメモリの容量が増えたとしても、多くて数倍程度でしょう。非効率なクエリや突発的なアクセスといった事象による負の効果は、数倍などというオーダーの性能アップでは到底安心出来るものではありません。CPU やメモリを増強しても、ボトルネックがディスク I/O やネットワークでは意味がありません。システム設計時には想定していなかったボトルネックが後から顕在化することもしばしばです。「いったん構成を決めたら終わり」ではなく、「将来的にマシンの増強が必要になるかも知れない」と想定しておくのが現実的でしょう。

25.4 アンチパターンを用いてもよい場合

対策を講じれば講じるほどコストは嵩みますので、無計画に対策を講じれば良いということではありません。仕組みや体制を考えるのは時間がかかる作業です。どこまでお金や手間をかけられるかは、そのシステムが停止することでどれだけの損失があるか、もしくは事業計画に影響があるか、どの程度の停止時間なら許容できるかといったことを考慮して検討する必要があります。オーバースペックにならないよう気をつけましょう。

スモールスタートで始めたウェブサービスなどではほとんど対策がなされていない場合がありますが、後から抜本的なアーキテクチャの見直しが必要になるかも知れません。Twitter や Facebook のような大規模サイトが何度もアーキテクチャの見直しを迫られているのは有名な話です。規模に応じて最適なシステムアーキテクチャは異なりますので、規模が大きくなるにつれて形を変えていくのは当然の話です。規模が大きくない時点で大規模サイトの真似をしても良いことは何もありません。

25.5 解決策

重大なインシデントを回避し、サービスの安定稼働を目指すのに必要なのは、どのようなトラブルが起こりうるかということを可能な限り想定しておくことです。トラブルは必ず起こります。壊れないハードウェア、バグのないソフトウェア、脆弱性のないアプリケーションという都合の良いものはありません。トラブルを「当然起きる日常的なもの」として捉えることが必要です。

ここでは、サービスの運用を始めるにあたって実施・想定しておくべき代表的な対策について紹介します。

25.5.1 ベンチマーク

大きなトラフィックが予想されるシステムでは、事前にどの程度まで処理が可能なのかということをベンチマークしておきましょう。運用を続けていくうちにトラフィックやデータ量の増大によって性能の問題に

突き当たるシステムは多く、まさに「石を投げれば当たる」状態です。トラフィックやデータ量の増大によって性能の限界に突き当たるのは「避けられない」現象です。水を入れ続ければコップがいっぱいになって溢れてしまうのと同じです。大切なのは、限界がいつやってくるのかということを予測し、限界が来る前に対策を取ることです。ボトルネックになりやすいのはデータベースだと言われて久しいですが、限界がコントロールできていれば怖くはありません。

そのための第一歩がベンチマークを取ることです。どの程度のトラフィックやデータ量まで耐えることが出来るかということを知っていれば、それらが増加するペースと照らし合わせることで、あとどのぐらいで限界を迎えるかという予測がある程度出来ます。限界が来てから慌てて対策をするのではなく、対策は計画的に行うべきです。ときにはアーキテクチャの変更が必要になる場合があるからです。(例えばシングルサーバーからレプリケーションを用いたスケールアウトやシャーディングへの転換など。)

ベンチマークをするにあたって重要な点は次の3つです。

1つめは、現実に即したシナリオを用いることです。ベンチマークはアプリケーションがどの程度の負荷まで耐えることが出来るかということを調べるためのものですから、アプリケーションと同じ負荷をかけなければ意味のある数字をはじき出すことは出来ません。ウェブアプリケーションであれば、ウェブアプリケーション用のロードジェネレーターを使い、現実に即したシナリオで負荷テストをするべきです。

2つめは、実際のアプリケーションで用いられるのと同程度のデータサイズで実施することです。データサイズが小さいうちは、データがすべてキャッシュに収まるために高速で、データサイズが大きくなってから急激に性能が低下してしまうというのはよくあることです。十分に大きなデータを使ってベンチマークをおこないましょう。

最後に、当然のことですが実際のシステムと同じハードウェアやOSを使いましょう。ベンチマーク用にテスト環境の型落ちのマシンを使うのはよくありません。ベンチマーク結果は使用するシステムによって大きく異なります。実際に使用するのと同じシステムでなければ、ベンチマークによって得られた数値には意味がありません。CPUの動作周波数が1.5倍違うからベンチマークの結果も1.5倍の差になるというふうにはならないのです。

25.5.2 テスト環境の構築

サービスをリリースしてからが本当のデバッグの始まりです。アプリケーションのデバッグはもちろんのこと、利用しているミドルウェアはデータベース管理システムのトラブルシューティングでもテスト環境は大活躍します。特に問題の切り分け(アプリケーションの問題なのか、それともミドルウェアやデータベースの問題なのか)や、本番環境では採取できない詳細なデータを採取する、本番環境に対する変更作業をリハーサルするといった用途には必須です。ベンチマークシナリオがよくできていれば、性能の問題でも活躍します。

テスト環境に用いるシステムは、本番環境で利用しているものと同じものを1セット用意するのが理想です。アプリケーションの修正を行った場合に性能が劣化してしまわないかを調べるため事前にベンチマークをするといった用途には、そっくり同じシステムでなければ意味がありません。有事に備え、テスト環境を事前に準備しておく、または直ぐに利用できる体制をとっておくようにしましょう。そういう意味ではパブリックかプライベートかは問わず、クラウド環境は有利かも知れません。

25.5.3 例外処理

データベース管理システムを用いたアプリケーションでは適切な例外処理を実装することが必須です。適切な例外処理は、「どのような例外が発生し得るのか？」という想定ができていることの裏返しです。そのためには、トランザクション実行中にどのようなエラーが発生するかを知っておく必要があります。

例えばトランザクションのデッドロックや（ロック待ちの）タイムアウトはどの製品でも起きるエラーであり、トランザクションをリトライする例外処理を仕込んでおくのが一般的です。その他、データベースサーバーへの接続が切れた場合の対処や、データベース製品固有の一時的な（致命的でない）エラーへの対処が必要となります。製品固有のエラーにもれなく対処するには、その製品に精通している必要があります。どのようなエラーが起きるかを想定するには製品知識は重要です。

構文エラーのように、明らかにバグまたはSQLインジェクションに対する脆弱性の可能性を示すような致命的なエラーが発生した場合には、即座に関係者へ連絡が行くような仕組みを作っておくと良いでしょう。

25.5.4 バックアップ

サービスの最後の生命線がバックアップです。ディスク装置の故障やオペミス、クラッカーによる攻撃などによって本番環境のデータが破壊されるというインシデントは起こりえます。データが破壊されればもちろんサービスは止まってしまいますが、肝心のデータが破壊されてしまっていては復旧できません。そんなとき、最後の防衛線となるのがバックアップです。この防衛ラインを超えたら後はありません。バックアップは万全にしておきましょう。

バックアップにまつわる誤解の代表格が、「RAIDで冗長化しているからバックアップはとらない」というものです。ディスクの冗長化はバックアップにはなりません。データが論理的に破壊されてしまうようなケースに対応できないからです。例えば誤って必要な行をDELETEした場合、ひとたびそれをCOMMITしてしまったならば、いかにディスク装置を冗長化していようともデータは元に戻りません。COMMITしてしまったら元に戻せないのがRDBMSの特性です。RAIDだけでなく、レプリケーションもバックアップの代わりに使用することはできません。オペミスによる変更などが即座に伝播してしまうからです。

バックアップ運用は万全に行いましょう。

25.5.5 高可用性

どれだけ高価なマシンを使おうと、マシンそのものの故障から完全に逃れることはできません。停止時間をできるかぎり短くするには、マシンを冗長化する仕組みを考えておく必要があります。マシンを冗長化しない場合、マシンが故障から復帰するまでサービスを再開することができないかも知れません。ハードウェアをまるごと交換するのでも無い限り、切り分けには時間がかかります。運が良ければOSのsyslogにエラーの部位を示す兆候が出ていたり、POST等で被疑箇所を特定できることがありますが、延々と切り分けができないケースもあります。また、新しく交換したパーツが故障していることもあるでしょう。マシンを冗長化しない運用は、かなり停止時間が長くなることを覚悟しなければなりません。（最悪数日という場合もありえます。）

一般的に高可用性構成と言えば、クラスタリングソフトウェアを用い、マシンがクラッシュした際にフェ

イルオーバーさせるものを指します。この場合、フェイルオーバー時にサービスを引き継いだ方のマシンにおいてファイルシステムやデータベースのクラッシュリカバリが行われます。ログファイルのサイズが大きかったり更新が激しかったりすると、クラッシュリカバリには時間がかかるかも知れません。

データベース製品によってはアクティブ / アクティブ構成やレプリケーションをサポートしているものがありますので、停止時間を短くするためにそういった機能を活用するのも重要です。コストと停止時間を天秤にかけて最適な構成を選びましょう。

25.5.6 ディザスタリカバリ

本当に重要なシステムでは、マシン単位の冗長化では十分でなく、データセンターないしはサイト全体の障害も考慮する必要があります。地震や津波などの自然災害、火災、電源の故障などにより、データセンターの一部または全体が使用不能になってしまった場合には、他のデータセンターで処理を引き継ぐというディザスタリカバリまで考慮しておくと良いでしょう。その場合には、データを継続的に複製し続けるレプリケーションなどの仕組みが必要になるでしょう。

25.5.7 運用ポリシーの策定

様々なテクニックを駆使して、問題が生じても自動でサービスを継続できる仕組みは素晴らしいですが、すべての事象をカバーできるわけではありません。システムの限界を超えた事象というのは起こりえます。

高可用性の限界を超えた障害

クラスターで二重化していたマシンが全滅したり、データセンターの特定のエリアの電源が全滅するというように、準備してある高可用性要件の限界を超えた場合にどうするべきかというのは1つの課題です。共有ディスクが故障してデータへアクセスできなくなる、ファイルシステムの損傷がひどくてfsckが終わらない、RDBMSのクラッシュリカバリが終わらないなどの要因で、フェイルオーバーが上手く行かない場合もあるでしょう。そのような場合、フェイルオーバーを繰り返すいわゆる「ピンポン」と呼ばれる現象が起きることになりますが、これはよく見かける風景です。そのような場合にどう対処をするかは現象次第ですが、ディスクを初期化してバックアップからリストアする、クラスターの使用をいったん取りやめて1つのマシンでサービスを起動するというような対処を念頭に置いておきましょう。

問題の調査

データベースと言えども、運用中に発生した問題の調査は厄介です。特に再現性がなく、トランザクションが失敗する程度の影響で、なおかつ稀にしか発生しないようなエラーは情報の採取も難しく、調査は難航することになります。本番環境から情報を採取するにしても、サービスへの影響を考えると詳細なデータが欲しくても手が出ないという状況にも陥りがちです。データベースソフトウェアには様々な情報採取の方法がありますが、オーバーヘッドが大きく、サービスへの影響があるものも少なくありません。影響があるからと言って一切手を出さないのも考えものです。リソースに十分な余裕があるなら影響があっても許容できるはずです。本番環境に対する影響がどの程度までなら許容出来るのかを事前に検討しておきましょう。本番環境から情報を採取できないのであれば、テスト環境で再現を試みる

しかありません。しかし再現性の低い問題を再現させるのは非常に手間がかかります。問題がサービスへ及ぼす影響が軽微である場合には、自動的にトランザクションをリトライして問題を無視するといった対処も視野に入れておくといいでしょう。

性能の劣化

処理に時間がかかるようになったり、マシンのリソースが限界まで消費されるような状況になった場合、最も効果的なのはクエリやスキーマをチューニングすることです。スケールアップやスケールアウトは、少しでもマシなクエリを書いてからにした方が良いでしょう。(NULLだらけのテーブルばかりといった、根本的に設計から見直す必要がある場合もありますが。) クエリを適切にチューニングしても現在のアーキテクチャで性能の限界を迎えてしまった場合に、どのように性能を伸ばしていくかという戦略は重要です。現行のアーキテクチャでスケールアップやスケールアウトでどこまで性能が伸びるのか、抜本的にアプリケーションのアーキテクチャを見直す必要があるのか、その際RDBMS製品を入れ替えるのかと言った検討を事前にしておくようにしましょう。

[どれだけ盤石な基盤を築けるかは、あなたがどれだけのインシデントを想定しているかが鍵なのです。]

奥野 幹也 (おくの みきや)

1975年生まれ。栃木県在住のギーク。フリー (自由な) ソフトウェアの普及をライフワークとしている。KDEを愛用。仕事ではMySQLのサポートに従事。著書には『エキスパートのためのMySQL[運用+管理]トラブルシューティングガイド』『MySQL Cluster構築・運用バイブル』(ともに技術評論社) がある。

V部
付録

付録 A
正規化のルール

若人よ。数学の世界において理解するということはない。ただ、慣れるのみである。
——ジョン・フォン・ノイマン（数学者）

リレーショナルデータベースの設計は、決して無作為なものでも不可解なものでもありません。あなたのアプリケーションを、エラーによる失敗から保護しつつ、無駄な冗長さのないデータ格納戦略を設計する際に役立つルールが数多く定義されているのです。それは、先に触れた「ポカヨケ」の例のようなものです。同様の考え方に「ディフェンシブ・デザイン」や「できるだけ早い段階で失敗すべし」などの喩えがありますが、こちらは聞き覚えがある方もいるのではないでしょうか。

正規化（normalization）のルールは複雑ではありませんが、分かりにくいところがあります。開発者はルールがどう適用されるかを見誤りがちです。ルールを実際以上に厳格なものとして捉え過ぎているのかもしれません。

もしくは、とにかくルールに従わされることにうんざりしてしまった可能性もあります。新規性、創造力、革新性に価値を見出す開発者にとって、ルールは厄介者です。ルールはある意味では自由の対極にあるとも言えるからです。

ソフトウェア開発者は常に、簡潔さと拡張性のバランスを求められます。あなたは、あらゆる種類のアプリケーションの開発において、車輪の再発明と、カスタマイズされたデータ管理ツールの開発に多くの労力を費すことでしょう。もし、設計をリレーショナル理論に準拠させることができるなら、先人の英知と技術というリレーショナル理論の強みを活かすことができます。

本書では学術的また理論的になり過ぎないよう、開発者のメリットや失敗談を例にとってアンチパターンについて述べてきました。本付録では、理論についても、より具体的な例でご紹介します。

A.1 リレーショナルとは何か

関係（relation：**リレーション**）という用語は、テーブル間の**関連**（relationship：**リレーションシップ**）を指すものでは**ありません**。リレーションとはテーブルそれ自体のことを意味するか、もしくは同じテーブル内の列間の関係性を意味しています。両方を指す場合もあります。

数学の世界では、**関係**（relation）を「各属性の定義域（ドメイン）から取り出された対の値で作りうる

組み合わせに、いくつか条件付けを行った結果として得られる部分集合」と定義しています。

野球のチーム名と都市名の対を例にとりましょう。チームと都市からなるペアの組み合わせは膨大な数になりますが、ここで必要とされるのはチームと本拠地という特定の部分集合です。有効な組み合わせは(レッドソックス，マイアミ)ではなく、(ホワイトソックス，シカゴ)、(カブス，シカゴ)、(レッドソックス，ボストン)などです。

リレーションという用語は、「この都市はあのチームの本拠地である」というようなルールの意味でも、またそのルールに基づいて条件付けを行った結果、すべての組み合わせから絞り込まれた部分集合の意味でも使われます。これをSQLで表現すると、2つの列が定義されたテーブルに、チームと都市のペアごとに1行格納されることになります。

もちろん「リレーション」が成立するのは2列の場合だけではありません。ドメインごとに何列でも組み合わせて「リレーション」をつくることができます。またドメインは32ビット整数型でも特定の長さの文字列型でも構いません。

テーブルの正規化を行うにあたっては、まず次のような基準に従って適切なリレーションが成立しているかどうかを確認する必要があります。

A.1.1　行に上下の順番がない

ORDER BY句で順番を指定しない限り、SQLクエリは予想できない順番で結果を返します。しかし順番を除けば、行の集合は等しくなります。

A.1.2　列に左右の順番がない

開発者のスティーブンに、製品「Open RoundFile」に対するバグ#1234のテストを依頼した場合も、バグ#1234の製品「Open RoundFile」での検証をスティーブンに依頼した場合も、結果は同じになるべきです。

列に左右の順番がないことは、名前ではなく位置で列を指定してしまう、18章「インプリシットカラム(暗黙の列)」アンチパターンと関連しています。

A.1.3　重複行を許可しない

1回述べた事実を、もう1回述べたところでその内容が変わるわけではありません。先ほどの野球チームの例では、チーム名によって都市が決まります。このとき、都市がチーム名に「従属している」といいます。

重複を防ぐためには、各行を区別し、個々の行を特定できなければなりません。このことをSQLで保証するには、1つまたは複数の列に主キー制約を宣言して、行を一意に識別できるようにします。

非キー列には、重複がある場合があります。例えば、ボストンを本拠地とするチームは2つあります。しかし、チーム名が異なっているために、1つの行としては一意性を維持できます。

A.1.4　すべての列は1つの型を持ち、各行に1つの値を持つ

リレーションには、列名とデータ型を定義する**ヘッダー**があります。すべての行は、ヘッダーと同じ列を

持っていなくてはなりません。またある列は、すべての行で同じ意味を持っていなくてはなりません。

このルールに2つの方法で反するアンチパターンを、5章「EAV（エンティティ・アトリビュート・バリュー）」で見ました。まずEAVテーブルは、すべてのインスタンスにおいて独自の属性を持つ可能性があるエンティティのモデルです。このため、エンティティはその属性を定義するヘッダーと対応しません。

2つ目の違反は、EAVのattr_value列が、バグの報告日、ステータス、割り当て先のアカウントなど、エンティティのすべての属性を含むことです。例えば、attr_value列のとある値1234は、2つの異なる属性の有効値となる可能性がありますが、属性によって意味付けがまったく異なるでしょう。

6章「ポリモーフィック関連」アンチパターンも、このルールに反します。1234のような値が、複数の親テーブルどれかの主キーを参照するからです。ある行の値1234が、別の行の値1234と同じものものであるとは断定できません。

A.1.5 行に隠されたコンポーネントがない

列には、行IDやオブジェクトIDのような物理ストレージの識別子ではなく、データの値そのものが格納されていなければなりません。21章「シュードキー・ニートフリーク（疑似キー潔癖症）」では、主キーは一意ではあるものの、行番号ではないことを説明しました。

データベースには、このルールに準拠せず、SQLへの拡張によって内部ストレージの詳細へのアクセスを可能にするものもあります（OracleのROWNUM疑似列、PostgreSQLのOIDなど）。ただし、これらの値は厳密にはリレーションの一部ではありません。

A.2 正規化の神話

極めて正確な定義がありながら、これほど誤解されているテーマもないかもしれません。正規化について、以下のような誤解を自信たっぷりに語る開発者は、石を投げれば当たるほど多くいます。

「正規化はデータベースをより遅くする。非正規化をすればデータベースは速くなる」

これは間違いです。正規化を適用した後、テーブルから属性を取得するためにJOINを使わなくてはならない場合があることは事実です。確かにデータを非正規化すれば、JOINのいくつかを回避できます。例えば、1章「ジェイウォーク（信号無視）」で見たカンマ区切りのリストは、あるバグに該当する複数の製品名のリストを格納します。しかし、特定の製品に該当するバグのリストが必要な場合はどうすればよいのでしょうか。非正規化は一般的に、1種類のクエリの利便性やパフォーマンスの向上には役立ちますが、他の種類のクエリで大きな代償を支払わなくてはならないのです。非正規化にも妥当な使用方法はあります。しかし非正規化を行うことを決定する前に、まずは一般的な形式でデータベースを設計すべきです。12章「インデックスショットガン（闇雲インデックス）」で紹介したインデックスのガイドラインMENTORは、非正規化にも当てはまります。変更を行う際には、その前後のパフォーマンスを測定して、適切に効率化を行うようにしましょう。

「正規化とは、データを子テーブルに追い出し、疑似キーを用いて参照することである」

これも間違いです。利便性、パフォーマンス、ストレージ効率などの目的で疑似キーを使うことは可能であり、妥当でもあります。しかし、これは正規化とは無関係です。

「正規化とは、EAV 設計のように、できる限り属性を切り離すことだ」

違います。開発者はよく、正規化によってデータが人間にとって読みにくくなったり、クエリを実行しにくくなるというような、誤った考えを持っています。しかし実際は、その反対なのです。

「第 4 正規形以上の正規化をする必要はない。第 3 正規形以外の正規形は難解であり、それが必要になる場面には出会わないだろう」

誤りです。ある研究によると、実業務に使われているデータベース設計の 20 パーセント以上が、第 3 正規形までには準拠しているが、第 4 正規形には違反しています。全体から見れば少数派とはいえ、これはかなりの割合です。例えば、もしデータ喪失が発生する可能性のあるバグがアプリケーションの 20 パーセントに存在しているとすれば、修正したいと考えるのではないでしょうか。

A.3　正規化とは何か

以下に、正規化の目的を挙げます。

- 人間が理解できる形で、現実世界の事実を表現する
- 事実の格納方法から冗長性を排除し、データの異常や不整合を防ぐ
- 整合性制約をサポートする

上記リストにパフォーマンスの改善が含まれていないことに注意しましょう。正規化は、「正確に」データを格納することと、問題の発生を防ぐことに寄与します。正規化されていないデータベースは、混乱に陥ることを避けられません。結果として、不整合や重複が生じたデータに対処するために、多くのコードを書かなくてはならなくなります。データの欠陥によって、ビジネスには遅延やコストが生じます。これらのシナリオを考慮すれば、データベースの正規化によってパフォーマンス上もメリットが得られることは明らかです[†1]。

正規化のルールを満たしているテーブルの状態を、**正規形**（normal form）と呼びます。一般的な正規化のレベルは 5 つあり、これらは正規化の進行の度合いを表します。それぞれの正規形は、リレーションの設計時に生じる特定のタイプの冗長性や不整合を、それぞれ排除します。一般に、ある正規形をテーブルが満たしている場合、それ以前のすべてのレベルの正規形も満たしています。他にも、研究者が規定した 3 つの正規形があります。正規形の進行を図 A-1に示します。

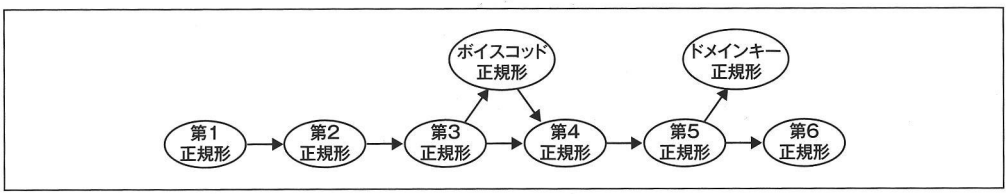


図 A-1　正規化の進行

†1　監訳注：外部キー、参照整合性制約とパフォーマンスに関しては 4 章「キーレスエントリ（外部キー嫌い）」で詳しく論じられています。

A.3.1 第1正規形

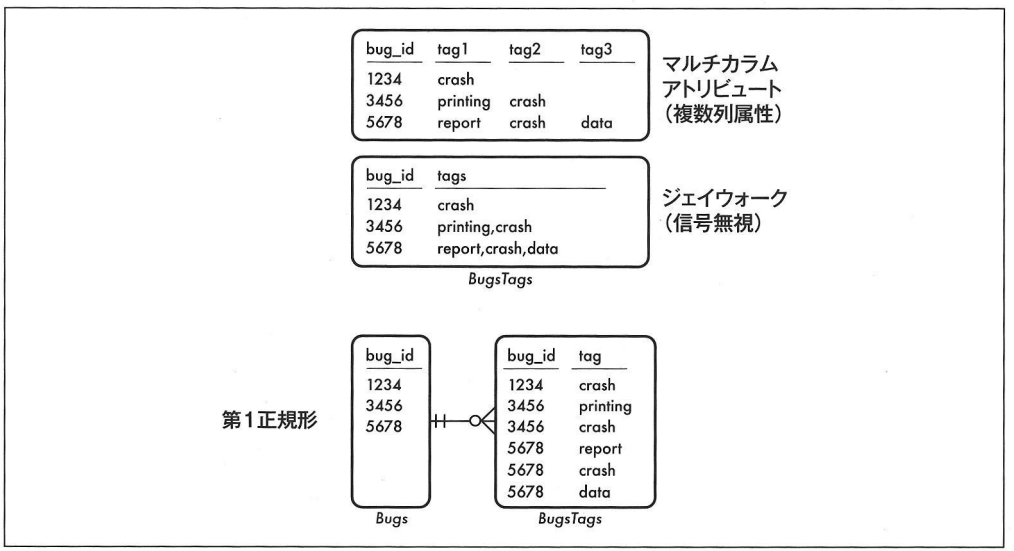
第1正規形の最も基本的な要件は、「テーブルがリレーションであること」です。この付録の冒頭で説明した「リレーション」の定義を満たしていない場合、テーブルを第1正規形またはそれ以降の正規形にすることはできません。

次の要件は、テーブルに「**繰返しグループ**（repeating group）がないこと」です。前述したように、リレーションの各行は複数の集合の組み合わせであり、各列の値は各集合から1つ選択されます。繰り返しグループとは、1つの行が、ある集合の複数の値を含んでいることを指します。

本書では、繰返しグループを生んでしまう以下の2つのアンチパターンを見てきました。

- 7章「マルチカラムアトリビュート（複数列属性）」は、同じドメイン（定義域）の複数の値を、複数の列に格納しています。
- 1章「ジェイウォーク（信号無視）」は、複数の値を単一の列に格納しています。

図A-2に、この2つのアンチパターンにおける繰返しグループの例を示します。第1正規形を満たすための適切な設計では、別テーブルを作成する必要があります。タグを新たに作成したBugsTagsテーブルの1つの列に格納し、行ごとに1つのタグ（tag）を格納することで、複数のタグをサポートできるようになります。



図A-2 繰り返しグループと第1正規形

A.3.2 第2正規形

第2正規形は、テーブルが複合主キー（bug_id, tag）を持つことを除いては、第1正規形とまったく同

じです。先ほどのタグの例を引き続き拡張して、どのユーザーがどのタグをバグに付けたか（tagger）を追跡管理するとしましょう。加えて、特定の名前のタグを誰が最初に作成したのか（coiner）についても記録することにします。

Normalization/2NF-anti.sql

```
CREATE TABLE BugsTags (
  bug_id  BIGINT NOT NULL,
  tag     VARCHAR(20) NOT NULL,
  tagger  BIGINT NOT NULL,
  coiner  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id),
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);
```

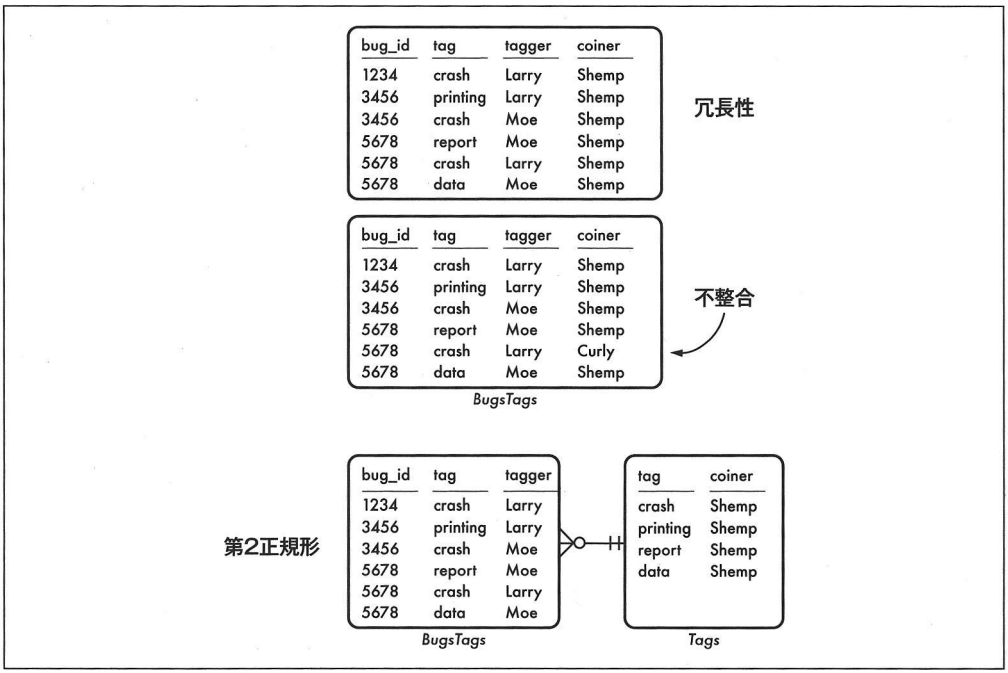


図 A-3　冗長性と第 2 正規形

図 A-3 を見ると、タグ作成者（coiner）の ID が重複して格納されていることがわかります[†2]。これは、誰かが「crash」タグの作成者 ID を変更したものの、同じ「crash」タグが付いた他の行に対して変更の反

†2　この図では、アカウント ID の代わりにアカウント名でユーザーを識別するものとします。

映を行わなかったために、データの不整合が発生していることを意味します。

第2正規形を満たすためには、あるタグの作成者を格納するのは1回だけにしなければなりません。このため、新たにTagsテーブルを定義します。Tagsテーブルでは、タグ名を主キー（tag）として、個別のタグをそれぞれ1行に格納するようにします。タグの作成者（coiner）をBugsTagsテーブルではなくTagsテーブルに格納することで、データ不整合を防ぎます。

Normalization/2NF-normal.sql

```
CREATE TABLE Tags (
  tag     VARCHAR(20) PRIMARY KEY,
  coiner  BIGINT NOT NULL,
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsTags (
  bug_id  BIGINT NOT NULL,
  tag     VARCHAR(20) NOT NULL,
  tagger  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tag) REFERENCES Tags(tag),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id)
);
```

A.3.3 第3正規形

続いて、Bugsテーブルに、バグ修正担当エンジニアの電子メールアドレスを格納したくなったものとしましょう。

Normalization/3NF-anti.sql

```
CREATE TABLE Bugs (
  bug_id         SERIAL PRIMARY KEY,
  -- 他の列. . .
  assigned_to    BIGINT,
  assigned_email VARCHAR(100),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

しかし、電子メールアドレスはエンジニアのアカウントの属性であり、厳密にはバグの属性ではありません。Bugsに電子メールアドレスを格納すると冗長になり、第2正規形違反の例で見たテーブルのように、データ不整合のリスクが生じてしまいます。

第2正規形違反の例では、違反した列（coiner）は複合主キー（bug_id, tag）のうち、少なくとも

一部（tag）との間に対応関係がありました。今回のテーブルは**第 3 正規形**に違反します。違反した列（assigned_email）は主キー（bug_id）にまったく対応付いていないからです。

第 3 正規形違反を修正するには、電子メールアドレスを Accounts テーブルに格納する必要があります。Bugs テーブルから列を分離した結果を図 A-4 に示します。電子メールアドレス列（email）が自身の属するテーブルの主キー（account_id）に直接対応しており、冗長性がなくなっています。

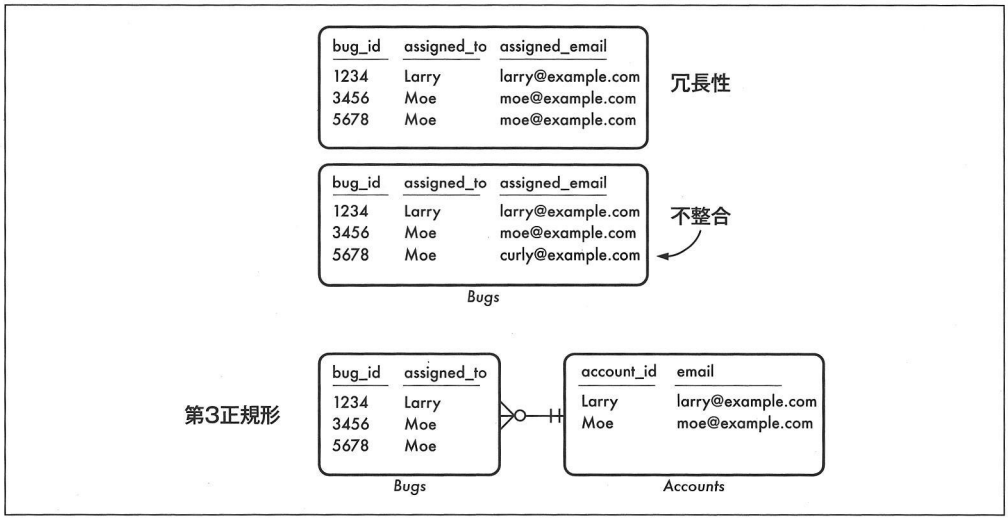


図 A-4　冗長性と第 3 正規形

A.3.4　ボイスコッド正規形

第 3 正規形をわずかに厳格にしたものを、**ボイスコッド正規形**（Boyce-Codd Normal Form：BCNF）と呼びます。この 2 つの正規形の違いは、第 3 正規形ではすべての非キー属性がテーブルのキーに従属しなければならないのに対し、ボイスコッド正規形ではキー列にもこのルールが適用される点です。ボイスコッド正規形のルールは、キーになる可能性がある列の組み合わせがテーブルに複数ある場合にのみ該当します。

例えば、タグには 3 つのタイプがあるとします。バグの影響を記述するタグ（impact）、バグが影響するサブシステムを記述するタグ（subsystem）、バグの修正ステータスを記述するタグ（fix）です。ここで、各々のバグには、タグタイプごとに最大でも 1 つのタグしか割り当てられないものとします。このときキーの候補となる可能性があるのは bug_id と tag の組み合わせですが、実は bug_id と tag_type の組み合わせでも一意になります。どちらの列の組も、個々の行を十分に識別できるからです。

図 A-5 に、第 3 正規形ではあるが、ボイスコッド正規形ではないテーブルと、その変更方法の例を示します。

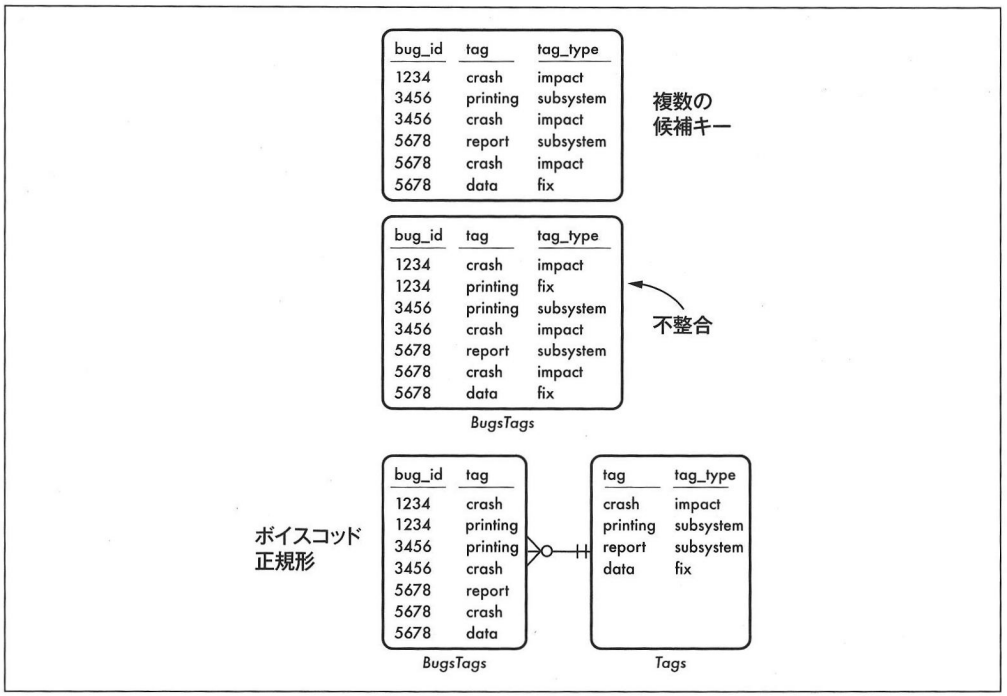


図 A-5 第 3 正規形とボイスコッド正規形

A.3.5 第 4 正規形

データベースを変更し、複数のユーザーからのバグ報告、複数の開発エンジニアへのバグ修正担当割り当て、複数の品質保証エンジニアへのバグ検証担当割り当てをできるようにします。多対多関連を作成するために、追加のテーブルを作成します。

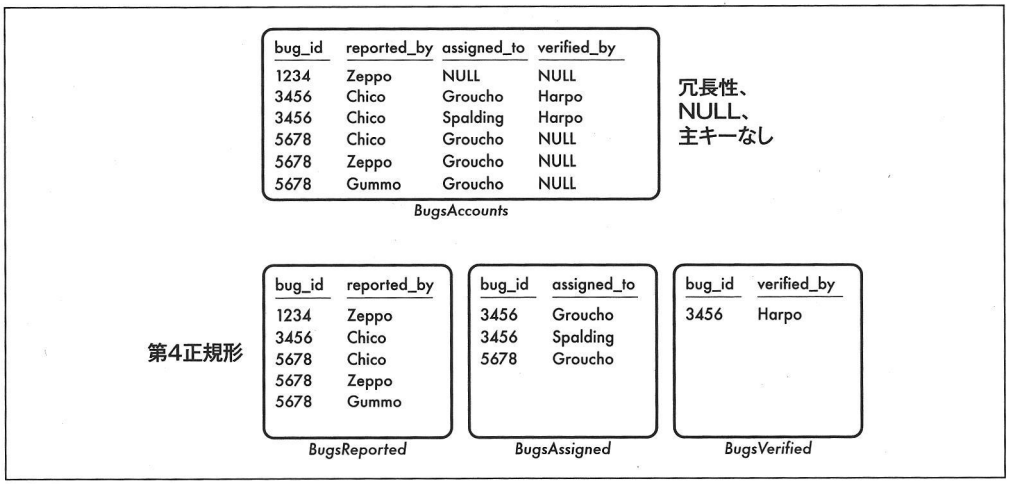
Normalization/4NF-anti.sql

```
CREATE TABLE BugsAccounts (
  bug_id       BIGINT NOT NULL,
  reported_by  BIGINT,
  assigned_to  BIGINT,
  verified_by  BIGINT,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

bug_idは、単独では主キーにできません。バグに対して複数のバグ報告者（reported_by）、複数のバグ

修正担当者（assigned_to）、複数のバグ検証担当者（verified_by）を関連付けるには、バグごとに複数の行が必要だからです。同様に、最初の2列または最初の3列では、まだ主キーにはなりません。最後の列(verified_by)に複数の値を関連付けることができなくなるからです。つまり、4つすべての列の組み合わせをもって、主キーとして宣言する必要があります。しかし、assigned_to列と verified_by列は NULLを許容するべきです。バグは、修正担当割り当て前、または検証担当割り当て前であっても報告できるためです。すべての主キー列は、基本的に NOT NULL制約を持たなければならないにもかかわらずです。

他にも問題があります。ある列に格納するアカウントが他の列より少ないと、冗長な値が生じてしまう場合があるのです。図A-6に、冗長な値が生じた状態を示します。



図A-6　混ざり合ったリレーションシップと第4正規形

ここまで述べたすべての問題は、2重の役割（ここでは3重ですが）を持つ交差テーブルを作ろうとしたことに起因しています。1つの交差テーブルで複数の多対多関連を表そうとすると、**第4正規形**に違反します。

図A-6では、この問題を解決するためにテーブルを分割し、各タイプの多対多関連ごとに1つの交差テーブルを持つように変更する方法を示しています。これにより、各列における冗長性と値の数の不一致の問題が解決されます。

Normalization/4NF-normal.sql

```
CREATE TABLE BugsReported (
  bug_id       BIGINT NOT NULL,
  reported_by  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, reported_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
```

```
);

CREATE TABLE BugsAssigned (
  bug_id       BIGINT NOT NULL,
  assigned_to  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsVerified (
  bug_id       BIGINT NOT NULL,
  verified_by  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, verified_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

A.3.6 第5正規形

ボイスコッド正規形の条件を満たし、複合主キーを持たないテーブルは、すでに**第5正規形**です。しかし、第5正規形を理解するために、例を見てみることにしましょう。

特定の製品群のみを担当するエンジニアがいるとします。あるエンジニアがある製品のあるバグに取り組んでいるという事実を、冗長性を最小限にしつつ記録できるように、データベースを設計しなければなりません。まず、BugsAssignedテーブルに product_id列を追加し、ある製品を担当するエンジニアを表すようにします。

Normalization/5NF-anti.sql

```
CREATE TABLE BugsAssigned (
  bug_id       BIGINT NOT NULL,
  assigned_to  BIGINT NOT NULL,
  product_id   BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

しかしこの設計は、エンジニアにどの製品が割り当て可能であるかを表していません。エンジニアが現在どの製品に割り当てられているかを表しているだけです。加えて、エンジニアが特定の製品に対して割り当てられているという事実を冗長に格納しています。これらの原因は、互いに独立した多対多関連についての

複数の事実を 1 つのテーブルに格納しようとすることで生じるもので、第 4 正規形の説明で見た問題と似ています。図 A-7 にこの冗長性を示します[†3]。

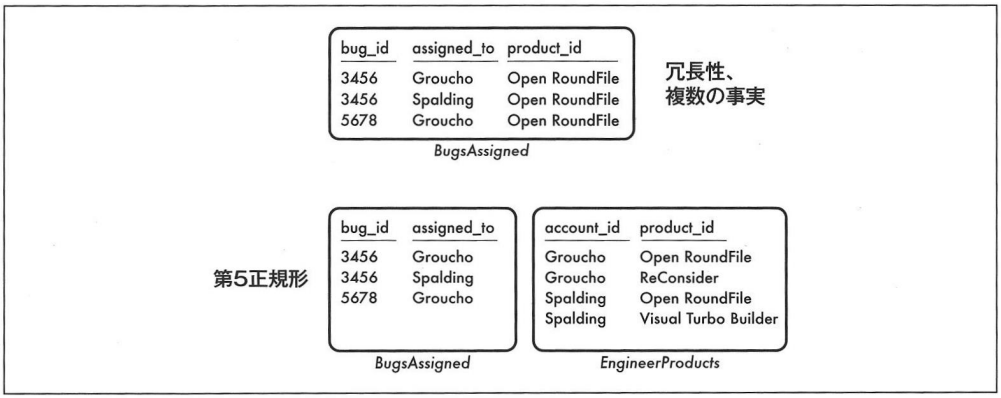


図 A-7　混ざり合ったリレーションシップと第 5 正規形

この問題は、各関連を別のテーブルに分離することで解決します。

Normalization/5NF-normal.sql

```
CREATE TABLE BugsAssigned (
  bug_id       BIGINT NOT NULL,
  assigned_to  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE EngineerProducts (
  account_id   BIGINT NOT NULL,
  product_id   BIGINT NOT NULL,
  PRIMARY KEY (account_id, product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

これで、あるエンジニアがある製品に割り当て可能であるという事実を、そのエンジニアがある製品のバグ修正に取り組んでいるという事実とは独立して記録できます。

†3　この図では、製品 ID の代わりに製品名で製品を識別するものとします。

A.3.7 他の正規形

ドメインキー正規形（Domain-Key Normal Form：DKNF）は、テーブルのすべての制約が、テーブルのドメイン制約とキー制約の論理的帰結であることを表します。第3正規形、第4正規形、第5正規形、ボイスコッド正規形はすべてDKNFの対象になります。

例えば、「NEW」または「DUPLICATE」のステータスを持つバグは、まだバグ修正が行われていないと判断できます。このため、このバグには作業時間 (hours) は記録されていないはずです。また、バグがまだ修正されていないにもかかわらず、verified_by列に品質保証エンジニアを割り当てることも意味をなしません。このような制約はトリガやCHECK制約で実装できますが、テーブルの非キー列間の制約であるため、DKNFの条件を満たしません。

第6正規形では、すべての**結合従属性** (join dependency) を排除します。これは通常、属性の変更履歴のサポートに使われます。例えばBugs.statusの値は、時間の経過とともに変化します。ステータスの変更履歴を、子テーブルに記録したい場合があります。同様に、変更日や変更者、他の詳細情報なども記録したいかもしれません。

Bugsテーブルが完全に第6正規形をサポートするには、ほぼすべての列に個別の履歴テーブルが必要になり、テーブル数が過剰に増えてしまいます。第6正規形は、ほとんどのアプリケーションにとって過剰な正規化です。ただし、データウェアハウスの一部では、第6正規形が使用されています[†4]。

A.4 正規化は常識的なもの

正規化のルールは、難解なものでも複雑なものでもありません。正規化のルールは冗長性を減らし、データの整合性を保つために用いられる、ごく常識的な技法なのです。

この付録で見てきたリレーションと正規形の概要を参考にしながら、ぜひ、今後のプロジェクトでより良いデータベース設計を目指してください。

†4 例えば、Anchor Modelingで使用されています。http://www.anchormodeling.com/

付録 B
参考文献

[BMMM98]

William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. AntiPatterns. John Wiley & Sons, New York, NY, 1998.
岩谷宏 訳、『アンチパターン—ソフトウェア危篤患者の救出』、ソフトバンククリエイティブ、2002 年

[Cel01]

Joe Celko. Joe Celko's SQL for smarties. Morgan Kaufmann, Second edition, 1999.
秋田昌幸 訳、『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 年

[Cel04]

Joe Celko. Joe Celko's Trees and Hierarchies in SQL for Smarties. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[Cel05]

Joe Celko. Joe Celko's SQL Programming Style. Morgan Kaufmann Publishers, San Francisco, CA, 2005.

[Cod70]

Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM. 13[6]:377 - 387, 1970, June.

[Dat94]

C.J.Date. An introduction to database systems. Addison Wesley, 6th edition, 1994.
藤原譲 監訳、『データベースシステム概論』、丸善株式会社、1997 年

[Enc06]

エンコアコンサルティング 著、『データベースパフォーマンスアップの教科書—基本原理編』、翔泳社、2006 年

[Eva03]

Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Longman, Reading, MA, First, 2003.

今関剛 監訳、和智右桂、牧野祐子 訳、『エリック・エヴァンスのドメイン駆動設計』、翔泳社、2011年

[Fow03]

Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Longman, Reading, MA, 2003.

長瀬嘉秀 監訳、株式会社テクノロジックアート 訳、『エンタープライズアプリケーションアーキテクチャパターン』、翔泳社、2005年

[GP03]

Peter Gulutzan and Trudy Pelzer. SQL Performance Tuning. Addison-Wesley, Reading, MA, 2003.

[Gla92]

Robert L. Glass. Facts and Fallacies of Software Engineering. Addison-Wesley Professional, Boston, MA, 1992.

山浦恒央 訳、『ソフトウェア開発—55の真実と10のウソ』、日経BP社、2004年

[Gol91]

David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Comput. Surv.. :5–48, 1991, March.

[HLV05]

Michael Howard, David LeBlanc, and John Viega. 19 Deadly Sins of Software Security. McGraw-Hill, Emeryville, CA, 2005.

[HT00]

Andrew Hunt and David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, Reading, MA, 2000.

村上雅章 翻訳、『達人プログラマー』、ピアソンエデュケーション、2000年

[IPA10]

独立行政法人情報処理推進機構（IPA）著、『安全なSQLの呼び出し方』http://www.ipa.go.jp/security/vuln/documents/website_security_sql.pdf、2012年

[JST12]

JSTQB（Japan Software Testing Qualifications Board）訳、『ソフトウェアテスト標準用語集（日本語版）Version 2.1.J01』http://www.jstqb.jp/dl/JSTQB-glossary.V2.1.J01.pdf、2012年

[Lar04]

Craig Larman. Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, Englewood Cliffs, NJ, Third, 2004.

依田智夫、今野睦 監訳、依田光江 訳、『実践UML第3版—オブジェクト指向分析設計と反復型開発入門』、ピアソンエデュケーション、2007年

[Mic12]

ミック 著、『達人に学ぶDB設計徹底指南書』、翔泳社、2012年

[RTH11]

Sam Ruby, Dave Thomas, and David Heinemeier Hansson. Agile Web Development with Rails, 4th Edition. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2011.

前田修吾 訳、『Rails によるアジャイル Web アプリケーション開発 第 4 版』、オーム社、2011 年

[Spo05]

Joel Spolsky. Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity. Apress, 2004.

青木靖 訳、『Joel on Software』、オーム社、2005 年

[SZTZ08]

Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny, Arjen Lentz, and Derek J. Balling. High Performance MySQL、Second Edition. O'Reilly & Associates,Inc., Sebastopol, CA, Second, 2008.

伊藤直也、田中慎司、吉川英興 監訳、株式会社クイープ 訳、『実践ハイパフォーマンス MySQL 第 2 版』、オライリージャパン、2009 年

[Tok11]

徳丸浩 著、『体系的に学ぶ安全な Web アプリケーションの作り方—脆弱性が生まれる原理と対策の実践』、ソフトバンククリエイティブ、2011 年

[Tro06]

Vadim Tropashko. SQL Design Patterns. Rampant Techpress, Kittrell, NC, USA, 2006.

Rampant Tech Pr (2007/04)

索 引

記号・数字

%.....174
<=>.....153
2 値論理.....151
31 のフレーバー.....115
3 値論理.....151

A

ABS.....106
Active Record.....61, 265
Adjacency List.....15
Anemic Domain Model.....267
ANSI SQL 規格.....131
Apache Lucene.....182
Apache Solr.....182
Authentication.....210
AVG.....5, 149, 158, 194

B

B-Tree.....135
BCNF.....296
Bcrypt.....218
BFILE.....125
BLOB.....125
Boyce-Codd Normal Form.....296

C

Cartesian product.....188
CASE.....26, 84, 149, 193, 209
CHECK 制約.....92, 115, 301
Class Table Inheritance.....62
Closure Table.....20, 26
COALESCE.....55, 77, 84, 154
Common Table Expression.....18
compound key.....37
Concrete Table Inheritance.....61, 68
CONTEXT.....177
correlated subquery.....160
COUNT.....5, 11, 16, 17, 51, 158, 190
CRUD 操作.....265
cryptographic hash function.....211
CTE.....19
CTXCAT.....177
CTXRULE.....178
CTXXPATH.....177

D

DAO.....265
Data Definition Language.....xvii
Data Manipulation Language.....xvii
database fixture.....270
DDL.....xvii
DECIMAL.....108

DEFAULT 154
denormalization 8
derived table 161
dictionary attack 213
DISTINCT 159, 190
DKNF 301
DML xvii
DOMAIN 56, 113
Domain-Key Normal Form 301
Don't Repeat Yourself 271
DRY 原則 271
dual-mode principle 220

E

EAV 51, 54, 58
EAV 属性 65
Entity-Attribute-Value 54
ENUM 112
ER 図 ..... xx, xxi, 254
ETL 114
Explain 137
exponent 107

F

FILESTREAM 125
FLOAT データ型 104
foreign key 34
fraction 107
functional dependency 160

G

GIN 179
Globally Unique IDentifier 240
GROUP BY 10, 51, 156, 159, 189, 239, 270
GROUP_CONCAT 163
GUID 240

H

HTTPS 216

I

Identification 210
ID リクワイアド 39
IEEE 754 107
IEEE 754 標準 104
inapplicable 146
INNER JOIN 58
intersection table 9
IN 述語 83, 231
IS DISTINCT FROM 152
IS NULL 152
ISNULL 154
isolation 259
isolation testing 259
Issue クラス図 52

J

JOIN 5, 162
join dependency 301

L

LAST_INSERT_ID 22, 39, 93
LIKE 139, 174

LIMIT 170, 239
LOAD_FILE 126
LONG RAW 126

M

MATCH 176
MAX 157
Measure 137
MEDIUMBLOB 126
MENTOR 136, 291
migrations 257
MVC 264

N

negative test 260
Nested Set 20, 22
Nominate 139
normal form 292
normalization 289
NOT NULL 55
NOT NULL 制約 153
null 146
NULLIF 84
NULL とリレーション 150
NUMERIC 108
NVL 154

O

ON DELETE 17, 49
ON DELETE CASCADE 17
ON UPDATE 49, 117, 237
ON UPDATE CASCADE 117, 237
Optimize 140
ORDER BY 117, 133, 166, 239, 290
ORM 40, 41, 73, 200, 227, 248, 256, 260, 265
OUTER JOIN 58, 70, 162

P

Pagination 239
Path Enumeration 20
PBKDF2 218
Polymorphic Associations 68
precision 108
primary key 34, 40, 88, 93, 160, 238, 291
pseudo key 34, 238, 298

Q

QA 254
QC 254
QEP 138
Quality Assurance 254
Quality Control 254, 258
Query Execution Plan 138

R

RAND 166, 217
RANDOM 166
Rebuild 140
REGEXP 述語 175
relation 289
relationship 255, 289
repeating group 293
ROW_NUMBER 170, 239

S

scale 108
seed data 257
selectivity 134
Serialized LOB 64
SHA-256 212
SHA2 212
sign 107
SIMILAR TO 174
Single Table Inheritance 60
Single-Value Rule 157
Sphinx Search 181
SQL Injection 219
SQL-99 標準 18, 152
SQL:2003 ..... 34, 239
SQL アンチパターン xvi
SQL インジェクション 219
SUM 5, 16, 20, 106, 149, 158, 190, 193
surrogate key 34

T

Technical Debt 251
Test 140
the law of parsimony 191
TSVECTOR 179

U

UDT 56, 113
UNION 62, 76, 93, 192
UNIQUE 36, 75
UNIQUE キー制約 40
UNIQUE 制約 37
Universally Unique Identifier 240
unknown 146
UPDATE ステートメント生成 195
USING 38
UUID 240

W

window function 170, 239
WITH 18
WorldPress.com 96

あ行

アイソレーションテスト 259
曖昧なグループ 156
アクティブレコード 61, 265
アドホック 201
アトミック 173
暗号学的ハッシュ関数 211
アンチパターン xvii
アンチパターンの共通パターン 87
アンビギュアスグループ 156
暗黙の列 200
一意性制約 255
一時パスワード 216
入れ子集合 20, 22
インターセクション 9
インデックス 5, 130, 166, 174
インデックスショットガン 134
インプリシットカラム 200
ウィンドウ関数 170, 239
エイリアス 193, 197
エスケープ 223
エドガー・F・コッド 52, 150

エンティティ・アトリビュート・バリュー 51, 54, 58
オープンスキーマ 54
同じことを繰り返さない 271

か行

外交特権 252
外部キー 34, 69
外部キー嫌い 47
外部キー制約 10, 14, 18, 44, 73, 118
外部結合 25, 58, 70, 162
仮数部 107
カスケード更新 48
カバーリングインデックス 139
カプセル化 269, 271, 276
関係 289
関数従属性 160
関連 255, 289
キーレスエントリ 47
疑似キー 34, 238, 298
疑似キー潔癖症 236, 238
技術的負債 251, 272
疑似乱数 166
基底テーブル 78
キャッシュメモリ 140
行 57
競合状態 39, 237
凝集度 267
共通テーブル式 19
行番号 239
恐怖の unknown 149
クエリ xx
クエリ実行計画 138
臭いものに蓋 244, 246
具象テーブル継承 61, 68
クラステーブル継承 62
グラフ構造 31
繰返しグループ 293
クリス・J・デイト 150
グローバル一意識別子 240
経路列挙 20
結合 5, 162
結合従属性 301
欠番 167, 168, 236
交差テーブル 3, 9, 11, 37, 74, 255, 298
構築済みクエリ 226
コード生成 195
コードレビュー 234
コントローラ 264

さ行

サーティワンフレーバー 115
再帰クエリ 19
サブタイプ 60, 78
参照整合性 10, 44, 73, 118
参照整合性制約 56, 69, 255, 292
参照テーブル 45, 116, 255
シー・ノー・エビル 244, 246
シーケンス 39, 93
シードデータ 257
ジェイウォーク 3, 7
辞書攻撃 213
指数部 107
自然キー 36, 41, 239
実体関連図 xx
シャーディング 96, 97

従属 290
従属テーブル 86, 87, 97, 99, 255
シュードキー・ニートフリーク 236, 238
集約関数 156
主キー 34, 35, 40, 88, 93, 160, 238, 291
シリアライズ LOB 64
シングルテーブル継承 60
信号無視 3, 7
垂直パーティショニング 97
水平パーティショニング 97
スキーマ進化ツール 257
スキーマレス 54
スケール 108
スタブ 276
ステートメント xx
ストアドプロシージャ 183, 226, 255
スパゲッティクエリ 190
スロークエリログ 137
正規化 289
正規形 292
正規表現 174
整数 240
精度 108
セキュア HTTP 216
節約の原則 191
選択性 134
全文検索 176
相関サブクエリ 160
双対原理 220
ソート 166, 192, 232
疎結合化 273, 276
素朴な木 18
ソルト 213

た行

第 1 正規形 293
第 2 正規形 293
第 3 正規形 295
第 4 正規形 298
第 5 正規形 299
第 6 正規形 301
代理キー 34
多対多 ..... xxi, 3, 40, 86 , 297
単一値の原則 157
ツリー構造 16
ディプロマティック・イミュニティ 252
データアクセスオブジェクト 265
データ設計の比較 30
データ操作言語 xvii
データ定義言語 xvii
データベースフィクスチャ 270
データへのメタデータの混入 70
テーブルスキャン 166, 174
デカルト積 188
適用不能 146
デザインパターン 264
テスティング 254
転置インデックス 183
テンポラリテーブル 162
導出テーブル 161
動的 SQL 220
トークン 217
独立 259
ドメイン 56, 113
ドメインキー正規形 301
ドメインモデル 273
ドメインモデル貧血症 267
トランザクション 121, 126, 238

とりあえずID 39
トリガー 185, 255, 301

な行

ナイーブツリー 18
内部結合 58
ナチュラルキー 36, 41, 239
名前 / 値ペア 54
認証 210
根 14
ノード 14

は行

葉 14
バージョン管理 254
バグ管理データベース図 xxiii
バスキン・ロビンス社 112
パターンマッチ 5
パターンマッチ述語 174
バックアップツール 122
パフォーマンス 174, 192, 291
汎用転置インデックス 179
非正規化 8
ビットマップインデックス 135
否定テスト 260
非等値結合 169
非葉ノード 14
ビュー 264
平文 208
品質管理 254, 258
品質保証 254
貧者のサーチエンジン 175
プアマンズ・サーチエンジン 175
ファントムファイル 124
フィア・オブ・ジ・アンノウン 149
ブートストラップデータ 257
複合キー 37, 39, 293
複数列属性 84, 85
符号 107
浮動小数点数の誤差累積 106
不明 146
ブランチ 261
プリペアドステートメント 147, 224, 230
プロトタイプ作成 271
プロファイリング 137
文書化 254
閉包テーブル 20, 26
ページネーション 239
別名 193, 197
ボイスコッド正規形 296
ポカヨケ 48, 202
ポリモーフィック関連 68, 72
本人識別 210

ま行

マイグレーション 257
マジックビーンズ 266, 271
魔法の豆 266, 271
幻のファイル 124
ママのSQLインジェクション 221
マルウェア 219
マルチカラムアトリビュート 84, 85
丸め誤差 107
無 146
無限精度 104
メタデータ大増殖 95

メタデータトリブル.....95
メタデータへのデータの混入.....91
モック.....276
モデル.....265
モデル・ビュー・コントローラ.....264
漏れのある抽象化.....270

や行

闇雲インデックス.....134
やりすぎに注意.....19
有限精度.....104
ユーザー定義型.....113
ユーザ定義型.....56
読み取り可能パスワード.....210
予約語 NULL.....146

ら行

ラウンディングエラー.....107
乱数.....166
ランダムセレクション.....167
リーダブルパスワード.....210
リーフ.....14
リレーショナル理論.....173, 289
リレーション.....289, 290
リレーションシップ.....255, 289
隣接リスト.....15
倫理.....211
ルート.....14
レインボーテーブル.....214
レガシーデータベース.....41

わ行

ワイルドカード.....98, 174, 198

● 著者紹介

Bill Karwin（ビル・カーウィン）

ソフトウェアエンジニアとして20年以上の経験があり、アプリケーション、ライブラリ、サーバー（PHP5用のZend Framework、InterBaseリレーショナルデータベース、Enhydra Javaアプリケーションサーバーなど）の開発とサポートに携わってきた。キャリアを通じて、自らの知識を他のプログラマーと共有し、成功と生産性向上を支援してきた。数多くの質問に答えるなかで、最も問題を生じさせやすいSQLの典型的なミスについての独自の見解を得るに至った。

● 監訳者紹介

和田 卓人（わだ たくと）

タワーズ・クエスト株式会社 取締役社長、プログラマ、テスト駆動開発者。学生時代にソフトウェア工学を学び、オブジェクト指向分析 / 設計に傾倒する。その後様々な縁に導かれソフトウェアパターンやXP（eXtreme Programming）を実践する人たちと出会い、後のテスト駆動開発の誕生を知る。テスト駆動開発に「完璧主義の呪い（完璧な設計を得るまではコードを書けないし良いシステムも出来ないという強迫観念）」を解いてもらってからは、文章を書いたり、講演を行ったり、ハンズオンイベントを開催するなどして、テスト駆動開発を広めようと努力している。今日もグリーンバンド（テスト駆動開発者の証）を左手に着け、テストと共にコードを書いている。『プログラマが知るべき97のこと』（オライリージャパン）監修。Twitterアカウントは（@t_wada）、Facebookとgithubは（twada）で活動している。

和田 省二（わだ しょうじ）

タワーズ・クエスト株式会社 代表取締役。多種多様な業種・業務形態に対してデータモデリングを行い、数多くのデータドメイン設計およびER図を作成する。その結果から、RDB論理設計・物理設計・インデックス設計を手掛ける。SQLについては、ストアドパッケージによるバッチ処理構築、パフォーマンスチューニング、教育等を幅広く手掛ける。『改訂新版SQL書き方ドリル』（技術評論社）共著。

● 訳者紹介

児島 修（こじま おさむ）

英日翻訳者。1970年生。ビジネス、IT、健康、スポーツなどの分野で活躍中。訳書に『Head First Rails ―頭とからだで覚えるRailsの基本』『ビューティフルテスティング ―ソフトウェアテストの美しい実践』『デザイニング・ウェブナビゲーション ―最適なユーザーエクスペリエンスの設計』（オライリージャパン刊）など。

SQL アンチパターン

2013年 1月25日 初版第1刷発行
2013年 3月15日 初版第3刷発行

著　　者　Bill Karwin（ビル・カーウィン）
監 訳 者　和田 卓人（わだ たくと）、和田 省二（わだ しょうじ）
訳　　者　児島 修（こじま おさむ）
発 行 人　ティム・オライリー
印刷・製本　日経印刷株式会社
発 行 所　株式会社オライリー・ジャパン
〒160-0002　東京都新宿区坂町26番地27　インテリジェントプラザビル1F
Tel（03）3356-5227
Fax（03）3356-5263
電子メール　japan@oreilly.co.jp
発 売 元　株式会社オーム社
〒101-8460　東京都千代田区神田錦町3-1
Tel（03）3233-0641（代表）
Fax（03）3233-3440

Printed in Japan（ISBN978-4-87311-589-4）
乱本、落丁の際はお取り替えいたします。

本書は著作権上の保護を受けています。本書の一部あるいは全部について、株式会社オライリー・ジャパンから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

ISBN978-4-87311-589-4

C3055 ¥3200E





SQL Antipatterns

SQLアンチパターン

本書はDB設計やSQL記述の際に避けるべき事柄を1章で1つ、25個紹介するという書籍です。リレーショナルデータベースを中心に据えたシステム開発には、様々な場面で陥りやすい失敗（アンチパターン）があります。本書はデータベース論理設計、データベース物理設計、クエリの記述、アプリケーション開発という4つのカテゴリに分け、それぞれの分野におけるアンチパターンを紹介し、失敗を避けるためのより良い方法を紹介します。複数の値を持つ属性や再帰的なツリー構造の格納から、小数値の丸めやNULLの扱いに起因する問題、全文検索やSQLインジェクション、MVCアーキテクチャなど、実践的かつ幅広いトピックを網羅します。データベースに関わるすべてのエンジニア必携の一冊です。

O'REILLY® 発行所／オライリー・ジャパン

Ohmsha

発売元／オーム社
定価（本体3,200円+税）

Pragmatic Bookshelf