プログラマのための SQL

第4版

すべてを知り尽くしたいあなたに

Joe Celko's SQL for Smarties:
Advanced SQL Programming, 4th Edition

著 ジョー・セルコ
監訳 ミック



SE
SHOEISHA

プログラマのための SQL 第4版

すべてを知り尽くしたいあなたに

Joe Celko's SQL for Smarties:
Advanced SQL Programming, 4th Edition

著 ジョー・セルコ
監訳 ミック

SE
SHOEISHA

本書内容に関するお問い合わせについて

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

- **ご質問される前に**

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

正誤表　https://www.shoeisha.co.jp/book/errata/

- **ご質問方法**

弊社Webサイトの「刊行物Q&A」をご利用ください。

刊行物Q&A　https://www.shoeisha.co.jp/book/qa/

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

- **回答について**

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

• ご質問に際してのご注意

本書の対象を越えるもの、記述個所を特定されないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、予めご了承ください。

• 郵便物送付先およびFAX番号

送付先住所　〒160-0006　東京都新宿区舟町 5

FAX番号　03-5362-3818

宛先　（株）翔泳社 愛読者サービスセンター

※本書に記載されたURL等は予告なく変更される場合があります。

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

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

※本書に記載されている会社名、製品名はそれぞれ各社の商標および登録商標です。

JOE CELKO'S SQL FOR SMARTIES: Advanced SQL Programming
by Joe Celko
ISBN: 978-0-12-382022-8
Copyright © 2011 by Elsevier Inc. All rights reserved.
This edition of JOE CELKO'S SQL FOR SMARTIES: Advanced SQL Programming by Joe Celko is published by arrangement with Elsevier Inc., a Delaware corporation having its principal place of business at

360 Park Avenue South, New York, NY 10010-1710, USA through
The English Agency (Japan) Ltd.

第4 版への序文

本書は、これまでの諸版と同様、現場に携わるDBエンジニアで、高度なプログラミングのチップスとテクニックを習得したいと思っている人に向けて書かれている。1 年程度、実務でSQL を使った経験がある読者層を対象としている。つまり本書は入門書ではない。そういうわけで、これまでの版に対して向けたような「初心者向けでない」という不満を、Amazonのレビューに書き込まないでほしい。重要なのでもう一度繰り返すが、本書は初心者向けではない。

本書の初版が刊行されたのは10 年前のことだった。本書は現場のDB エンジニアの間で、ちょっとした古典になった。私は、訪れた職場のプログラマの机の上や、書店のいたるところに本書が置かれているのを見た。本の著者にとって最大の賛辞は、本にペタペタと貼られた付箋である。それを見るたびに、ああ、本当に読んで、使ってくれているのだなぁ、と感じたものだ。

10 年で変わったこと

階層データベースとネットワークデータベースは、今でも大企業のミッションクリティカルなレガシーシステム上で動いている。リレーショナルデータベースのエンジニアは、IMSや伝統的なファイルがいまだフォーチュン500 企業で動いていることを認めたくないであろう。だが、SQLに基づくシステムがこの数十年で遂げた大躍進を誇りに思ってかまわない。我々はいまや、多くの新しいアプリケーションと重要な小規模のデータベースを持っている。

この10 年の変化といえば、ほかにもある。オブジェクト指向言語はその地位を確かなものにした。だが、次の10 年で関数型プログラミングにとって代わられるだろう。オブジェクト指向データベースおよびオブジェクトリレーショナルデータベースはニッチ市場を見つけたが、主流にはならなかった。2010 年のいま、XML の流行も終わった。技術的には、XML はデータを記述し、あるプラットフォームから別のプラットフォームへ移動させるための構文であるが、XML をサポートするツール群によって、検索とフォーマット変換もできる。INCITS H2（以前はANSI X3H2 データベース標準委員会と呼ばれていた機関）には、SQL/XML の下部委員会があって、SQLとXMLの親和性が高いことを認めている。

データウェアハウスはもはや大企業だけの贅沢品ではなくなった。ハードウェアとソフトウェアの低価格化が進んだことで、中規模の企業であってもこの技術を利用できる。そして、OLAP 用のクエリを書くことは、OLTPのクエリを書くこととはまったく違う。早晩、OLAP用の「達人」本が必要とされるであろう。

オープンソースデータベースは活況を呈しており、ますますSQL標準への適合性を高めている。LAMP（Linux、Apache、MySQL、Python/PHP）は多くのWebサイトで利用されている。Ingres、PostgreSQL、Firebirdといった製品も、ANSIのSQL-92やSQL-99の大部分の機能を搭載しており、中にはSQL:2003の機能を持っているものも見られる。

カラム指向データベース、並列処理、楽観的同時実行制御といった先進的な機能も、すでに実験室の中だけでなく、商用製品に実装されつつある。

SQL 標準は常に変化してきたが、しかし必ずしも良い方向にではなかった。よりリレーショナルで集合指向的な変化があった一方で、明らかに手続き的で非リレーショナルなデータを、ファイルシステムのモデルに基づいて処理する方向に向かう変化もあった。

ディヴィッド・マクガヴァランの言葉を借りれば、「標準化委員会は、どんな新機能でもすぐに標準化しようとする」のである。彼はまったく正しいと思う。だがそうした混乱を考慮に入れても、ANSI/ISO のSQL-92 は、いかなるデータベースでも動作する共通的なサブセットである。実際1 年前には、米国政府はSQL-99を「進行中の標準」と評して、SQL-92を連邦契約に対する適合条件として要求したぐらいだ。

SQL-92 の開発においては、FIPS-127 適合基準テスト【訳注1】があり、これによってすべてのベンダーは同じ方向を目指すことができた。残念なことに、クリントン政権はこのプログラムを止めてしまい、適合基準は迷走を始めた。Whitemarsh Information Systems社

のトップであるマイケル・M・ゴーマンは、INCITS H2 で20 年にわたり委員を務めた人物で、SQL の歴史について政治的側面からエッセイを書いている【訳注2】。これは一読に値する内容だ。

　今日、SQL-99は多くのプラットフォームで汎用的に利用できる標準である。しかし、ベンダーはSQL:2003の機能を急速にサポートしつつある。そのようなわけで、私としては、こうした新機能も大いに利用すべきだと考えている。

第4 版で追加された内容

第2 版において、理論的な解説を本書から削除し、拙著『Data and Databases』（Morgan Kaufmann，1999）に移した。本版においても、その路線を変える理由はない。また、木と階層構造を扱う技術についても、分量が非常に増えたため、一冊の本とした（拙著『Trees and Hierarchies in SQL for Smarties』）。本書でもこの技術について少し触れているが、詳細は前掲書を参照してほしい。

また、新人向けのプログラミング技術については拙著『SQL Programming Style』（Morgan Kaufmann、2005）を参照してほしい。本書はあくまで上級を目指すエンジニア向けであり、読者はすでに実務で製品固有の方言しか知らなかったり、ネイティブのプログラミング言語にちょっと付け足しただけではなく、本格的にSQLを使っていると想定している。また、読者は標準SQLを製品固有の方言に変換することも容易にできるぐらいのレベルには達していると想定している。

私はこれまで、さまざまな解法について、なぜそれがうまくいくのかの説明を行ってきた。それにより、背景にある原理についての読者の理解を助け、異なる状況においても応用できるようになってくれれば幸いだ。

本書には、多くの方からの直接的な、あるいはWeb上のニュースグループを通じた間接的な支援をいただいた。ここで全員に感謝を述べることはできないが、できるだけコードには、考えついた人の名前を付すよう努めた。とはいえ、漏れてしまっている人もいるか

もしれないので、ここでまとめて謝意を表しておく。以下の人々に感謝する。

　アーロン・バートランド、アレジャンドロ・メサ、アニス・セン、クレイグ・マリンズ（本書のこれまでの版について査読をしてくれた）、ダニエル・A・モーガン、デヴィッド・ポータス、デヴィッド・クレッセイ、ドーン・M・ウォルタイス、ドン・バールソン、エーランド・ゾマースコグ、イツァク・ベンガン、ジョン・ギルソン、クヌート・シュトルツェ、ケン・ヘンダーソン、ルイス・デヴィッドソン、ダン・グズマン、フーゴ・コーネリス、リチャード・レムレー、サージ・リュロー、スティーヴ・キャス、トム・モロー、トレース・アーヴィン、ヴァディム・トロパシュコ、プラメン・ラチェフ、ゲルトジャン・シュトリク……。これ以外にも、まだまだ名前を失念している方々もいると思うが、あわせて感謝する。

著者について

ジョー・セルコ（Joe Celko）はANSI/ISO SQL標準化委員会の委員を10年間務め、SQL-89 およびSQL-92 規格の策定に貢献。データベース分野の雑誌・書籍への寄稿は900 篇を超え、本書（原書）の版元のMorgan Kaufmannからは以下の8冊を著している。

- *SQL for Smarties* (1995, 1999, 2005, 2010)（本書）
- *SQL Puzzles and Answers* (1997, 2006)（邦訳：『SQLパズル：プログラミングが変わる書き方／考え方』ミック訳、翔泳社、2007）
- *Data and Databases* (1999)
- *Trees and Hierarchies in SQL for Smarties* (2004, 2012)
- *SQL Programming Style* (2005)
- *Analytics and OLAP in SQL* (2006)
- *Thinking in Sets* (2008)
- *Data, Measurements and Standards in SQL* (2009)

翻訳・監修にあたって　――ミック

本書は、米国データベース界を代表するエンジニアであるジョー・セルコ氏の著書『Joe Celko's SQL for Smarties』第4 版の邦訳である。SQLに関して、おそらく世界で最も網羅的かつ体系的に記述された本であり、また高度で実践的なテクニックを満載していることから、英語圏でSQLを学んだり仕事で使う人々にとっては、不動のバイブルとして位置づけられている本である。

原書第2版の邦訳は、2001年にピアソン・エデュケーションから刊行されたが、それから第3 版を飛ばして、久々の訳出となる。著者も述べるように、この間のSQLの進歩には目覚ましいものがあり、本書もSQLの新機能に対応するべく大幅なアップグレードが行われている。「新しいSQL」を学ぶすべての人にとっての導きの書となるに相応しい重厚な内容であり、本書を訳出できたことは、一人のDBエンジニアとして非常に嬉しく思う。

さて、本書を読む人々のために、ここで本書の内容について簡単なガイドを行いたい。本書は、全39 章からなるが、大きく前半と後半に分けられる。前半は第12 章まで、後半は第13 章以降である。前半は主にSQLおよびリレーショナルデータベースの基礎的な概念や理論的枠組みについての解説が行われる。後半は、それを踏まえたうえでSQLの持つ多彩な機能の解説と高度な応用テクニックが紹介される。基本的には第1章から順に読んでいくシーケンシャルな読み方でかまわないが、もし読者が手っ取り早くSQLのテクニック

を知りたい、あるいは前半をパラパラめくってみて、内容をある程度承知していると感じたならば、第13 章から読み始めて、適宜必要な前半を参照する、という読み方をしてもよい。第13章は正確にはSQLのテクニックを解説した章ではないが、必ずすべての読者に読んでもらいたい。「本書の中で最も重要な章は？」と訊ねられたら、迷わず第13章と答えるだろう。

第14章までの前半では、テーブル、ビュー、プロシージャ、トリガーといったデータベースで扱われるオブジェクト、SQLのデータ型、および制約といった概念が扱われる。そのとき核となるのは、リレーショナルモデルと、ファイルシステムや手続き型に基礎を持つ言語との考え方の違いである。私たちはまず、こうした伝統的な言語で身につけた考え方をいったん忘れて、まったく新しい観点で物事を見ることを要求される（SQLに限らず、何の分野であれ、これが一番難しい奥義なのだが）。第13 章以降では、EXISTS、LIKE、結合、HAVING、ウィンドウ関数、CASE式といった、SQLの持つ強力な武器の多彩な応用方法を学んでいく。

なお、本書ではいわゆるモデリングや正規化についてはほとんど触れられない。正規化については第9章で取り上げているが、主に理論的解説にとどまり、実践的なケーススタディなどは扱われない。また、トランザクションや同時実行制御についても、第2章で簡単に触れられるのみであるため、こうしたテーマについては別に適切な書籍にあたってもらいたい。

また、本書は原則として標準SQL に準拠する構文で書かれているため、基本的には特定のDBMSに依存しないSQLを学ぶことができ

る。「基本的には」と留保を付けたのは、標準SQL に対する準拠のレベルは実装によってバラつきがあり、特定のDBMSではそのままの形では使えない構文も含まれているからだ。そのような場合は、適宜訳注にて説明を補っているので参考にしてもらいたい。願わくば、次の版ではこのような訳注の数が減ってほしいものだ。ちなみに、本書のサンプルコードをそのままの形で最も多く実行できる実装は、おそらくPostgreSQL であろう（翻訳時に動作確認環境としてPostgreSQL 9.1.1を使用した）。

ただし、例外的にストアドプロシージャやファンクションについては、Oracle Database のPL/SQLによる構文を掲載している。この理由は、標準SQLによる構文はどの実装でも動作しないため、原著に掲載されているプロシージャやファンクションのサンプルコードを動かせる実装がなかったためである。そこで本書では、動かないコードを載せるよりは実際に動くコードのほうがよいと考え、シェアの高いDBMSであるOracle DatabaseのPL/SQLを採用した（悪いことに、プロシージャやファンクションの構文はDBMS間でもバラバラである）。

最初に述べたように、本書はSQLの教科書としては疑いなく最高峰の水準である。だがそれだけに、本書を読み進めるのに難しさを感じる人もいるかもしれない。その場合は、あわせて読むことを推奨したい本がある。1つは、同じ著者による『SQLパズルは、同じ著者による『SQLパズル第2版』（翔泳社、2007）である。これは、さまざまなパズルを通じてSQLのテクニックを身につけるという、いわば「練習問題集」にあたる本である。もう1 つが、拙著『達人に

学ぶSQL 徹底指南書』（翔泳社、2008）である。これは、もともと本書の理解を助けるための「副読本」として書いたものである。肝心の教科書の登場が一番遅れてしまったが、ともあれ、これで教科書、問題集、副読本が揃ったことになる。

最後に、本書を買うかどうか迷っている人に対して、判断の指針を与えたい。もしパラパラとめくってみて、「これはちょっと、まだ歯が立たなそうだ」と感じたならば、いったん本書を棚に戻し力を蓄えてから戻ってくる、というのは賢明な判断である。著者も言うように、本書はお世辞にも初心者に優しい書き方はしていない。だが、もし本書の値段を見て躊躇しているならば、間違いなく本書は「買い」であると背中を押したい。というのも、見かけの値段に反して、実は本書は「安い」本だからだ。その理由は、本書をとことん読み込んでしまえば、あとは中途半端なSQLの参考書など1冊も読む必要がなくなるからだ。その点で、本書は非常に「コスパ」の良い本なのだ。

サンプルコードのダウンロード

本書のサンプルコードは、次のWebサイトからダウンロードできます。

https://www.shoeisha.co.jp/book/download/9784798128023

【訳注1】　FIPS（Federal Information Processing Standard：連邦情報処理標準）は、米国連邦政府公認の情報処理標準規

格。

【訳注2】　http://wiscorp.com/SQLStandards.html

目次

第4版への序文

著者について

翻訳・監修にあたって　――ミック

サンプルコードのダウンロード

1 データベース VS ファイルシステム

1.1　エンティティとしてのテーブル

1.2　関連としてのテーブル

1.3　行 VS レコード

1.4　列 VS フィールド

1.5　オブジェクトとしてのテーブル

2 トランザクションと同時実行制御

2.1　セッション

2.2　トランザクションとACID特性

2.2.1　原子性（Atomicity）

2.2.2　一貫性（Consistency）

2.2.3　独立性（Isolation）

2.2.4　耐久性（Durability）

2.3 同時実行制御
2.3.1 5つの現象
2.3.2 分離レベル
2.4 悲観的な同時実行制御
2.5 スナップショット分離と楽観的同時実行制御
2.6 論理的な同時実行制御
2.7 デッドロックとライブロック

3
スキーマレベルのオブジェクト

3.1 CREATE SCHEMA文
3.2 CREATE DOMAIN文
3.3 CREATE SEQUENCE文
3.4 CREATE ASSERTION文
3.4.1 スキーマレベルの制約をビューで実現する
3.4.2 制約のために主キーと表明を使う
3.5 キャラクタセットとそれに関連する諸概念
3.5.1 CREATE CHARACTER SET文
3.5.2 CREATE COLLATION文
3.5.3 CREATE TRANSLATION文

4
ロケータと特別な数

4.1 露出した物理的ロケータ

4.1.1 ROWIDと物理ディスクのアドレス
4.1.2 IDENTITY列
4.2 生成された識別子
4.2.1 GUID
4.2.2 UUID
4.2.3 業界標準の一意識別子
4.2.4 国防総省のUID
4.2.5 検証するための情報源
4.3 重複行について
4.4 シーケンス生成関数
4.4.1 一意な値の生成
4.4.2 欠番のある連番
4.5 事前に割り当てられる値
4.6 特別な数列
4.6.1 数列テーブル
4.6.2 素数
4.6.3 順序のランダムな数列
4.6.4 その他の数列
5
基底テーブルとそれに関連する要素
5.1 CREATE TABLE文
5.1.1 列制約
5.1.2 DEFAULT句
5.1.3 NOT NULL制約
5.1.4 CHECK制約

5.1.5　UNIQUE制約とPRIMARY KEY制約
5.1.6　REFERENCES句
5.2　入れ子のUNIQUE制約
5.2.1　オーバーラップするキー
5.2.2　一列の一意性と複数列の一意性
5.3　CREATE ASSERTION文
5.4　一時テーブル
5.4.1　一時テーブルの宣言
5.5　テーブルを操作する
5.5.1　DROP TABLE文
5.5.2　ALTER TABLE文
5.6　属性分割はしてはいけない
5.6.1　テーブルレベルの属性分割
5.6.2　行レベルの属性分割
5.7　DDLにおけるモデリングクラスの階層
5.8　その他のスキーマオブジェクト
5.8.1　スキーマテーブル
5.9　CREATE DOMAIN文
5.10　CREATE TRIGGER文
5.11　CREATE PROCEDURE文

6

手続き型プログラミング、半手続き型プログラミング、宣言型プログラミング

6.1　ソフトウェア開発の基礎

6.2 凝集度

6.3 結合度

6.4 大いなる跳躍

6.4.1 ありがちな間違い

6.4.2 一歩を踏み出せ

6.5 半手続き型への書き換え

6.5.1 データテーブル対ジェネレータコード

6.5.2 参照を計算で置き換える

6.5.3 フィボナッチ数列

6.6 述語で使うための関数

6.7 手続き VS 論理的分解

6.7.1 手続き的分解による解法

6.7.2 論理的分解による解法

7

手続き型の遺産

7.1 CREATE PROCEDURE文

7.2 CREATE TRIGGER文

7.3 カーソル

7.3.1 DECLARE CURSOR文

7.3.2 ORDER BY句

7.3.3 OPEN文

7.3.4 FETCH文

7.3.5 CLOSE文

7.3.6 DEALLOCATE文

7.3.7 カーソルの使い方

7.3.8　位置付けられたUPDATE文またはDELETE文
7.4　シーケンス
7.5　生成列
7.6　テーブル関数
8
補助テーブル
8.1　数列テーブル
8.1.1　リストの列挙
8.1.2　数列を循環数列にマッピングする
8.1.3　ループを置き換える
8.2　参照補助テーブル
8.2.1　単純な読み替えを行う補助テーブル
8.2.2　複数回の読み替えを行う補助テーブル
8.2.3　複数のパラメータを持つ補助テーブル
8.2.4　範囲補助テーブル
8.2.5　階層補助テーブル
8.2.6　単一参照テーブル
8.3　補助関数テーブル
8.3.1　補助テーブルを使った逆関数
8.3.2　補助関数テーブルを使った内挿
8.4　グローバル定数テーブル
9
正規化

9.1 関数従属性と多値従属性
9.2 第1正規形（1NF）
9.2.1 繰り返しグループについての注意
9.3 第2正規形（2NF）
9.4 第3正規形（3NF）
9.5 基本キー正規形（EKNF）
9.6 ボイス－コッド正規形（BCNF）
9.7 第4正規形（4NF）
9.8 第5正規形（5NF）
9.9 ドメイン－キー正規形（DKNF）
9.10 正規化を行うにあたっての実践的なヒント
9.11 キーのデータ型
9.11.1 自然キー
9.11.2 人工的なキー
9.11.3 露出した物理的ロケータ
9.12 非正規化に対する実践的なヒント

10 数値型

10.1 数値型
10.1.1 BIT、BYTE、BOOLEANデータ型
10.2 数値型の型変換
10.2.1 丸めと切り捨て
10.2.2 CAST関数
10.3 四則演算

10.4 算術とNULL
10.5 値からNULL、NULLから値への変換
10.5.1 NULLIF関数
10.5.2 COALESCE関数
10.6 数学関数
10.6.1 数論演算子
10.6.2 指数関数
10.6.3 小数を扱う関数
10.6.4 数値から文字への変換
10.7 IPアドレス
10.7.1 CHAR(39)形式での格納
10.7.2 バイナリ形式での格納
10.7.3 独立したSMALLINTでの格納

11 時間型

11.1 カレンダー標準についての注意事項
11.2 SQLの時間型
11.2.1 データベース内部での時間の表現
11.2.2 表示フォーマットの標準
11.2.3 タイムスタンプの扱い
11.2.4 時間の扱い
11.2.5 タイムゾーンとサマータイム
11.3 INTERVAL型
11.4 時間の計算
11.5 時間データモデルの性質

11.5.1　期間のモデリング
11.5.2　期間同士の関係

12
文字列型

12.1　SQLの文字列における問題
12.1.1　文字列の同値性に関する問題
12.1.2　文字列の順序に関する問題
12.1.3　文字列をグループ化するときの問題
12.2　標準の文字列関数
12.3　一般的なベンダー拡張
12.3.1　音声マッチング
12.4　カッター分類法
12.5　関数を入れ子にした文字列置換

13
NULL：SQL における失われたデータ

13.1　空のテーブルと失われたテーブル
13.2　列における失われた値
13.3　文脈と失われた値
13.4　NULLの比較
13.5　NULLと論理
13.5.1　サブクエリの述語におけるNULL
13.5.2　論理値述語

13.6 数学とNULL
13.7 関数とNULL
13.8 NULLとホスト言語
13.9 設計上のNULLの扱いに関するアドバイス
13.9.1 ホスト言語においてもNULLを避けるべし
13.10 NULLの複数の意味を使い分ける方法

14
複数列のデータ要素

14.1 距離関数
14.2 IPv4アドレスの格納方法
14.2.1 単純なVARCHAR(15)列で格納する方法
14.2.2 1つの整数列で格納する方法
14.2.3 4つのSMALLINT列を使う方法
14.3 IPv6 アドレスの格納方法
14.3.1 単一のCHAR(32)列
14.4 通貨と他の単位との変換
14.5 社会保障番号
14.6 有理数

15
テーブルの操作

15.1 DELETE FROM文
15.1.1 DELETE FROM句

15.1.2 WHERE句
15.1.3 別のテーブルに基づいた削除
15.1.4 同一テーブルのデータを削除する
15.1.5 参照整合性制約を使わずに複数のテーブルから削除する

15.2 INSERT INTO文

15.2.1 INSERT INTO句
15.2.2 INSERTの性質
15.2.3 バルクロードとデータ抽出ツール

15.3 UPDATE文

15.3.1 UPDATE句
15.3.2 WHERE句
15.3.3 SET句
15.3.4 別のテーブルを使った更新
15.3.5 UPDATE文でCASE式を使う

15.4 ベンダー拡張が持つ欠陥について

15.5 MERGE文

16

比較演算子またはシータ演算子

16.1 型変換

16.1.1 データ表示フォーマット
16.1.2 他の表示フォーマット

16.2 SQLにおける行比較

16.3 IS [NOT] DISTINCT FROM演算子

17

評価述語

17.1 IS NULL

17.1.1 NULLが発生する理由

17.2 IS [NOT] {TRUE | FALSE | UNKNOWN} 述語

17.3 IS [NOT] NORMALIZED述語

18

CASE 式

18.1 CASE式

18.1.1 COALESCE関数とNULLIF関数

18.1.2 集約関数とともにCASE式を使う

18.1.3 CASE式、CHECK制約、条件法

18.2 ローゼンシュタインの特性関数

18.3 行のソート

19

LIKE 述語とSIMILAR TO 述語

19.1 文字列パターンのトリック

19.2 NULLおよび空文字列の結果

19.3 LIKEは同値性を意味するのではない

19.4 結合ではLIKE述語を避ける

19.5 CASE式とLIKEの検索条件

19.6 SIMILAR TO述語
19.7 文字列のトリック
19.7.1 文字列の中身を調べる
19.7.2 文字列の検索 VS 文字列の宣言
19.7.3 文字列にインデックスを作る

20
BETWEEN 述語とOVERLAPS 述語

20.1 BETWEEN述語
20.1.1 入力がNULLの場合の結果
20.1.2 入力が空集合の場合の結果
20.1.3 BETWEENを使ったプログラミングの小技
20.2 OVERLAPS述語
20.2.1 時間の期間とOVERLAPS述語

21
[NOT] IN 述語

21.1 IN述語の最適化
21.2 ORをIN述語で置き換える
21.3 NULLとIN述語
21.4 IN述語と参照整合性制約
21.5 IN述語とスカラサブクエリ

22

EXISTS 述語

22.1　EXISTSとNULL

22.2　EXISTSと内部結合

22.3　NOT EXISTSと外部結合

22.4　EXISTSと量化子

22.5　EXISTSと参照整合性制約

22.6　EXISTSと3値論理

23

量化子とサブクエリ述語

23.1　スカラサブクエリの比較

23.2　量化子と失われたデータ

23.3　ALL述語と極値関数

23.4　UNIQUE述語

24

単純なSELECT 文

24.1　SELECT文の実行順序

24.2　1レベルのSELECT文

25

高度なSELECT

25.1　相関サブクエリ

25.2　標準的な構文の内部結合

25.3　外部結合

25.3.1　外部結合の歴史

25.3.2　NULLと外部結合

25.3.3　自然外部結合と条件付き外部結合

25.3.4　自己外部結合

25.3.5　2つ以上の外部結合

25.3.6　外部結合と集約関数

25.3.7　完全外部結合

25.4　UNION JOIN 演算子

25.5　スカラSELECT式

25.6　結合構文の新旧対決

25.7　条件付きの結合

25.7.1　在庫と注文

25.7.2　安定な結婚

25.7.3　ボールの箱詰め問題

25.8　コッドのT-結合

25.8.1　手続き型のアプローチ

26

仮想テーブル――ビュー、導出テーブル、共通表式、マテリアライズドクエリテーブル

26.1　クエリにおけるビュー

26.2　更新可能なビューと読み取り専用ビュー

26.3 ビューの分類

26.3.1 単一のテーブルからの射影および制限

26.3.2 計算列

26.3.3 コード化された列

26.3.4 グループ化されたビュー

26.3.5 UNIONを使ったビュー

26.3.6 結合を使ったビュー

26.3.7 入れ子のビュー

26.4 データベースエンジンにおけるビューの扱い方

26.4.1 ビューの列リスト

26.4.2 ビューの実体化

26.4.3 インライン展開

26.4.4 ポインタ構造

26.4.5 インデックスとビュー

26.5 WITH CHECK OPTION句

26.5.1 CHECK制約としてのWITH CHECK OPTION

26.6 ビューの削除

26.7 ビューと一時テーブルのどちらを使うかの判断についてのヒント

26.7.1 ビューの使い方

26.7.2 一時テーブルの使い方

26.7.3 ビューを使ったテーブルの展開

26.8 導出テーブルの使い方

26.8.1 FROM句で導出テーブルを使う

26.8.2 VALUES構築子を使った導出テーブル

26.9 共通表式

26.9.1 単純な共通表式

26.10 再帰的な共通表式

26.10.1 単純な加算

26.10.2 単純なツリー探索

26.11 マテリアライズドクエリテーブル

27

クエリによるデータの分割

27.1 被覆とパーティション

27.1.1 範囲による分割

27.1.2 単一列の範囲テーブル

27.1.3 関数による分割

27.1.4 数列による分割

27.1.5 ウィンドウによる分割

27.2 関係除算

27.2.1 剰余を持った除算

27.2.2 厳密な除算

27.2.3 パフォーマンスに関する注意

27.2.4 トッドの除算

27.2.5 結合を使った除算

27.2.6 集合演算を使った除算

27.3 レムレーの除算

27.4 RDBMSにおけるブール式

27.5 FIFOとLIFOの部分集合

28

グルーピング演算子

28.1 GROUP BY句

28.1.1 NULLとグループ

28.2 GROUP BY句とHAVING句

28.2.1 グループの特性とHAVING句

28.3 集約の階層

28.3.1 グループ化したビューによる集約の階層化

28.3.2 サブクエリによる集約の階層化

28.3.3 CASE式による集約の階層化

28.4 計算列によるグルーピング

28.5 GROUP BY句でペアを作る

28.6 GROUP BY句とソート

29

単純な集約関数

29.1 COUNT関数

29.1.1 DISTINCT付きの集約関数を最適化する

29.2 SUM関数

29.3 AVG関数

29.3.1 データがないグループを含む平均

29.3.2 複数列を対象にした平均

29.4 極値関数

29.4.1 簡単な極値関数

29.4.2 極値関数の一般化

29.4.3　複数の基準の極値関数

29.5　LIST関数

29.5.1　再帰共通表式とLIST関数

29.5.2　クロステーブルとLIST関数の併用

29.6　PRD関数

29.6.1　式によるPRD関数

29.6.2　対数によるPRD関数

29.7　ビット単位の集約関数

29.7.1　ビット単位のOR集約関数

29.7.2　ビット単位のAND集約関数

30

高度な集約、ウィンドウ関数、OLAP

30.1　スタースキーマ

30.2　GROUPING演算子

30.2.1　GROUP BY GROUPING SETS

30.2.2　ROLLUP

30.2.3　CUBE

30.2.4　SQLによるOLAPの例

30.3　ウィンドウ句

30.3.1　PARTITION BY句

30.3.2　ORDER BY句

30.3.3　ウィンドウフレーム句

30.4　ウィンドウ集約関数

30.5　順序関数

30.5.1　ROW_NUMBER

30.5.2　RANK関数とDENSE_RANK関数
30.5.3　PERCENT_RANK関数とCUME_DIST関数
30.5.4　ウィンドウ関数の使用例
30.6　ベンダー拡張
30.6.1　LEAD関数とLAG関数
30.6.2　FIRST_VALUE関数とLAST_VALUE関数
30.7　ウィンドウ関数小史

31

SQL における記述統計

31.1　最頻値
31.2　AVG関数
31.3　中央値
31.3.1　プログラミング問題としての中央値
31.3.2　特性関数を使った中央値
31.3.3　セルコの中央値
31.3.4　ウィンドウ関数を使った中央値
31.4　分散と標準偏差
31.5　平均偏差
31.6　累積統計
31.6.1　累積差分
31.6.2　累積率
31.6.3　順序関数
31.6.4　五分位と関連の指標
31.7　クロス表
31.7.1　CASE式によるクロス表

31.8　調和平均と幾何平均
31.9　SQLの記述統計における多変量
31.9.1　共分散
31.9.2　ピアソンのr
31.9.3　多変量の記述統計におけるNULL
31.10　SQL:2006における統計関数
31.10.1　分散、標準偏差、記述統計
31.10.2　相関

32

SQL における数列の扱い

32.1　大きさnのリージョンを見つける
32.2　リージョンに番号を割り振る
32.3　最も大きいリージョンを見つける
32.4　境界クエリ
32.5　ランを求めるクエリ
32.5.1　歯抜けの穴埋め
32.6　数列の歯抜け
32.7　数列の合計
32.8　リストの値を入れ替えたり、移動させる
32.9　シーケンスの始点と終点を見つける
32.10　数のリストを折り返して表示する
32.11　オーバーラップする被覆

33

SQL における配列

33.1 名前を持った列による擬似配列

33.2 添え字列を使った配列

33.3 SQLにおける行列操作

33.3.1 行列の相等性

33.3.2 行列の加法

33.3.3 行列の乗法

33.3.4 行列の転置

33.3.5 行列のソート

33.3.6 その他の行列操作

33.4 テーブルを配列に展開する

33.5 テーブル形式の配列を計算する

34

集合演算

34.1 UNIONとUNION ALL

34.1.1 UNIONの実行順序

34.1.2 UNIONとUNION ALLの混在

34.1.3 同じテーブルの列に対するUNION

34.2 INTERSECTとEXCEPT

34.2.1 NULLと重複行がない場合のINTERSECTとEXCEPT

34.2.2 NULLと重複行に対するINTERSECTとEXCEPT

35

部分集合

35.1 テーブルにおけるN番目ごとの行

35.2 テーブルからランダムな行を選択する

35.3 包含演算子

35.3.1 真部分集合演算子

35.3.2 テーブルの相等性

36

SQL で木と階層構造を扱う

36.1 隣接リストモデル

36.1.1 複雑な制約

36.1.2 手続き的な木の探索

36.1.3 テーブルの更新

36.2 経路列挙モデル

36.2.1 部分木とノードを見つける

36.2.2 木の深さと部分木を見つける

36.2.3 ノードと部分木の削除

36.2.4 整合性制約

36.3 入れ子集合モデル

36.3.1 座標からわかる情報

36.3.2 包含関係

36.3.3 同レベルのノード間の序列

36.3.4 階層集約

36.3.5 ノードと部分木を削除する

36.3.6 隣接リストモデルを入れ子集合モデルに変換する

36.4　木と階層構造を表現するその他のモデル

37

SQL におけるグラフ

37.1　グラフを隣接リストモデルで表す

37.1.1　SQLと隣接リストモデル

37.1.2　隣接行列モデル

37.2　グラフを入れ子集合モデルで表す

37.2.1　グラフのすべてのノード

37.2.2　経路の両端

37.2.3　到達可能ノード

37.2.4　エッジ

37.2.5　入次数と出次数

37.2.6　さまざまなタイプのノードを見つける

37.2.7　非循環グラフを入れ子集合に変換する

37.3　多角形の中の点

37.4　グラフ理論の参考文献

38

時間を扱うクエリ

38.1　時間の計算

38.2　個人的なカレンダー

38.3　期間のシーケンス

38.3.1　期間のシーケンスにおける欠落

38.3.2　連続的な期間

38.3.3　連続的な期間における失われた時間
38.3.4　日付の引き算
38.3.5　開始日と終了日
38.3.6　開始時間と終了時間
38.4　ユリウス通日
38.5　その他の時間関数
38.6　週
38.6.1　曜日によるソート
38.7　時間データのモデリング
38.7.1　期間のペア
38.8　カレンダー補助テーブル
38.8.1　イベントと日付
38.9　2000年問題
38.9.1　ゼロ
38.9.2　閏年
38.9.3　千年に一度
38.9.4　レガシーデータにおける奇妙な日付
38.9.5　2000年問題の影響

39

SQL の最適化

39.1　アクセス方法
39.1.1　シーケンシャルアクセス
39.1.2　ツリーインデックス
39.1.3　ハッシュインデックス
39.1.4　ビットベクトルインデックス

39.2　インデックス設計

39.2.1　シンプルな検索条件

39.2.2　シンプルな文字列式

39.2.3　シンプルな時間式

39.3　その他の注意事項

39.4　複数列に対するインデックスは慎重に

39.5　IN述語には要注意

39.6　UNIONを避ける

39.7　サブクエリを避けて結合を使う

39.8　SQL文の数を減らせ

39.9　ソートを避ける

39.10　クロス結合を避ける

39.11　汝、自らのオプティマイザを知れ

39.12　スキーマが変更されたら静的SQLをリコンパイルする

39.13　一時テーブルも時には役に立つ

39.14　統計情報を更新する

39.15　新機能は信用しない

参考文献

翻訳・監修を終えて　――ミック

索引

1

データベースVS ファイルシステム

Databases versus File Systems

> 我々を窮地に陥れるのは、我々が理解していない物事ではない。間違って理解している物事だ。
> ――アルテマス・ウォード（ウィリアム・グラハム・サムナー）、アメリカの作家、1834–67

データベース、特にRDBMSは、COBOL、FORTRAN、C、BASIC、PL/I、Javaその他あらゆる手続き型やオブジェクト指向のプログラミング言語が扱うファイルシステムとはまったく違う。かつてSQLは、「言語としての資格に欠ける（Scarcely Qualifies as a Language）」の略だと言われたものだ。たしかに、SQL は独自のI/O 機能を持たない。データを取り出してエンドユーザーに返すには、ホスト言語の助けを借りなければならない。

プログラミング言語は一般に、背景となるモデルを持っている。それゆえ、モデルを理解すれば、その言語をより深く理解できる。たとえば、FORTRAN は代数を基礎としている。これはFORTRAN が代数そのものである、という意味ではないが、もし代数を知っていれば、FORTRANを見てもそれほど奇妙に感じることはないだろう。代入式に式を書くこともできるし、見たことのない関数でも名前からどんな機能か推測することもできる。

プログラマという人種は、長い間、ほとんど毎日、プログラミング言語を使ってファイルを扱う仕事をしている。この「ファイル」という概念の設計は、紙の様式がもとになっているので、非常に物理的で、ホストのプログラミング言語に依存している概念だ。COBOL のファイルをFORTRAN で読むことはまずできない

し、その逆もまた然り。実際のところ、同じ言語で書かれたプログラム同士の間でも、ファイルを共有することはそう簡単ではない。

ファイルの最も原始的な形式は、ファイルの中で順序づけられ、物理的配置によってアクセスされる一連のレコード群である。ファイルをオープンし、最初のレコードを読み込み、EOF （end of file）に突き当たるまで次々にレコードを読み込む。このとき、一度に読み込み、操作できるレコードは1 行である。1 つのファイルに対して行った操作は、同じプログラム内に存在しないファイルには一切の影響を与えない。プログラムだけがファイルを変更することができる。

一方、SQL のモデルは、集合としてのデータであって、物理的なファイルではない。SQL における「作業単位」はスキーマ全体であって、個々のテーブルではない。集合というのは、学校で習ったあの数学的な抽象概念だ。集合に含まれる要素は、どれも同じタイプに属し、そして重要な性質は、順序を持たないことだ。集合に対して操作を行う場合、その操作は「一度に全要素」を対象に行われる。たとえば、正の整数の集合から奇数の部分集合を求める場合、答えはすべての奇数を含む単一の集合として得られる。奇数を1つずつ調べて順番に集合を作っていくようなことはしない。ただ、奇数を「2で割ったときの余りが1になる」という条件を満たす数として定義するだけだ。このルールはどんな整数を対象にしてもテスト可能で、分類できる。こうした理由から、集合指向モデルは並列処理に向いており、それはこのモデルの数多くの利点の1つである【訳注1】。

SQLは、FORTRANが完璧な代数言語であるほどには、完璧な集合指向言語ではない。それは、本書を読み進めればわかる。しかし、もしSQLで、この問題はどうやればいいのだろう、と疑問に思うことがあったら、集合の観点で考えるとどうやれば解けるだろうか、と考えてみることだ。そうすれば、きっと正しい解を得られるだろう。

さて、SQLは以下の3つのサブ言語から構成されている。

- DDL：データ宣言言語（Data Declaration Language）
- DML：データ操作言語（Data Manipulation Language）
- DCL：データ制御言語（Data Control Language）

DDLは、データベースに含まれる中身を定義し、データの整合性を保つ。ファイルにおいては、整合性制約、デフォルト値、他のテーブルとの関連などは一切定義されない。そのため、もしあるプログラムがファイルの中のデータをでたらめに書き換えてしまうと、次にそのデータを参照するプログラムは簡単にだまされる。年輩のプログラマに訊けば、FORTRAN でCOBOL のファイルを読んでしまい、エラーではなく間違った結果が出力された経験を語ってくれるだろう。ファイルは物理的に書き換えてしまうことが可能であり、データの論理的一貫性を保つ機能を持っていないため、そのようなことが起きる。SQLのテーブルは、そうではない。SQLでは、DDLをより詳細かつ厳密に記述することで、RDBMSはより賢く動くことができるようになる。また、DDLは常にDMLとDCLとともに動作する。SQLは統合された全体であって、分離された部分ではないのだ。

DML は、読者にとってもお馴染みのものだろう。選択（SELECT）、挿入（INSERT）、更新（UPDATE）、削除（DELETE）の4 つの操作を行う言語である。データをきちんと正規化し、きれいなスキーマを作っていれば、仕事はとても簡単になり、結果も信頼できるものになる。DML で記述されるSQL文は、手続き型のコードとは根本的に異なる。手続き型のコードは何度コンパイルしても同じ結果になるが、SQL はそうではない。DML のSQL 文が実行されるたびに、実行計画は変わる可能性がある。それは、データベースの状態が刻々と変化しているからだ。「すべては流れ行く。とどまるものは何もない」（プラトン『クラテュロス』より）

DCLは、データを保全するための言語ではない。アクセス制御のための言語だ【訳注2】。DCLはデータを暗号化したりしない。そもそも、暗号化は標準SQLには入っておらず、ベンダーの独自オプションだ。一般的なSQL の解説書ではDCL はそれほど重視されていないが、私はその扱いについては賛同できない。DCLはそれだけで一冊の小さな本を書けるほどの重要性がある。SQL の三脚のうち、最も見過ごされているといっていい。いつか、私自身が、そのような本を書きたいと思っている。

では、基本的な概念をざっと見ていこう。もしすでに読者が伝統的なファイルシステムを使ったデータ処理に慣れ親しんでいるならば、まずはそこで学んだこ

とを一度白紙に戻してほしい。

1. データベースのスキーマはファイルの寄せ集めではない。ファイルは、相互の関連を持っていない。ファイルの関連性はアプリケーション内で定義される。SQLは物理的な記憶領域については一切記述しないが、ファイルは、物理的に連続した記憶領域に基づいている。ファイルの祖先は、パンチカードである。次にそれを磁気テープで模倣し、それから初期のディスクドライブへと進化を遂げた。この観点を最初に取り上げたのは、ここからすべての問題が始まるからだ。
2. テーブルはファイルではない。テーブルはスキーマの一部だからだ。スキーマはSQLの基本的な作業単位であり、スキーマ内で同じ名前のテーブルを定義することはできない。ファイルシステムは、それが物理ドライブにマウントされたときにファイルへ名前を割り当てる。一方、テーブルはデータベース内において最初から名前を持っている。ファイルは物理的存在だが、テーブルは論理的存在である（ビュー、共通表式【訳注3】、クエリの結果セットなどもそうである）。
3. 行（row）はレコードではない。レコードはアプリケーションが読み込むことで初めて意味を持つ。レコードはシーケンシャルで、最初、最後、次、その前といった順序が意味を持つ。行はいかなる物理的な順序も持たない（`ORDER BY` はカーソルにおける句であって、SQLの一部ではない【訳注4】）。レコードはポインタや行番号のような物理的ロケータを持っている。行は関係モデルの意味でのキーを持っており、それはデータモデルにおける属性の部分集合によって与えられる一意性に基礎づけられている。だが、それを実現するメカニズムは特に決められているわけではなく、データベース製品によって大きく異なる。
4. 列（column）はフィールドではない。フィールドはそれを読み込むアプリケーションから意味を得る。またアプリケーションによっては複数の意味を持つことがある。フィールドはレコードの中で順序づけられており、データ型も制約もデフォルト値も持っていない。すなわち、列とフィールドの違いは、前者が能動的データであるのに対し、後者は受動的データである、とい

う点が大きな違いである！列はまた、NULLであることも許される。これはフィールドには存在しない概念だ。これも、フィールドが物理的な存在であるのに対し、列は論理的存在であることに起因する違いである。SQL ではテーブルに計算列を持つことができるが、ファイルではそれはできない。もし計算列を持ちたければ、それはファイルではなく、アプリケーション側で持つことになる。

テーブルとファイルの、もう1つの概念的な違いを挙げるなら、ファイルが通常は業務プロセス全体を扱うことだ。ファイルは、アプリケーションが1つの業務プロセスを処理するのに十分なデータを、その中に含んでいる必要がある。それゆえ、ファイルは「混合」データになりがちだ。たとえば「人事」ファイルとか、そんな感じの業務プロセスの名前を持つ。他方、テーブルは業務プロセスにおけるエンティティか関連【訳注5】のどちらかを表す。これはつまり、1つのファイルに含まれるデータをテーブルで持とうとした場合、多くの場合、複数のテーブルに分割して保持されるということだ。その意味で、テーブルは「純粋」なデータである――その中に何が入っているか、一言で表現できるような。「人事」ファイルの例を使えば、このデータをテーブルで保持しようとするなら、タイムカード、社員、プロジェクト、などなどの個別のテーブルに分解されることになる。

1.1

エンティティとしてのテーブル

エンティティ（実体）とは、物理的または概念的な、それ単独で意味を持つ「モノ」のことだ。たとえば、人間、売上げ、製品など。リレーショナルデータベースでは、エンティティは属性によって定義される。エンティティの個々のインスタンスは、テーブルにおける1つ1つの行である。属性は行における列である。属性の値はスカラ値【訳注6】である。

ユーザーにテーブルがエンティティの集合であることを思い出させるために私が好んで使うのが、テーブルの名前に集合名詞や複数形の名前を付けてエンティティの機能を表現する方法だ。たとえば、「Employees」は複数形なので、「Employee」よりも良いテーブル名だ。さらに良いのは「Personnel」である。これは集合名詞なので、個々人という心的イメージを一切喚起しない【訳注7】。この命名ルールは、メタデータに対するISO 11179標準にも準拠している。この点については、拙著『SQL Programming Style』（Morgan Kaufmann、2005）で詳細を論じているので、そちらを参照してほしい。

もし完全に同一の構造のテーブルが2 つある場合、それらはどちらも同じ種類の要素の集合である。だがこの場合、テーブルは1つであるべきだ！反対にファイルは、ストレージの物理的に分離された記憶単位であるため、同一のデータを異なるファイルで保持することが許される。テープやディスクのファイルは、手続きにおける各ステップを表現している。たとえば、編集前のデータから、編集済みのデータへ移行し、最後にはアーカイブデータになる。これらはすべて異なるファイルによって管理される。SQLでは、こうした状態遷移は1つのテーブルにおけるフラグで表すのが正しい【訳注8】。

1.2

関連としてのテーブル

　テーブルにおける関連は、1つ以上のエンティティテーブルを参照する列によって表現される。エンティティがなければ、関連は意味を持たないが、関連エンティティはそれ自身、独自の属性を持つこともある。たとえば、ショービジネスの契約には、俳優と代理人と雇用者が関係するだろう。このとき、支払い方法は、契約というエンティティそれ自身の属性であって、三者のいずれの属性でもない【訳注9】。つまり、ある列が別のテーブルを参照することが起きるのである。ファイルとフィールドには、そういうことは起きない。

1.3

行 VS レコード

行はレコードではない。レコードはそれを読み込むアプリケーションプログラムの中で定義される。行はデータベーススキーマの中で定義されるのであって、それが意味を持つためにプログラムを必要としない。フィールドの名前は、アプリケーションの`READ`文や`INPUT`文の中で定義されるのに対し、行はデータベーススキーマの中で名付けられる。同様に、`READ`文におけるフィールド名の物理的順序は絶対である（`READ a, b, c`は、`READ c, a, b`ではない。しかし、`SELECT a, b, c`は`SELECT c, a, b`と同じデータである）。

空のファイルは、どれも同じように見える。これはつまり、OSにおいてある名前のディレクトリのエントリはあるのだが、中は0バイト長という状態だ。だが空のテーブルは、1行も行がないとしても、なお列を持っており、制約やセキュリティ権限といった他の構造も保持している。

これは、テーブルが集合論の理論的モデルに即しているからだ。集合論では、空集合は他の集合と何ら変わるところのない集合である。SQLのテーブルと標準的な集合論における集合との違いは、集合論では空集合は1 つしか存在しないが、SQLでは、個々のテーブルは異なる構造を持つため、たとえ空であっても異なる集合として扱われることだ。

テーブルにおける行のもう1つの特徴は、すべての行が構造的に同一で、モデルにおいて「同じ種類の物」を表すことだ。ファイルシステムでは、レコードはサイズもデータ型もまちまちで、データストリーム中のフラグによって、プログラムにデータをどのように解釈するかを知らせている。典型的な例は、Pascalの`variant`レコードや、C言語の構造体、COBOLの`OCCURS`句だ。

COBOLのOCCURSキーワードと、Pascalの`variant`レコードは、プログラムに、今読んでいるレコードの中で、何回サブレコード構造が繰り返されるかを知らせるための数字を持っている。C言語の共用体は`variant`レコードではないが、同じ物理メモリにさまざまな型をマッピングする。たとえば、次のような文を考えよう。

```
union x {int ival; char j[4];} mystuff;
```

これは、mystuffをinteger型（大半のCコンパイラでは4バイトだが環境によって異なるので、このコードに移植性はない）または、4バイトの配列として定義する。どちらとして使うかは、mystuff.ivalとmystuff.j[0]のどちらを使うかで決まる。

だがそれ以上に両者が違うのは、ファイルはしばしば他のレコードの部分集合の合計（いわゆる「コントロールブレイクレポート」）を保持する点だ。ファイル内のレコードをどのように関係付けるかについて、特に制限はない——レコードは文字どおり、バイナリデータのストリームであり、その意味は読み出すプログラムが割り当てるのだ。

1.4

列 VS フィールド

レコード内のフィールドは、それを読み込むアプリケーションによって定義される。一方、テーブルの行における列は、データベーススキーマによって定義される。列のデータ型は、常にスカラ値である。

アプリケーションがREAD文やINPUT文で変数を読む順序は重要である。なぜなら、値がプログラムの変数に読み込まれるときの順序は、それによって決まるからだ。SQL では、列は列名だけによってアクセスされる。もちろん、SELECT *やINSERT INTO <table name>という列名を明記しない省略形は存在する。これは列名のリストを、テーブル定義で列が定義された物理的順序で展開する。しかし、これらはあくまで列名をずらずら並べる代わりの省略形にすぎない。

SQLにおけるNULLの使用もまた、この言語特有だ。フィールドは、このような失われた値のマーカーを、フィールド、レコード、またはファイル自身の一部としてサポートしない。それどころか、SQLにおけるDEFAULT 制約やCHECK 制約のような制約すら、フィールドは持っていない。

ファイルは受動的であり、アプリケーションプログラムが行う行為を大した反対もせず受け入れる。ファイルはまた、互いに独立している。それは、アプリケーションプログラムにファイルが結び付けられるときは、一度に1つのファイルが結び付けられるのであって、そのとき、他のファイルがどうなっているかは気にしないからである。

他方、データベースは能動的にすべてのデータの正しさを保つように動く。そのための道具は、トリガー、制約、そして宣言整合性制約である。

宣言整合性制約（DRI： Declarative Referential Integrity）は、1つのテーブルがもう1つのテーブル（同じテーブルであってもかまわないが）とデータの関係を保つ保証する機能である。それだけでなく、データベースに対する変更そのものを宣言整合性制約を通じて実装することも可能だ。たとえば、在庫にない商品を売ることはできないというビジネスルールを考えよう。

このルールは、在庫（Inventory）テーブルを参照するREFERENCES句を受注（Orders）テーブルに宣言することで実現できる。このとき、ON DELETE

CASCADEオプションも忘れないでほしい。あるいは、同じことをやる手段としては、トリガーの方が一般的である。トリガーはひとまとまりの手続き型のコードで、INSERT INTOやUPDATE文が実行される前、後、あるいはそうした文の代わりに実行することができる。宣言整合性制約によって実現できることはトリガーによって実現できるし、トリガーでは、それ以上のこともできる。

しかしながら、トリガーにはいくつかの問題もある。SQL-92で決められた標準的な構文はあるのだが、多くのベンダーはそれに準拠しておらず、実装依存のコードになってしまう。また、トリガーはオプティマイザに情報を渡せないため、パフォーマンス上の最適化を受けられないという難点もある。本節での例では、Orders テーブルにあるすべての製品番号が、Inventory テーブルにもあると仮定している。オプティマイザはクエリのEXISTS述語と結合の実行計画を立てる際に、その情報を利用することができる。トリガーの手続き型のコードでは、その関連を突き止めるうまいパース方法が存在しない。

SQL-92には、CREATE ASSERTION文という機能がある【訳注10】。表明（ASSERTION）は、スキーマ全体に対して制約条件を課すもので、CHECK制約とは異なるが、その違いはわずかだ。CHECK 制約は、それが付与されたテーブルに行が存在する場合に実行される。裏を返すと、テーブルが空の場合は、すべてのCHECK句が即座に真（TRUE）となる。したがって、Inventoryテーブルが絶対に空にならないことを保証するためには、次のように記述する必要がある。

```
CREATE TABLE Inventory
( ...
     CONSTRAINT inventory_not_empty
     CHECK ((SELECT COUNT(*) FROM Inventory) > 0),
  ... );
```

だが、CHECK 制約でこういう構文を使ってもうまくいかない。こういうときは、代わりに、ASSERTIONを使って書くことができるのだ。

```
CREATE ASSERTION Inventory_not_empty
CHECK ((SELECT COUNT(*) FROM Inventory) > 0);
```

これで狙いどおりの制約を実行できる。

1.5

オブジェクトとしてのテーブル

概念的には、テーブルはゼロ行以上の行の集合である。そして、行は1 つ以上の列の集合である。この階層は重要である。というのも、データベースに対する操作は、スキーマ、テーブル、行、列の各レベルで作用するからである。たとえば、DELETE FROM文は、行を削除するのであって、列を削除するわけではない。かつ、スキーマには基底テーブルも残す（DELETE FROM 文はスキーマオブジェクトを削除するわけではないのだ）。列を行から削除することはできない。

列はおのおのデータ型と制約を持っており、それによって、抽象的なドメインを実装している。テーブルが物理的にどう実装されているかは問題ではない。なぜなら、テーブルにアクセスする方法は、あくまでSQL経由だからであり、SQLはデータの物理的配置を気にしない。後の詳細はデータベースエンジンがよろしくやってくれるので、物理ファイルを扱うときのように、内部の詳細に気をわずらわせる必要はないのだ。実際には、データベース製品は1 つとして同じ内部構造を持っているものはないのだが、それでもなお、ユーザーは物理層を意識する必要がない。

ファイルシステムとパソコンに慣れたプログラマが犯しやすい概念上の間違いを2 つ紹介しよう。1つ目は、テーブルをファイルだと考えること、2つ目は、テーブルをスプレッドシートだと考えることである。テーブルはこのどちらとも違う。この基本的概念を理解していないと、のちのち面食らうことになるだろう。

テーブルがファイルだと想像することはたやすい。行はレコードで、列はフィールド――この連想は、データをSQLからホスト言語に移動させたときにもしっくりくる。データを表示したり加工したりするには、ホスト言語におけるデータ型とデータ構造に変換しなければならない。ホスト言語は自身の中にファイルシステムを組み込んでいる。

ファイルシステムとSQLを扱うときの大きな違いは、SQLがホスト言語と接するときの方法にある。ファイルシステムを使うとき、プログラムは、個々のファイルを開いたり閉じたりしなければならない。SQLでは、スキーマ全体がプログラムと接続されたり切り離されたりする1つの単位なのだ。ホストプログラムは、スキーマ内のすべてのテーブルや他のスキーマオブジェクトを参照したり操作する権

限を与えられてはいないかもしれない。だが、プログラムがアクセス権限を持つテーブルへの接続は、スキーマ全体への接続の一部に含まれている。

ファイル内のフィールドを定義するのは、プログラムだ。一方、スキーマ内の列を定義するのはSQL だ。FORTRAN はFORMAT 文とREAD 文を使ってデータをファイルから得る。同様に、COBOLもDATA DIVISIONでフィールドを定義し、READ文でデータをフェッチする。すべての3GL言語も同様である。コンセプトは同じで、構文とオプションが異なるだけだ。

ファイルシステムは、同じデータを、プログラムに応じて異なる名前で参照することができる。また、ファイルのレイアウトが変われば、そのファイルを使っているすべてのプログラムを修正しなければならない。ファイルが空であれば、他の空のファイルとまったく同じように見える。空のファイルを読み出そうと思えば、EOFフラグが上がって、プログラムは常に同じ行動を取る。だが、テーブル内の列名とデータ型は、データベーススキーマの中で定義されていて、合理的な範囲内であれば、テーブルはホスト言語が意識することなしに変えることができる。

ホスト言語が心配するべきは、データベースから自身の持つ変数へ、正しく値を受け渡せるかどうかだけなのだ。さて、ここで読者にちょっと聞いてみたいのだが、空集合がどういう扱いを受けるか、高校のときの数学を覚えているだろうか？ 空集合は正当な集合である。テーブルが空の場合であっても、なおテーブルは列とゼロ個の行を持つ。ここには例外を発生させるEOFフラグはない。なぜなら、テーブルにはそもそも「最終行」はないからだ。

ファイルとテーブルのもう1つの大きな違いは、テーブルと列には制約を付けることができることだ。制約は、個々のトランザクションが終わった後に、データベースについて満たされなければならないルールだ。この意味でデータベースは、従来の受動的なファイルシステムより、オブジェクトの集合に似ている。

テーブルはスプレッドシートでもない。たしかに両者は、ディスプレイや紙の上で見るとよく似ている。スプレッドシートでは、カーソルを操作することで行、列、セル、セルの集合にアクセスできる。だがテーブルには、こうしたカーソル操作の概念はない。スプレッドシートにおけるセルは、データだけでなく見

出しを保持することもできるし、スプレッドシートでは、行と列に本質的な違いはない。行列をそっくり入れ替えることもできるし、それでも妥当なデータが得られる。これはSQLのテーブルではあり得ないことだ。唯一、共通点と呼べるものを挙げるなら、スプレッドシートもまた、宣言型プログラミング言語だということだ。スプレッドシートは、図らずも非線形言語なのだ。通常のプログラミング言語は一方向にしか読むことができないが、スプレッドシートは、行、列、ページなど複数の方向から読むことが可能だ。

【訳注1】 一方でSQLの「一度に全要素」を操作対象とする特性は、欠点として現れることもある。SQLは計算の解を見つけたら途中で計算をやめる、ということが難しいため、組み合わせ問題などでは計算量が膨大になり、パフォーマンスが悪い。例としては第25章で取り上げる問題「安定な結婚」を参照。

【訳注2】 DCLには`GRANT`や`REVOKE`など権限を操作する文が含まれる。

【訳注3】 共通表式（Common Table Expression：CTE）はSQL:1999で導入された機能。クエリの先頭で`WITH`句によってビューを定義することで、そのクエリに限定したビューを利用することが可能になる。

【訳注4】 カーソルも標準SQLで定義されているという意味では広義のSQLの一部だが、ここで著者は「SQL」ということで集合操作のみを行うコア部分を想定している。

【訳注5】 多対多の関連を消去するために導入される関連エンティティのことを指している。

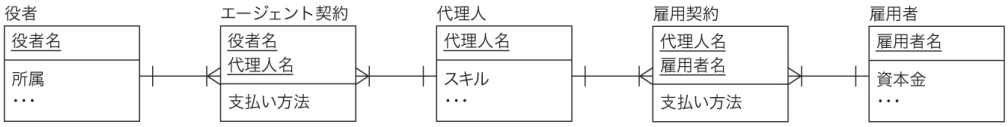
【訳注6】 スカラ値とは、それ以上分解不可能な原子的な値のこと。

【訳注7】 日本語の名詞では、集合名詞に対応する区分をあまり意識しないことと、複数形と単数形の使い分けもあいまいなため、この議論は少し日本人にはわかりにくい。要するに、テーブルが同じ種類の物の集合だ、という点をわかってもらえば十分である。

【訳注8】 この要請は、いささか理想的に過ぎると感じる人も多いだろう。実際には、古くなった行をあるテーブルからアーカイブ用のテーブルに移動させるような運用は、よく行われている。それは、結局のとこ

ろテーブルの行数が物理的なところでSQLのパフォーマンスに影響するからだ。著者はここで、そうした物理的制約をいったん脇に置いたうえで、論理的な原則を述べていることに注意してほしい。

【訳注9】　この三者のエンティティについて、ER図を描くと、下図のようになる。ここで、「役者–代理人」と「代理人–映画会社」は多対多の関連であるため、契約という関連エンティティが生じることになる。「支払い方法」という属性は、この契約エンティティにしか保持することはできない。



【訳注10】　表明をサポートしている実装は多くない。

2

トランザクションと同時実行制御

Transactions and Concurrency Control

太古の昔、人類は洞窟に住み、そしてバッチファイルシステムをメインフレームで動かしていた。その頃のトランザクション処理はシンプルなものだった。トランザクションファイルをマスタファイルから更新するバッチを組み、テープドライブからマスタファイルに対してバッチを実行すれば、トランザクションファイルはソートされ、編集された。この処理の出力が、新しいマスタファイルであり、古いマスタファイルとトランザクションファイルは磁気テープにログとして保存される。後は会社の地下室にある巨大な倉庫にテープを保存すれば任務完了だ。

やがて、ディスクドライブ、マルチユーザーシステム、そしてデータベースが誕生し、物事はもっと複雑になった。SQLもまた、事態を複雑にすることに手を貸している。しかしありがたいことに、DBMSがほとんど肩代わりしてくれているので、ユーザーは普段、その複雑さをあまり深く理解する必要はない。だが、DBMSが内部で行っていることを多少なりとも理解しておくと、いろいろと役に立つ局面も増えるはずだ。ここでは、その複雑さの一端を見てみることにしよう。

2.1

セッション

ユーザーセッションの概念の核は、これによってユーザーは初めてデータベースとの接続を確立できる、ということにある。これは、誰かに電話をかける行為に似ている。ただし電話と違うのは、データベースにつなぐときはパスワードが必要なことだ。標準SQLにおけるセッションを確立する構文は次のとおりだ。

```
CONNECT TO <connection target>
<connection target> ::=
<SQL-server name>
[AS <connection name>]
[USER <user name>]
| DEFAULT
```

もっとも、この構文はベンダー依存の部分が多く、SQL製品によって異なる点が多い。

いったんデータベースとの接続が確立されると、ユーザーはデータベースのすべての箇所にアクセスすることが可能になる（ただし権限を与えられている範囲に限るが）。セッション中、ユーザーは0個以上のトランザクションを実行することができる。データベースにおいて、ユーザーがINSERT、UPDATE、DELETE を実行するとき、ユーザーがそのトランザクションに対してCOMMITを行うまでは、そうした変更は永続的なものにはならない。もし変更を永続的なものにしたくなければ、ROLLBACKコマンドによって、データベースをトランザクションが実行される前の状態に戻すことができる。

2.2

トランザクションとACID 特性

トランザクションが満たすべき特徴について、覚えやすい略称がある。それがACID特性である。これはトランザクション処理システムが満たすべき4つの特性の頭文字を表している。

1. 原子性（Atomicity）
2. 一貫性（Consistency）
3. 独立性（Isolation）【訳注1】
4. 耐久性（Durability）

2.2.1　原子性（Atomicity）

原子性とは、データベースの中でトランザクション全体が永続的であるか、それとも永続的でないかの2 つの状態しか取らないことだ。標準SQL では、データはCOMMIT 文が成功したときに永続化される。ROLLBACK文は、トランザクションを破棄し、データベースをトランザクションが始まる前の（一貫した）状態に復元する。

COMMITやROLLBACKはユーザーが明示的に実行する場合もあるし、データベースエンジンがエラーを発見したときに、その対処として実行する場合もある。たいていのデータベースエンジンは、デフォルト設定のままであれば、そういう場合はROLLBACKを発行するはずだ。

原子性に従えば、たとえば100万行を挿入（INSERT）しようとして、そのうちの1行だけが整合性制約に違反していた場合でも、100万行すべてが拒否され、データベースは自動的にROLLBACKを発行する。

ここにはトレードオフがある。もし1 つの長いトランザクションを実行すると、ほんの些細なエラーによってすべてを台無しにしてしまう危険が生じる。一方、1つのセッションの中で細かく分割したトランザクションを実行すると、トランザクションの合間に他のユーザーがアクセスすることで、データに変更を加えて意図せぬ処理結果が生じ、読者を驚かせるかもしれない。

SQL:2006 ではSAVEPOINT が導入された。SAVEPOINT は、トランザクションにおける付箋みたいなものだ。トランザクションは、その実行の最中にSAVEPOINTを設定し、トランザクションは、そのチェックポイントまで限定的なロールバックを実行できる。SAVEPOINTは1つのトランザクションに複数設定することができ、任意のSAVEPOINTに戻ることができる。

先ほどの例を使うと、たとえば1,000 行単位でセーブポイントを設定するとしよう。すると、999,999行目のINSERTでエラーになったとしてロールバックが実行されても、データベースエンジンが破棄するのは、最後のセーブポイントの後に実行された処理だけである。そしてトランザクションは、コミットされていない状態（999,000行）に復元される。

セーブポイントの構文は、以下のとおりである。

```
<savepoint statement> ::= SAVEPOINT <savepoint specifier>
<savepoint specifier> ::= <savepoint name>
```

1つのトランザクションに設定できるセーブポイントの上限値は、実装依存である。セーブポイントは、入れ子にすることもできる。現在有効なレベルを見つけるには、次のようにする。

```
<savepoint level indication> ::=
NEW SAVEPOINT LEVEL | OLD SAVEPOINT LEVEL
```

セーブポイントを削除する構文は以下のとおりである。

```
<release savepoint statement> ::= RELEASE SAVEPOINT
<savepoint specifier>
```

COMMIT文は、当該のレベルで行われた処理、または一連のセーブポイントで行われた処理を確定する。

```
<commit statement> ::= COMMIT [WORK] [AND [NO] CHAIN]
```

同様に、ロールバックを使えば、トランザクション全体を破棄するか、特定のセーブポイントまで戻ることができる。

```
<rollback statement> ::= ROLLBACK [WORK] [AND [NO] CHAIN]
[<savepoint clause>]
<savepoint clause> ::= TO SAVEPOINT <savepoint specifier>
```

セーブポイントについては以上である。読者の使っている製品で、こうした機能が利用可能か、チェックしてみてほしい。セーブポイントの代替手段としては、トランザクションを細かく分割するか、データベースにロードする前に完璧にデータの整合性をチェックし、エラーを排除してくれるETL（Extract Transform Load）ツールを使うかのどちらかである。

2.2.2　一貫性（Consistency）

トランザクションが開始したとき、データベースは一貫した状態にある。そしてトランザクションが永続化されたときも、やはりデ

ータベースは一貫した状態にある。この「一貫した状態」という言葉の意味は、すべてのデータ整合性制約、参照整合性制約など、すべての制約が満たされた状態であることだ。

　といっても、これは別に、データベースがトランザクションを実行中に、非一貫的な状態をとってはならないということを意味しているわけではない。標準SQLでは、制約にDEFERRABLE やNOT DEFERRABLEオプションを付けて宣言することができる。いわゆる「遅延制約」の機能である。つまり、一貫性のルールは、あくまでセッションの終了時点で適用されるのだ。このため、トランザクションが複数の文を含んでいたり、他のテーブルに対してトリガーが実行されたりするときは、少しトリッキーに感じるかもしれない。

2.2.3　独立性（Isolation）

　ある1つのトランザクションは、他のいかなるトランザクションからも独立している。というのも、この性質は、トランザクションがあたかも互いに独立に、つまり直列に実行されたときと同じ結果になる、という意味だからだ。独立性は、直列性とも呼ばれるが、それは、この性質を保証する方法の1つが、バッチシステムでやるように、本当にトランザクションを直列に実行することだからである。もちろん、多くのトランザクションが並列で実行される現実のシステムにおいては、これは優れた解とは言えない。そのため、システムは、いかにしてトランザクション同士の干渉を防ぎ、直列のときと同じ結果を得られるかを考えなければならない。

　これが難しい問題なのは、あるトランザクションが、他のトランザクションがINSERTしたり、UPDATEしたり、DELETEしたデータを

参照するかしないかは一般に決まっていないからだ。この点については、トランザクション分離レベルについて述べている2.3.2項で詳しく見ていく。

2.2.4　耐久性（Durability）

　データベースは耐久性のある媒体に保存されている。そのため、データベースのプログラムが壊れたとしても、データベースそのものは保全される。さらに、データベースシステムが復旧した際には、データベースは一貫した状態に復元することができる。この性質を満たすために、データベースが持っているのが、ロギングとバックアップの機能である。この2つのメカニズムは、トランザクションの処理中にディスク書き込みが発生しているときにも動作している。

　もし、データベースに対して変更するユーザーが、一度に1人しかいないのなら話は簡単だ。しかし、データベースが必要とされる理由の1 つが、常にデータに対して複数のユーザーがアクセスするからなのだ。この問題に対処するために、同時実行制御について考える必要がある。

2.3

同時実行制御

同時実行制御は、複数のユーザーがデータベースを共有して使う環境で、いかにして互いに干渉せずにトランザクションを扱うか、ということを考える分野である。渋滞や交通事故をいかにして防ぐかが重要な点で、交通システムに似ているかもしれない。この問題を回避する方法の1 つは、データベースを使用できるユーザーを一度に1 人に制限することだ。この解決策の唯一の問題点は、他のユーザーが待たされる時間が長くなることである。読者は、何百万人ものユーザーが順番待ちをする銀行の窓口システムや飛行機の予約システムを想像できるだろうか？

2.3.1　5 つの現象

もしユーザーの発行するSQLが検索のみであるなら、ACID特性は放っておいても保たれる。問題は、2人以上のユーザーが同時に更新を行う場合だ。SQLモデルでは、1つのトランザクションが別のトランザクションに影響を与えるパターンには5種類ある。以下の説明は、ジム・グレイらによるドキュメントによっている【原注1】。オリジナルのANSIモデルでは、P1、P2、P3のみを扱っていた。その他の定義が最初に行われたのは、上記ドキュメントにおいてである。

- P0（ダーティライト）：トランザクションT1 がデータを変更するとする。別のトランザクションT2 が、T1がコミットかロールバックを実行する前にさらに同じデータを変更する。このとき、T1とT2 のどちらかがロールバックを実行すると、正しいデータの値は何であるべきか不明確になる。ダーティライトが悪い理由の

1つは、データベースの一貫性を損なうからだ。xとyの間に制約（たとえばx=y）があると仮定しよう。この場合、T1とT2は、単独で実行される限り、どちらのトランザクションも一貫性を維持する。しかしながら、2 つのトランザクションがxとyを異なる順序で書き換えると、この制約は簡単に破られてしまう【訳注2】。これが起きるのは、ダーティライトを許す場合だけである。

- P1（ダーティリード）：トランザクションT1 がある行を変更する。次に、T1 がコミットまたはロールバックをする前に、トランザクションT2がその行を読み出す。次に、T1がロールバックを実行すると、T2が読み出した値は、コミットされる前のものであり、ロールバック後にはもう存在していない。
- P2（反復不能読み取り）：トランザクションT1がある行を読む。次に、T1がコミットまたはロールバックする前に、トランザクションT2 がその行を変更するか削除する。すると、T1が再度同じ行を読み出そうとしても、T2による変更後の値を得るか、あるいは削除されたために行を発見することができない。「ファジーリード」とも呼ばれる。
- P3（ファントム）：トランザクションT1がある検索条件に合致するN 行の行を読み出す。次にトランザクションT2 がその検索条件に合致するような1 行以上の行を`INSERT` する。すると、T1 が先ほどと同じ条件でもう一度読み出すと、先ほどは存在しなかった行が結果に含まれる。
- P4（失われた更新）：失われた更新が起きるのは、例外的な状況だ。トランザクションT1 がデータを読み出す。次にT2 がそのデ

ータを更新する。そして最後に、T1が（更新前の値に基づいて）、データを更新し、コミットする。これによって、T2の行った更新は消えてしまう。

これらの現象は常に悪いわけではない。もしデータベースで実行されるSQL が検索のみで、データに対する更新が一切発生しないのならば、そもそもこうした問題は発生しない。また、こうした問題から身を守る必要がない場合、データベースは非常に高速に処理を実行する。特定の条件下で更新が行われる場合には、これらの現象を許容できる場合もあるだろう。

たとえば、世界中の車の情報を持つテーブルがあると仮定しよう。このテーブルから、赤いスポーツカーに乗っているドライバーの平均年齢を求めるクエリを実行する。このクエリの実行には少し時間がかかるだろう。そしてその間にも、何台かの車は事故を起こし、売られたり買われたり、新車が組み立てられたりする。しかし、クエリを実行した時点から結果が返ってくるまでの間に、平均年齢が大きく変わることはないだろうから、この5つの現象を受け入れてもよいと思うだろう。小数点第2位以下が変わる程度の変化が問題になることは少ない。

だが、夫が共同名義の口座への預け入れを行い、妻が引き出しを行うデータベースにおいては話は別だ。こういう場合は、いずれの現象も受け入れたくないだろう。この問題を解決するために導入する仕組みが、トランザクション分離レベルである。

2.3.2　分離レベル

標準SQLにおいて、ユーザーは、セッションで分離レベルを設定する。設定する分離レベルに応じて、上記の現象のいくつかを避け、またトランザクションに関する情報をデータベースに与えることができる。分離レベル設定の構文は以下のとおりである。

```
<set transaction statement> ::= SET TRANSACTION <transaction mode list>
<transaction mode list> ::=
<isolation level>
| <transaction access mode>
| <diagnostics size>
<diagnostics size> ::= DIAGNOSTICS SIZE <number of conditions>
<transaction access mode> ::= READ ONLY | READ WRITE
<isolation level> ::= ISOLATION LEVEL <level of isolation>
<level of isolation> ::=
READ UNCOMMITTED
| READ COMMITTED
| REPEATABLE READ
| SERIALIZABLE
```

オプションの<diagnostics size>句は、指定されたサイズでのエラーメッセージのリストを設定するよう、データベースに指示する。これは標準SQLの機能であるため、特定の製品では使えないかもしれない【訳注3】。その理由は、単一の文が複数のエラーを返すことがあり、データベースエンジンは、それらすべてを見つけて、GET DIAGNOSTICS文経由でホスト言語に通知することになっているからである。

<transaction access mode>はその名のとおりである。READ ONLYオプションが付いていると、クエリだけしか発行しないため処理が読み取り専用となり、SQLエンジンはいくらか気をゆるめることができるとわかる。READ WRITEオプションは、SQLエンジンに、行

が変更される可能性があることを知らせる。そのため、SQL エンジンは5 つの現象が発生しないか監視する必要がある。

　重要なのは<isolation level>句である。これは、最近のデータベース製品のほとんどで実装されている。トランザクション分離レベルは、トランザクションが同時実行されている別のトランザクションから、どの程度の干渉を受けるかの度合いを定義する。分離レベルのデフォルトはSERIALIZABLEである。しかし、ユーザーは<set transaction statement>で明示的にレベルを設定できる。いずれの分離レベルも、トランザクションの原子性を担保し、いかなる更新も失われないことを保証する。データベースは、2つ以上のトランザクションの直列性が保証できない場合、または復元不可能なエラーを検知したときは、自動的にロールバックを行う。

　分離レベルと代表的な3 つの現象の対応を見てみよう（表2.1）。○印は、当該の分離レベルにおいてその現象が発生しうることを意味する。SERIALIZABLEレベルでは、複数のトランザクションが並列実行された場合でも、それぞれが直列に実行された場合と同じ結果が生成されることが保証される。直列実行とは、1つのトランザクションが終わるまでは、次のトランザクションが開始されないような実行のしかたである。まるでユーザーが、データベースを利用するために一列に並んでいるようなものだ。

表2.1　分離レベルと3 つの現象

トランザクション分離レベル	P1（ダーティリ	P2（反復不能読	P3（ファント

	ード）	み取り）	ム）
SERIALIZABLE			
REPEATABLE READ			○
READ COMMITTED		○	○
READ UNCOMMITTED	○	○	○

REPEATABLE READレベルでは、同じセッションの間は、同じ読み取り結果が保証される。

READ COMMITTEDレベルでは同じセッションの間でも、他のトランザクションが変更し、かつコミットしたデータが読み出される。

READ UNCOMMITTEDレベルでは、同じセッションにおいて、他のトランザクションがコミットしなくても、変更したデータが見える。

なお、分離レベルによらず、SQL 文の実行以外、たとえば整合性制約をチェックするときのスキーマ定義の暗黙の読み込みや、参照整合性制約によって実施される参照動作の実行においては、P1、P2、P3の現象は起きない。

CURSOR STABILITY 分離レベル

CURSOR STABILITYレベルは、READ COMMITTEDのSQLカーソルに対するロック動作を、カーソルからのフェッチに新しい読み込み動作を追加することで拡張したものだ。このレベルでは、カーソル

のカレント項目に対してロックがかけられ、（普通はコミットによって）カーソルが動くかクローズされるまでロックが継続する。当然のことだが、フェッチするトランザクションは、行を更新できるし、その場合には、たとえカーソルが後続項目のフェッチのために動いたとしても、COMMITされるまで対象行に書き込みロックが取得される。これによって、CURSOR STABIL ITYはREAD COMMITTEDよりも強く、REPEATABLE READよりも弱いレベルとなっている。

CURSOR STABILITY は、カーソル経由で読まれた行の失われた更新を防ぐために、多くのDBMSで実装されている。中には、READ COMMITTEDのほうがCURSOR STABILITYより強いシステムもあるが、ANSI標準ではこれを許可している。

実のところ、標準SQLは、こうした結果を保証するための実装方法については何も規定していない。だが同時実行制御の方法は、楽観的なものと悲観的なものに大別できる。この2 つの種類を、ベンダーは独自の実装で実現している。

2.4

悲観的な同時実行制御

悲観的な同時実行制御は、トランザクション同士が互いに衝突するという前提に基づいて考えられている。したがって、「問題は起こる前に回避する」という設計方針になっている。

悲観的な同時実行制御は、必ずロックを使う。ロックとはデータベースに設定されるフラグであり、このフラグが、スキーマオブジェクトに対するユーザーの排他的アクセスを許可する。飛行機内のトイレのドアにぶら下がっている「使用中」という札を想像するといい。

一方、これとは異なるロックメカニズムもある。たとえば、DB2 for z/OSは「ラッチ」という仕組みを備えている【訳注4】。これは伝統的なロックとは少し異なる。重要な違いは、ロックのレベルである。いちいちフラグをオン／オフにするのは時間とリソースを消費する。もしデータベース全体をロックすれば、そのデータベースは直列で動くバッチ処理システムと化す。一度に1つのトランザクションしか動けないからだ。だが実際は、このようにデータベースを使うのは、データベース全体をメンテナンスするときだけだろう。もしテーブルレベルでロックすれば、多くのユーザーがアクセスするテーブルに対する待機が発生して、パフォーマンスが犠牲になる。しかし、テーブル全体を処理対象にするトランザクションがあるのなら、それらのトランザクションはただ1つのフラグしか使わない。

もしテーブルを行レベルでロックすれば、他のユーザーもテーブルの他の行へアクセスできる。これが最も望ましい共有アクセスの方法だ。ただし、この場合、非常に多くのフラグを扱うことにな

り、やはりパフォーマンスが問題になる。このアプローチは、一般には現実的でない【訳注5】。

テーブルロックと行ロックの間に位置するのがページロックだ【訳注6】。このアプローチでは、テーブル内の（ロックしたい行を含む）行のサブセットに対してロックをかける。ページロックという名前の由来は、物理ストレージにおいて、テーブルがページという単位で実装されていることが多いからだ。パフォーマンスは、物理ストレージにおける統計的な分散に依存するが、一般的にはそれほど悪くないレベルになる。

2.5

スナップショット分離と楽観的同時実行制御

楽観的同時実行制御は、トランザクション同士がそんな頻繁には衝突しないという前提に基づいている。したがって、「問題が起きてから対処しよう」という設計方針になっている。このときよく使われる手段が、スナップショット分離である。

スナップショット分離では、個々のトランザクションは、データを、トランザクションが開始された時点の（コミットされた）データのスナップショットから読み出す。この開始タイムスタンプを「t0」と呼ぼう。これは、トランザクションが最初に読み込みを行うよりも前の時刻である。スナップショット分離では、実行中のトランザクションは、読み込みにおいてブロックされることはない。なぜなら、トランザクションが読み込むデータは、そのトランザクションのためだけのコピーデータだからだ。これが意味するのは、個々のデータは常に、アクティブなトランザクションあるいはコミットを行ったトランザクションによって作られた複数のバージョンを持つ可能性がある、ということである。

トランザクションT1がコミットする準備ができると、T1はコミットタイムスタンプを取得する。これはすべての既存の開始タイムスタンプとコミットタイムスタンプよりも後の時刻である。トランザクションT1がコミットを成功させられるのは、T1の実行期間（つまり開始タイムスタンプからコミットタイムスタンプの間）に、T1が書き込もうとしたのと同じデータに書き込むためのコミットタイムスタンプを取得したトランザクションT2がいない場合だけである。もしそういう「先客」のT2 がいた場合、T1 はロールバックする。こ

の「コミット早い者勝ち」戦略によって、失われた更新（P4）が防止できる。そしてT1がコミットすると、その変更が、T1のコミットタイムスタンプより後の開始タイムスタンプを持つすべてのトランザクションから見えるようになる。

　スナップショット分離は、直列性を満たしていない。なぜなら、トランザクションの読み込みはある時刻に起こり、その書き込みは別の時刻に起こるからだ。たとえば、同じデータに対して処理を行う複数のトランザクションがあるとする。そして（x+y）が正の数になるという制約があるとする。このとき、xとyに新しい値を書こうとする個々のトランザクションは、制約を満たさなければならない。だが、T1とT2は、単独で動くときは制約を満たすことができても、同時に更新しようとすると制約を破ることがある。起こりうる問題を整理すると、以下のようになる【訳注7】。Microsoft Researchが公開しているドキュメント（原注1を参照）から引用する。

> - A5（データ項目の制約違反）：制約Cが2つのデータ項目xとyの間に成立するとする。このとき、制約違反によって以下の2つの更新異常が発生する。
> - A5A（読み込み異常）：トランザクションT1がxを読み、次にトランザクションT2 がxとyを新しい値に更新し、コミットする。その後でT1が再度yを読み込むと、T1 からはx とy の間の制約が崩れているように見えることがある。したがって、出力として一貫性のない状態を生じさせる。

- A5B（書き込み異常）：T1 が、制約C によって一貫性のあるx とy を読み込むとする。その次に、T2がxとyを読み込み、xに新しい値を書き込み、コミットする。その後、T1がyを更新すると、xとyの間にあった制約が破られる可能性がある。

反復不能読み取り（P2）はx=yの場合に起きる読み込み異常の変化版だ。典型的には、あるトランザクションが2つの異なる、しかし（参照整合性制約などによって）関連付けられた項目を読み込む場合に起きる。

書き込み異常（A5B）が起こりうるケースとしては、銀行の口座管理システムの制約を考えてみるといい。同一名義によって開かれている口座の残高合計がマイナスにならない限り、特定の口座残高はマイナスの値になってもかまわない。

P2 が防止されている限り、A5A もA5B も起きないことは明らかである。なぜなら、A5AとA5Bはともに、コミットを行っていないT1によって読み出された値を、T2が更新したときに起きるからである。したがって、A5AとA5Bで注意すべきなのは、`REPEA TABLE READ`より低い分離レベルを区別したい場合だけである。

楽観的同時実行制御は、データベースが成立する何十年も前から存在していた。企業がマイクロフィルムにデータを保存し始めた頃、記録を一元管理する部署があって、この仕組みを人力で運用し

ていたのだ。この仕組みでは、ユーザーはマイクロフィルムの原本を手に入れることはできない。代わりに、タイムスタンプが押されたコピーを渡される。そのコピーを自席のデスクの上に置いて、赤入れをして、管理部に返す。すると管理部の職員が、ドキュメントの更新された時間を記録し、写真を撮って、マイクロフィルムの履歴の一番最後に追加する。

だがこのとき、もし2人目のユーザー（ユーザー2）が管理部に来て、同じドキュメントをタイムスタンプ付きでコピーした場合はどうなるだろうか。管理部の職員は、2つのタイムスタンプを見比べて、決断を下さなければならない。もしユーザー1が、ユーザー2がまだコピー2で作業中に、コピー1での作業結果をデータベースに書き込もうとした場合、職員は、コピー1を預かるか、ユーザー1にいったん返却するか、いずれにせよユーザー2がコピー2を返却するまで待たなければならない。

両方のコピーが戻ってきたときには、職員は、その2つをライトの下で重ねて見て、矛盾がないかチェックする必要がある。そしてどちらの更新もデータベースに反映可能ということになれば、職員は2つの更新を実施する。もし矛盾があれば、ルールに従って矛盾を解消するか、または両方のトランザクションを拒否しなければならない。これは事後的対応ではあるが、一種の行レベルロックである。

2.6

論理的な同時実行制御

論理的な同時実行制御は、作業待ちの一群のSQL 文に対して、条件を機械的に分析することができるとき、純粋に論理的なレベルでそれを処理し、どのSQL文同士ならば同時に実行することが許されるかを決める、という方法だ。

たとえばSELECT 文ならば、絶対にデータを更新しないので、どんな組み合わせであっても、同時実行は許される。だがそれ以外のケースでは、どの文がどの文と衝突するかを判断するのは、複雑な作業だ。たとえば、互いに異なるテーブルを更新する2 つのUPDATE文を考えよう。問題は、主キーや外部キーの制約によって、この2つのUPDATE文は、特定の順番でしか実行できない可能性があることだ。また、同じテーブルを更新する2 つのUPDATE文の場合も、やはり同じ行を更新し、異なる結果になる場合は同時実行は許されないことがある。他方、同じテーブルを更新するUPDATE文同士であっても、同時実行が許される場合もある。たとえば、異なる行を更新するため、何ら衝突を起こさないケースだ。

このアプローチにおける問題の1つは、キューに長時間並んでいる文をどうするか、というものだ。これは、次節で取り上げるライブロック（livelock）問題の1つである。一般的な解決策としては、処理待ちのトランザクションに優先度を示す番号を割り当て、一定時間待ったら、その番号を減らしていく（つまり、優先度を上げる）。すると最終的には、どのトランザクションも優先度1番の番号に到達し、優先的に実行されるときがくる。この方法は、優先的に実行したいトランザクションを列に割り込ませる場合にも使える。もっと

もこれによって、どうでもいいジョブを重要なジョブよりも前に割り込ませてしまうこともある。ソリティアのプレイと給与明細の印刷を戦わせることになりかねない。

2.7

デッドロックとライブロック

ユーザーがトランザクションを完了させられない理由はハードウェア障害だけではない。たとえばデッドロックがある。デッドロックとは、2 人以上のユーザーが互いに必要とするリソースを確保してしまい、相手の必要なリソースを解放できなくなる状態である。具体的な例として、ユーザーAとBが、ともにテーブルXとYを使いたいケースを考えよう。ユーザーA がテーブルXをロックし、Bがテーブルをロックする。次に、ユーザーAはテーブルYをロックしたいと思い、ユーザーBはテーブルXをロックしたいと思うが、相手が確保しているのでそれができず、利用可能になるまで待機を続ける――永遠に利用可能になることはないのだが。デッドロックに対する一般的な解決策は、データベース管理者（DBA）に片方のユーザーのセッションを終了させてもらい、ロールバックさせることだ。

他方、ライブロックという現象もある。これは、あるユーザーがあるリソースを使いたくて待っているのだが、他の（複数の）ユーザーがずっとリソースを確保していて、永遠に順番が回ってこないことである。デッドロックのように、1 人のユーザーがずっとリソースを占有しているということはないが、入れかわり立ちかわり占有していくので、結局自分には回ってこない。

具体的な事例を考えよう。ユーザーAがテーブルXの全行を必要としているとする。しかし、テーブルXは常に数百というユーザーから更新を受けている。それゆえ、ユーザーAはロックのかかっていないページを1つとして見つけることができない。仕方ないので、ユーザ

ーAはじっと座ってすべてのページが利用可能になるまで待つしかない。しかし、業務時間中にテーブルXが空く気配はない。

　これを防ぐため、DBAが関連するセッションをいくつか終了させ、ロールバックさせることはできる。DBAがライブロックされているセッションの優先度を上げることができるシステムもある。

　こうした問題は、決して些細な問題ではない。どんなデータベースも、独自のトランザクション処理と同時実行制御の仕組みを持っている。こうしたことは、DBAが責任を持つべき領域であり、アプリケーションプログラマにあまり意識させるべきではない。だが、水面下で何が起こっているかを知るのは、また楽しいことでもあるのだ。

【訳注1】　分離性、隔離性とも訳される。

【原注1】　Microsoft Research Technical Report: MSR-TR-95-51, Hal Berenson, Phil Bernstein, Jim Gray, Jim Melt on, Elizabeth O'Neil, and Patrick O'Neil "A Critique of ANSI SQL Isolation Levels" (1995) http://research.microsoft.com/apps/pubs/default.aspx?id=69541

【訳注2】　この説明はわかりにくいが、原注1のドキュメントでも挙げられている以下の例を考えるとわかりやすい。(x, y) = (0, 0)を初期値として、以下のようなトランザクションT1とT2の動作を考える。

① T1がx=1に更新する。 (x, y) = (1, 0)

② T2がx=2に更新する。 (x, y) = (2, 0)

③ T2がy=2に更新する。 (x, y) = (2, 2)

④ T2がコミットする。 (x, y) = (2, 2)

⑤ T1がy=1に更新する。 (x, y) = (2, 1)

⑥ T1がコミットする。 (x, y) = (2, 1)

この結果、最終的に (x, y) = (2, 1)となり、x=yという仮定した制約に違反する。ポイントは、⑤において、T1は、自分としては、x=yという制約を満たすような変更を行ったつもりだ、ということにある。というのも、T1は①でx を1に更新しているため、同じトランザクション内でy を1に更新すれば、T1としては⑥コミットの時点で(x, y)=(1, 1)の状態が作れたはずである――T2が邪魔をしなければ。

【訳注3】 標準SQLと個々の実装の関係について、ここで少し読者の注意を促しておきたい。「標準SQLはすべての実装で使えるはずではないのか？」と思う読者もいるかもしれないが、SQLにおいては、必ずしもそうではない。また、実装が標準にない独自構文や機能を持っていることも多い。

ただ、現在は基本的には、多くの実装は標準に準拠することを意識しており、標準SQLの大部分はそのまま実装で利用できる。そうでない例外については、適宜本文および訳注で注意を喚起している。

【訳注4】 ラッチは一般に、DBMS 内部のリソースに対して取得される非常に短く、細かいロックを指して使われる。ロ

ックと違い、ユーザーが明示的にラッチを取得することは、普通できない。

【訳注5】　著者はこう言うが、Oracleをはじめ行ロックを基礎としているDBMSも少なくない。

【訳注6】　ページロックを採用している有名なDBMSとしては、Microsoft社のSQL Serverがある。ページロックに関しては、本文でも言及されているように、ロック対象としたい行以外の行までロックしてしまうという問題のほか、粒度の細かいロックが増えてコストが高いと判断すると、DBMSがロック粒度を自動的に上げる（ロックエスカレーション）といった動作も行う。たとえばSQL Serverでは、行ロック→ページロック→テーブルロックというふうにロック粒度が上がっていく。こうした機能は、同時実行の競合を増やしパフォーマンスにトレードオフをもたらすほか、デッドロック等の問題を発生させる要因となることもあり、その功罪については議論がある。詳細については、Microsoft社の以下の技術文書などを参照。

- MSDN ロックの粒度と階層 SQL Server 2008 R2
  http://msdn.microsoft.com/ja-jp/library/ms189849
- MSDN ロックのエスカレーション（データベースエンジン）SQL Server 2008 R2
  http://msdn.microsoft.com/ja-jp/library/ms184286

【訳注7】　いきなり「A5」という記号が出てきて驚いたかもしれないが、AはAnomaly（異常）の略である。A1からA4は原注1のドキュメントで言及されている。いきなり中途半端なところから引用が始まったので奇妙に感じるだけである。

3

スキーマレベルのオブジェクト

Schema Level Objects

　データベースはただのテーブルの束ではない――そういう使われ方をすることも多いが。データベースにはテーブル以外にも、ユーザーが作ったストアドプロシージャ、ユーザー定義関数、カーソルなどが含まれる。さらに直接アクセスすることはできないが、インデックスその他のアクセス手段も含まれる。

　本章では、ユーザーが作ることのできるスキーマオブジェクトについて見ていく。なお、標準SQLではユーザーをUSERロールとADMINロールに分けている。上に挙げたオブジェクトは、作ったり、変更したり、削除するためにはADMIN 権限を必要とする。USER権限を持つユーザーは、呼び出して結果にアクセスすることしかできない。

3.1

CREATE SCHEMA 文

CREATE SCHEMA文は標準SQLでは、スキーマ全体を一挙に作成することになっている。実際の製品では、ユーティリティプログラムが物理的ストレージの割り当てとスキーマ定義を行う。構文にも違いが多く、ベンダー独自構文の大半は、物理的なストレージの割り当てに関するものだ。

スキーマは名前とキャラクタセットを持たなければならない。数年前、デフォルトのキャラクタセットはASCIIかローカルのアルファベット（8ビット）だった。いまではUnicode（16ビット）のほうをよく見かける。スキーマ定義にはオプションのAUTHENTIFICATION句がある。これはセキュリティのためにスキーマ認証識別子を定義する。スキーマ認証識別子のあとには、以下のようなスキーマ要素がリストされる。

```
<schema element> ::=
<domain definition> | <table definition> | <view definition>
| <grant statement> | <assertion definition>
| <character set definition>
| <collation definition> | <translation definition>
```

スキーマはSQLデータベースのスケルトン（骨格）となり、スキーマオブジェクトと、それを操作する際に従うべきルールを定義する。データはそのスケルトンの周りに付く肉である。

SQLにおける唯一のデータ構造はテーブルである。テーブルは永続的であるが（基底テーブル）、作業用のストレージ（一時テーブル）や仮想テーブル（ビュー、共通表式、導出テーブル）にも利用できる。これらさまざまなタイプの区別は、実装方法によるもので

あって、パフォーマンスの違いに基づくものではない（たしかにパフォーマンスも違うのだが）。SQLが1つしかデータ構造を持たないことにはいくつか利点があるが、その1つとして、テーブルに対するあらゆる操作の結果がテーブルになることが挙げられる【訳注1】。それゆえ、構造を変換したり、特別な演算子を書いたりといった、他の言語でデータを扱うときにつきもののイレギュラーな特性に悩まされずに済むのだ。

なお、<grant statement>は、ユーザーがアクセスするスキーマオブジェクトを制限するために使われる文だ。<assertion definition>は、実装されている製品はまだ少ないが、要するにスキーマ全体に対する制約と考えてよい。そして最後の<character set definition>、<collation definition>、<translation definition>はデータの表示に関わる。これらは、スキーマオブジェクトには何の関係もない。設定するのはDBAであり、プログラマは変更しない。

テーブルとビューはSQLにおける基本的な作業単位であるので、別章（第5 章）で取り上げる。

プロシージャを構成する文は、SQL/PSM や他の言語で書かれた手続き型コードのモジュールをデータベースに作成する。これらは必要に応じて呼び出される。これらのオブジェクトについても、別章（第5章）で論じる。

3.2

CREATE DOMAIN 文

ドメイン（定義域）は、標準SQLにおけるスキーマ要素だ。これを使って、ユーザーは、頻繁に使う列定義に対して、スキーマ内の1か所で、インラインマクロを宣言することができる。構文は以下のとおりである。

```
<domain definition> ::=
CREATE DOMAIN <domain name> [AS] <data type>
[<default clause>]
[<domain constraint> ...]
[<collate clause>]

<domain constraint> ::=
[<constraint name definition>]
<check constraint definition> [<constraint attributes>]
```

```
<alter domain statement> ::=
ALTER DOMAIN <domain name> <alter domain action>

<alter domain action> ::=
<set domain default clause>
| <drop domain default clause>
| <add domain constraint definition>
| <drop domain constraint definition>
```

ドメインについて注意すべき点は、定義するときは、他のドメインではなく、基本的なデータ型を使って定義しなければならないということだ。要するに、ドメインの上にドメインを定義することはできない。いったん宣言されると、ドメインは列に対するデータ型の宣言の代わりに使うことができるようになる。

CHECK句には、チェックディジット、範囲、リスト、その他の条件をチェックするためのコードを書くことができる。たとえば、米国の州コードをドメインとするスケルトンはこうなる。

```
CREATE DOMAIN StateCode AS CHAR(2)
DEFAULT '??'
CONSTRAINT valid_state_code
CHECK (VALUE IN ('AL', 'AK', 'AZ', ...));
```

ドメインはスキーマ内で1か所にしか存在しないため、このドメインから列を定義する場合はいつでも、正しいデータの範囲が設定されているのだと安心できる。もしDOMAIN句がなければ、同じCHECK句をあちこちのテーブル定義にコピーしなければならない。なお、ALTER DOMAINとDROP DOMAIN文は、簡単なので特に説明の必要はないだろう。

3.3

CREATE SEQUENCE 文

シーケンスは、呼び出されるたびに連続した値を生み出すジェネレータである。呼び出すときは関数のように使うことができて、数列における次の値を得られる。

私は、昔書いた本の中で1～nの整数値を保持する「Sequence」という名前のテーブルを使ったことがある。だがこの「Sequence」という語は、いまではSQL の予約語になっているので、オブジェクト名に使うのはよくない。そのため、本書ではこういう機能を持つテーブル名を「Series」に変えている。

シーケンスを作る構文は次のとおりである。

```
<sequence definition> ::= CREATE SEQUENCE <seq name> AS <data type>
START WITH <value>
INCREMENT BY <value>
[MAXVALUE <value>]
[MINVALUE <value>]
[[NO] CYCLE]
```

シーケンスから値を得るには、妥当なデータ型であれば常に次の式を使う。

```
NEXT VALUE FOR <seq name>
```

シーケンスをリセットしたい場合は、数列を再開するためのオプションを付ける。

```
ALTER SEQUENCE <seq name>
RESTART WITH <value>; -- この値から再開する
```

シーケンスを削除する文は、もちろんこうだ。

```
DROP SEQUENCE <seq name>;
```

　このシーケンス機能は、現在ではOracle、DB2、PostgreSQL など多くの実装で利用可能になっているが、それでもなお使うべきではない【訳注2】。これは非リレーショナルな拡張であり、集合指向的というより、まるでシーケンシャルファイルに対する操作か、手続き型言語のように振る舞うからだ。

3.4

CREATE ASSERTION 文

標準SQLでは、CREATE ASSERTION文によって、同じスキーマ内の複数のテーブルに対して制約を適用する。これは「表明」と呼ばれ、特定の1 つのテーブルに対する制約とは異なる【訳注3】。構文は以下のとおりである。

```
<assertion definition> ::=
CREATE ASSERTION <constraint name> <assertion check>
[<constraint attributes>]
<assertion check> ::=
CHECK (<search condition>)
```

ご想像のとおり、DROP ASSERTION文もあるが、ALTER ASSERTION文はない。表明を使えば、単一のテーブルに付与されたCHECK制約にはできないことが可能になる。なぜなら、これはテーブルの外側に存在するからだ。

たとえば、会社の従業員数が複数の健康計画テーブルの従業員数の合計と一致する、というルールをCHECK 制約で作ることは非常に困難だ。CHECK 制約は、テーブルが空の場合は常にTRUEになってしまうからだ。だが、表明を使えばこれは簡単な仕事だ。

```
CREATE ASSERTION Total_Health_Coverage
CHECK (SELECT COUNT(*) FROM Personnel) =
+ (SELECT COUNT(*) FROM HealthPlan_1)
+ (SELECT COUNT(*) FROM HealthPlan_2)
+ (SELECT COUNT(*) FROM HealthPlan_3);
```

CREATE ASSERTION文はスキーマ全体に作用するので、制約名もスキーマに対してグローバルになる。CHECK制約のように、それが宣言されたテーブル内に閉じるわけではない。

3.4.1　スキーマレベルの制約をビューで実現する

CREATE ASSERTION文が使えない環境では、同じことを実現するにはプロシージャやトリガーを使わなければならない。3つのテーブルから成るチェーンストアのスキーマを考えてみよう。

```
CREATE TABLE Stores
(store_nbr INTEGER NOT NULL PRIMARY KEY,
 store_name CHAR(35) NOT NULL);
```

Stores：店舗　　store_nbr：店舗番号　　store_name：店舗名

```
CREATE TABLE Personnel
(emp_id CHAR(9) NOT NULL PRIMARY KEY,
 last_name CHAR(15) NOT NULL,
 first_name CHAR(15) NOT NULL);
```

Personnel：社員　　emp_id：社員ID　　last_name：姓　　first_name：名

この2つのテーブルの意味は自明だ。3つ目のテーブルは、店舗と従業員の関連、つまり、誰がいつ、どの店舗で何の仕事をしていたかを表すテーブルだ。

```
CREATE TABLE JobAssignments
(store_nbr INTEGER NOT NULL
   REFERENCES Stores (store_nbr)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 emp_id CHAR(9) NOT NULL
   REFERENCES Personnel(emp_id)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
 end_date TIMESTAMP,
   CHECK (start_date <= end_date),
 job_type INTEGER DEFAULT 0 NOT NULL -- 未割り当ては「0」
   CHECK (job_type BETWEEN 0 AND 99),
   PRIMARY KEY (store_nbr, emp_id, start_date));
```

JobAssignments：仕事割り当て　　store_nbr：店舗番号　　emp_id：社員ID
start_date：開始日　　end_date：終了日　　job_type：職種

このテーブルにおいて、「job_type」列のコードは、

0＝未割り当て
1＝棚卸し
⋮
99＝店長

というように定義しよう。そして、各店舗はいずれも店長は0人または1人とする。標準SQLでは、このビジネスルールを次のような制約で実装できる。

```
CREATE ASSERTION ManagerVerification
CHECK (1 >= ALL (SELECT COUNT(*)
                   FROM JobAssignments
                  WHERE job_type = 99
                  GROUP BY store_nbr));
```

この制約は見た目よりも少し微妙な動きをする。もし>=を=に変えたら、1人でも店員がいる限り、店舗は常に1人の店長を持たなければならない。

だが繰り返しになるが、大半のDBMSは、まだテーブル全体に適用できるCHECK 制約も、スキーマレベルに適用できるCREATE ASSERTION文も実装していない。

それでは、どのようにしてこのルールを実装すればよいだろう？トリガーを使うという方法もあるかもしれない。だがそうすると、ベンダー依存の手続き型のコードを書くことになる。標準の

SQL/PSMの代わりに、多くのベンダーは互いに似ても似つかないトリガーのモデルを実装し、独自の4GL言語をトリガー本体で使っている。

　また、テーブルの整合性を保つためにトリガーを使おうとすると、INSERTやUPDATEをするたびにトリガーを実行することになるし、従業員を削除する場合には、トリガーが自動的に店舗につき最大でも店長が1人になるよう調整してくれるわけではない。

　それでもあえてトリガーでやるなら、このルールを実装するトリガーのスケルトンは、こんな感じになるだろう【訳注4】。

```
CREATE TRIGGER CheckManagers
AFTER UPDATE ON JobAssignments -- INSERTの場合も同じ
IF 1 >= ALL (SELECT COUNT(*)
                FROM JobAssignments
               WHERE job_type = 99
               GROUP BY store_nbr)
THEN COMMIT;
ELSE ROLLBACK;
END IF;
```

　人は「こだわりすぎ」と言うかもしれないが、それでもやはり、私としてはピュアSQLによる、宣言的な解決方法にこだわりたい。しかも、現在のデータベース製品の制限の範囲内で可能な方法でだ。

　そこで2つのテーブルを作ろう。1つ目のテーブルは、店長専用の社員テーブルである。店長は従業員ID（emp_id）で管理されている。「job_type」列のDEFAULT 句とCHECK 制約に注目してほしい。これは、事実上このテーブルが「店長専用」であることを示している。

```
CREATE TABLE Job_99_Assignments
(store_nbr INTEGER NOT NULL PRIMARY KEY
  REFERENCES Stores (store_nbr)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
 emp_id CHAR(9) NOT NULL
  REFERENCES Personnel (emp_id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
 start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
 end_date TIMESTAMP,
  CHECK (start_date <= end_date),
 job_type INTEGER DEFAULT 99 NOT NULL
  CHECK (job_type = 99));
```

Job_99_Assignments：店長割り当て　store_nbr：店舗番号　emp_id：社員ID
start_date：開始日　end_date：終了日　job_type：職種

2 つ目のテーブルは、店長以外の従業員のための社員テーブルである。従業員はやはりemp_id列で管理されている。job_type列のDEFAULT 句とCHECK 制約は、今度は従業員が「店長ではない」ことを保証している。

```
CREATE TABLE Job_not99_Assignments
(store_nbr INTEGER NOT NULL
   REFERENCES Stores (store_nbr)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 emp_id CHAR(9) NOT NULL PRIMARY KEY
   REFERENCES Personnel (emp_id)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
 end_date TIMESTAMP,
   CHECK (start_date <= end_date),
 job_type INTEGER DEFAULT 0 NOT NULL
   CHECK (job_type BETWEEN 0 AND 98));  -- 99ではない
```

Job_not99_Assignments：店員割り当て　store_nbr：店舗番号　emp_id：社員ID
start_date：開始日　end_date：終了日　job_type：職種

次に、この2 つのテーブルから、すべてのレコードをUNIONでくっつけたビューを作る。いままでJobAssignmentsテーブルを検索していたが、今度からはこちらを検索するようにする。

```
CREATE VIEW JobAssignmentsView (store_nbr, emp_id, start_date, end_date, job_type)
AS
(SELECT store_nbr, emp_id, start_date, end_date, job_type
   FROM Job_not99_Assignments
 UNION ALL
 SELECT store_nbr, emp_id, start_date, end_date, job_type
   FROM Job_99_Assignments);
```

ビューの中の2 つのテーブルに付与されているキーとjob_type列の制約が同時に作用するため、1 つの店に最大でも1 人しか店長がいないことが保証される。次のステップは、INSTEAD OFトリガーをビューに付与するか【訳注5】、ストアドプロシージャを書くことである。ユーザーはそれを使うことで、簡単に挿入、更新、削除を行うことができる。エラーハンドラや入力チェックを省略した簡略なストアドプロシージャを以下に示す。まずは店員の登録用からだ。

```
CREATE OR REPLACE PROCEDURE InsertJobAssignments
(store_nbr IN INTEGER, new_emp_id IN CHAR, new_start_date IN DATE, new_end_date IN ➥
DATE, new_job_type IN INTEGER)
IS
BEGIN
    IF new_job_type <> 99 THEN
        --店長以外の社員を追加
        INSERT INTO Job_not99_Assignments
        VALUES (store_nbr, new_emp_id, new_start_date, new_end_date, new_job_type);
    ELSE
        --店長となる社員を追加
        INSERT INTO Job_99_Assignments
        VALUES (store_nbr, new_emp_id, new_start_date, new_end_date, new_job_type);
    END IF;
END;
```

同様に、店員の店舗割り当てを解除するときはこうだ。

```
CREATE PROCEDURE FireEmployee (new_emp_id CHAR, new_job_type IN INTEGER)
IS
BEGIN
    IF new_job_type <> 99
    THEN
         DELETE FROM Job_not99_Assignments WHERE emp_id = new_emp_id;
    ELSE
         DELETE FROM Job_99_Assignments WHERE emp_id = new_emp_id;
    END IF;
END;
```

開発者がINSERT文、UPDATE文、DELETE文などによって直接Job_Assignmentsビューを更新しようとすると、ビューは更新可能ではないというエラーメッセージを受け取ることになる。というのは、このビューはUNIONを含んでいるからだ【訳注6】。これは、もうけものだ。そのおかげで、開発者は社員の店舗への割り当てを変更するには、ストアドプロシージャを使わざるを得ないからだ。

もう一度繰り返すが、これは特定の制限の課せられた状態での、プログラミングによる解決を図る練習だ。決して理想形ではないし、ビューよりもトリガーのほうがパフォーマンスは良いかもしれない。

3.4.2 制約のために主キーと表明を使う

前節で見た「店舗と社員」問題の別バージョンを考えてみよう。

```
CREATE TABLE JobAssignments
(emp_id CHAR(9) NOT NULL PRIMARY KEY -- 店舗をかけもちする従業員はいない
   REFERENCES Personnel (emp_id)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 store_nbr INTEGER NOT NULL
   REFERENCES Stores (store_nbr)
     ON UPDATE CASCADE
     ON DELETE CASCADE);
```

社員ID（emp_id）に対する主キー制約は、複数の店舗をかけもちする従業員がおらず、かつ、1 つの店舗では複数の従業員が働けることを保証している。理想的には、各従業員が本当に店舗に割り当てられているかをチェックする制約もほしいところだ。これを実現するための制約として、最初に思いつくコードは次のようなものである。

```
CREATE ASSERTION Nobody_Unassigned
CHECK (NOT EXISTS
             (SELECT *
                FROM Personnel AS P
                       LEFT OUTER JOIN
                            JobAssignments AS J
                         ON P.emp_id = J.emp_id
               WHERE J.emp_id IS NULL
                 AND P.emp_id IN (SELECT emp_id FROM JobAssignments
                                   UNION
                                  SELECT emp_id FROM Personnel)));
```

だがこれはちょっと大げさだ。emp_id列は主キーなので、Personnelテーブルと JobAssignmentsテーブルには、インデックスが存在するだろう。このことから、COUNT関数を使う次のような表明が、パフォーマンス的には有利なはずだ【訳注7】。

```
CREATE ASSERTION Everyone_assigned_one_store
CHECK ((SELECT COUNT(emp_id) FROM JobAssignments)
     = (SELECT COUNT(emp_id) FROM Personnel));
```

これを初めて見ると、驚く人もいるかもしれない。それは、従業員と仕事の割り当てという1 対1 対応は結合で表すもの、と思い込んでいるからだ。しかし、主キーと外部キーの制約によって、両者が対応するならば、それは1 対1になることが保証されている。もし仕

事を割り当てられていない「手持ちぶさた」の従業員がいれば、Personnel テーブルはJobAssignments よりも行数が多くなる。この表明によって、両者の行数が等しいことを保証すれば、常に1 対1 対応が成立する。賢いオプティマイザであれば、こうした条件も述語に展開して使うだろう。これが、トリガーやアプリケーション側のロジックで実装する代わりに、宣言整合性制約（DRI）を使う理由である。

ストアドプロシージャを使って整合性を担保しようとすると、INSERTのときは、1つのトランザクションで両方のテーブルへ挿入しなければならないし、Personnelを更新や削除するときは、カスケードでJobAssignmentsテーブルもきれいにしなくてはならない。

さて、ここで業務要件をちょっと変更して、従業員は複数の店舗で働けると考えてみよう。すると、JobAssignments テーブルの主キーを(emp_id, store_nbr)という複合キーに変えることになる。

```
CREATE TABLE JobAssignments
(emp_id CHAR(9) NOT NULL
   REFERENCES Personnel (emp_id)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 store_nbr INTEGER NOT NULL
   REFERENCES Stores (store_nbr)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
   PRIMARY KEY (emp_id, store_nbr));
```

この場合の「手持ち無沙汰な社員がいないこと」という制約を表現するには、表明でCOUNT (DISTINCT ...)を使えばよい。

```
CREATE ASSERTION Everyone_assigned_at_least_once
CHECK ((SELECT COUNT(DISTINCT emp_id) FROM JobAssignments)
     = (SELECT COUNT(emp_id) FROM Personnel));
```

一意性制約と表明は連動して動作することに注意してほしい。したがって、どちらか一方、または両方を変更する場合は、この1対1対応のテスト条件も変わることになる。

3.5

キャラクタセットとそれに関連する諸概念

文字を扱うためのスキーマレベルの概念がいくつかある。さまざまな言語や特定の目的のための名前付きのキャラクタセット（文字集合）を作ることができるし【訳注8】、その集合に対して1つ以上の照合順序を定義したり、1つの集合を別の集合に翻訳したりすることもできる。

今日では、Unicode 標準規格が広く使われている。実用されている文字の大半がUnicodeの名前と事前に定義された照合順序を持っている。たとえば、ISO 8859-1で定義されているように、SQL 文は、Latin-1 で書くこととされている。このキャラクタセットは、HTML でも使われており、191 文字のラテンアルファベットから成る。このキャラクタセットは、アメリカ、西ヨーロッパ、オセアニア、アフリカなど非常に広い地域で利用されている。また、東アジア言語のローマ字標準でもある。

1991 年以降、Unicode コンソーシアムはISO やIEC と協力して、Unicode 標準とISO/IEC 10646を作成した。後者は、タンデム社のマシンで採用されたUCS（Universal Character Set）である。現在、UnicodeとISO/IEC 10646 は約10 万文字を、100 万以上のコードポイントから成るコード空間に割り当てている。さらに、いくつかの標準的なエンコーディングも定義されており、任意の利用可能なコードポイントを表すことができる。UnicodeとUCSの標準的なエンコーディングは、以下のとおり。

- 1～4個の8ビット符号単位を使う（UTF-8）

- 1～2個の16ビット符号単位を使う（UTF-16）
- 1個の32ビットの符号単位を使う（UTF-32およびUCS-4）

1個の16ビット符号単位を使う古いエンコーディング（UCS-2）もあり、この方式で利用できるのは、17 面の符号位置だけである（UTF-16はUCS-2の拡張版だ）。こうしたエンコーディング形式のうち、UTF-8のバイト列のみが固定された順序を持っている。他はいずれも、プラットフォーム依存のバイト列の順序に依存する【訳注9】。

3.5.1 CREATE CHARACTER SET 文

おそらく読者は、次のような構文を見ることはないだろう。ベンダーは地域に合わせた言語設定に基づいて、システムレベルのキャラクタセットをデフォルト設定しているからである。

```
<character set definition> ::=
CREATE CHARACTER SET <character set name> [AS]
<character set source> [<collate clause>]
<character set source> ::=
GET <character set specification>
```

<collate clause>は普通、デフォルト定義されているが、名前付き照合順序を使うこともできる。これを次に見よう。

3.5.2 CREATE COLLATION 文

```
<collation definition> ::=
CREATE COLLATION <collation name>
FOR <character set specification>
FROM <existing collation name> [<pad characteristic>]
<pad characteristic> ::= NO PAD | PAD SPACE
```

<pad characteristic>オプションは、文字列を比較する方法を指定するために使う。もしNO PADが指定されていた場合、短い値のほうに空白を埋めることはせず、短い値は長い値よりも小さいとみなされる。PAD SPACEが指定された場合は、短い値は長い値と同じ文字長になるようにスペースが右側に追加される。

SQLは通常、短い方の文字列の語末をスペースで埋めて、1文字ずつ比較を行う。

3.5.3 CREATE TRANSLATION 文

この文は、1つのキャラクタセットが別のキャラクタセットに、どのようにマッピングされるかを定義する。重要なことは、これがマッピングに名前を与える文だということだ。

```
<transliteration definition> ::=
CREATE TRANSLATION <transliteration name>
FOR <source character set specification>
TO <target character set specification>
FROM <transliteration source>
<source character set specification> ::=
<character set specification>
<target character set specification> ::=
<character set specification>
<transliteration source> ::=
<existing transliteration name> | <transliteration routine>
<existing transliteration name> ::= <transliteration name>
<transliteration routine> ::= <specific routine designator>
```

単純なマッピングは、入れ子のREPLACE関数と同じような使い方ができる。このマッピングに名前が付けられるおかげで、これをTRANSLATE関数で、入れ子のREPLACE関数の代わりに使うことができる。その構文は簡単だ。

```
TRANSLATE (<character value expression> USING <transliteration name>)
```

DB2 その他の実装では、TRANSLATE 関数を、探索対象の文字列と置換後の文字列を引数に取れるよう一般化している【訳注10】。これによって、たくさんの編集処理を1つの式で実現できる。この点については、文字列関数を扱うときにまた見ることにしよう（第14章参照）。

【訳注1】 この性質を閉包性（closure property）と呼ぶ。サブクエリの基礎となる数学的性質である。閉包性については、第20章でも取り上げているので、そちらも参照。

【訳注2】 この断固とした主張に、驚く人もいると思う。この著者の主張の当否をここで判断することはしないが、こうした「順序」を意識する構造が、本来、関係モデル的ではないという点は、本書を読み進めるうえでのキーポイントなので、留意しておいてほしい。

【訳注3】 表明を実装しているDBMSはまだ少ない。

【訳注4】 Oracleなどでは、トリガーの中でコミット／ロールバックを記述できないという制限があるため、このようなトリガーを実装することはできない。そのため、このスケルトンは、あくまで考え方の方針にすぎない。

【訳注5】 INSTEAD OFトリガーは名前のとおり、更新処理の「代わり」に実行されるトリガー。詳細は7.2節参照。

【訳注6】 ビューは特定の条件に該当すると更新不可能になる。UNIONの使用はその具体例の1つである。網羅的な条件のリストについては26.2節を参照。

【訳注7】　多くのDBMSでは、主キーのインデックスをスキャンすることでCOUNT関数の高速化を図っているため。

【訳注8】　キャラクタセット（文字集合）は、その名のとおり、特定の文字の集まりを指す。たとえば、英語で使われるアルファベットの文字集合は、{a, b, c ... z}という26種の文字から構成される（大文字と小文字を区別すれば、倍の52種となる）。

【訳注9】　たとえばUTF-16では、キャラクタセットを上位と下位の2つのバイト列として表現するため、どちらを先に並べるかを考える必要がある。上位8ビットが先頭に来るバイト順をビッグエンディアン、下位8ビットが先頭に来るバイト順をリトルエンディアンと呼ぶ。どちらの順序を採用しているかを示すマークをBOM（Byte Order Mark）と呼ぶ。

【訳注10】　DB2のTRANSLATE関数は、次のような構文を取ることができる。

TRANSLATE(<入力文字列>, <変換後文字列>, <変換対象文字列>)

面白いのは、文字のマッピングを複数定義できることだ。たとえば、次のようになる。

```
TRANSLATE('test', 'ab', 'st')='beab'
```

この例では、以下のマッピングを定義している。

s → a

t → b

4

ロケータと特別な数

Locating Data and Special Numbers

　SQLの実装は、データが物理的ストレージのどこにあるのかを示すためのベンダー独自機能、すなわちロケータを備えている。こうしたロケータのいくつかは、ハードウェアから情報を取得しているものもあれば、ハードウェアからは独立しているものもある。こうしたロケータの表現形式としては、順序数、乱数、特殊な性質を持った数列を使うのが一般的だ。

4.1

露出した物理的ロケータ

SQLではキーを利用する。このキーというのは、物理ストレージとは無関係な論理的概念である。だが残念なことに、センスのないSQLプログラマは、露出した物理的ロケータをハードウェアに生成させるような実装依存の機能を使おうとする。こうして生成された数は、ハードウェアにおけるあるイベントや位置を表すが、論理モデルとは一切関係ない。

このような露出した物理ロケータを、サロゲートキーと混同してはならない。コッドの言葉を借りるなら、「データベースユーザーはシステムにサロゲートキーを生成させたり削除させたりすることはできる。しかし、その値はコントロールできないし、そもそもユーザーにサロゲートキーの値が見えることはない･･･」【訳注1】。実際、大半の実装において、インデックスがどのように機能するか考えてみてほしい。それは、決してユーザーが直接アクセスするものではない。

4.1.1　ROWID と物理ディスクのアドレス

Oracleには、HDDなどの物理ディスクにおける行の物理アドレスをユーザーに見せるための変数ROWID がある。読み込みや書き込みのヘッドはダイレクトに行の位置へ移動するため、これを指定して行う検索は、テーブルにおける行の位置を特定する最速の方法である。この下層のストレージ情報を論理レベルで（つまりSQLで）見えるようにしているということは、Oracle は行を格納するうえで連続的なストレージの使用を前提としている、ということである。これ

が意味するのは、ハッシュや、分散データベース、動的ビットベクトルといった、比較的新しい大規模データベース技術のいくつかは、Oracleとは相性が悪いということだ。何らかの理由でデータベースが移動されたり再編成されたりすると、ROWIDの値は変わってしまう【訳注2】。

4.1.2 IDENTITY 列

テーブルに行が追加されるたびに、データベースエンジンはIDENTITY列と呼ばれる、自動的に一意な数列を生成する。一意に特定する必要のある行を追加するテーブルを作るときは、IDENTITY列をテーブルに含めることで条件を満たせる。行が一意な数値を持つことを保証するために、IDENTITY 列にユニークインデックスを定義するか、IDENTITY 列が主キーだと宣言するべきだろう。なお、テーブルを一度作ってからIDENTITY列を追加するようテーブル定義を変更することはできない。

もしテーブルに行を追加する際にIDENTITY 列に明示的に値を割り当てた場合、データベース内部で生成されるIDENTITY 列の値が更新されないため、次に追加するとき、テーブルにすでに存在する値と衝突する可能性がある。そのときIDENTITY 列に主キー制約や一意性制約によって一意性が強制されていれば、値の重複によってエラーメッセージが返される。IDENTITY 列のBNF記法の構文は以下のとおり。

```
<table definition> ::= CREATE TABLE <table name>(
<column name> INTEGER NOT NULL GENERATED [ALWAYS | BY DEFAULT]
AS IDENTITY (START WITH <start value>, INCREMENT BY <increment value>))
```

最初に追加される行には<start value>の値が使われる。そして、あとから追加される行には、<increment value>によって増分が指定される。GENERATED ALWAYSが定義されていると、IDENTITY列は常にデータベースエンジンによって生成され、アプリケーションが明示的に値を与えることはできない。一方、GENERATED BY DEFAULTが定義されていると、IDENTITY列にアプリケーションが明示的に値を与えることができる。アプリケーションが値を与えなければ、データベースエンジンが生成する。この場合、アプリケーション側では値をコントロールすることはできないので、SQLエンジンは値の一意性を保証することはできなくなる。GENERATED BY DEFAULT句は、既存のテーブルから中身をコピーしたり、既存のテーブルからデータをアンロードして、新しいテーブルに再登録するようなケースで使うことを想定した機能だ。

IDENTITY 列とシーケンスは似ているが、違いもある。IDENTITY 列は、次の条件を満たさなければならない。

条件1

IDENTITY列は、テーブルが作られるとき、DDLで定義する必要がある。テーブルを作成したあとに追加することはできない（ただし、既存のIDENTITY 列の定義を変更することはできる）。

条件2

Sybase/Microsoft T-SQLのIDENTITY 列は、単一のテーブルに対して自動的に値を生成する。IDENTITY列がGENERATED

> ALWAYSで定義されている場合、値は常にデータベースエンジンによって生成される。テーブルのデータを変更している間も、アプリケーションは、独自の値を与えることはできない。一方、ANSI のシーケンスは、テーブルとは独立のオブジェクトであり、いかなる目的でもどんなテーブルにでも使える。

IDENTITY 列は、INSERT処理が行われている間のマシンの物理的状態の一部を見えるようにする、という考えに基づいている。これは、コッドのリレーショナルデータベースのルールに違反している（コッドのルール8番：物理的データ独立【訳注3】）。また、エラーがあった場合の修正も極めて難しい。

昔のデータベース製品は、その当時あったファイルシステムの上に構築された。データは物理的に連続したページ、物理的に連続した行に保存され、物理的に連続した列から構成されていた――端的に言えば、パンチカードの束や磁気テープのイメージだ。IDENTITY 列やシーケンスのような自動インクリメント機能の多くは、SQLが追放したはずの物理的シーケンスを再登場させることを意図している。それによって我々は、あたかも連続的なストレージを相手にしているときのように考えることができる。

しかし、物理的に連続したストレージは、リレーショナルデータベースを構築する方法の1つにすぎないし、最善というわけでもない。それは別としても、そもそものリレーショナルデータベースの理念である、ユーザーはデータがどのように保存されているかを意識する必要などない、という方針に反している。ましてや、特定の製品の特定のリリースにおける特定の物理的表現に依存したコード

を書くなど、これほどリレーショナルデータベース的でないこともほかにない。

当然、リレーショナルデータベースの実装方法は個々の製品によって異なる。しかし、それらを使って得られる結果は（内部的な動作は予見不可能で冗長だとしても）皆同じでなければならない。もしすでにテーブルに適切なキーを持っているなら、IDENTITY列を付け加えるのは蛇足以外の何ものでもない。

かつて、こういう列を使ったほうが結合コストが「安上がりだ」という議論もあった。単純な整数を使うほうが、列長の長い列を使うより計算コストが低いから、IDENTITY 列を積極的にキーに使おう、と。だが、これは最近のRDBMSにはもはや当てはまらない。実際、ハッシュ的なアルゴリズムの中には、長い複合キーのほうが性能の良いものもある。キーが長くて複雑なほど、ハッシュによる分散も効果的だからだ。

自動インクリメント列をキーとして使うことの、もう1つの大きな欠点は、チェックディジットを持たないことだ。それゆえ、その値が妥当なのかどうか確認する術がない（チェックディジットの議論については、拙著を参照：『Data and Databases: Concepts in Practice』Morgan Kaufmann、1999）。

それなら、なぜ人々はこういう数列を使うのか？ システム生成の値を使うのが、主キーを作る一番手っ取り早い方法だからだ。これを使えば、何も調べなくてよいし、現実のデータモデルを考慮する必要もない。ドラッグを濫用するのが、人生の問題を解決する一番

お手軽な方法なのと同じだ。私は、どちらに手を出すこともすすめない。

Sybase/SQL Server ファミリーでは、IDENTITY 列に厳密な数列を生成する擬似列を持つことができる。これは、テーブルへ試みられた物理的なINSERTの回数を数えるものだ。「試みられた」という語に注意してほしい。INSERTに失敗したり、ロールバックされると、数列にギャップが生じる【訳注4】。これは（物理レベルには依存していないという点で）実にリレーショナルな機能だが、しばしば新米の、あるいはあまり有能でないSQL プログラマは、この機能を使って、テーブルを、レコードの位置によってデータにアクセスするシーケンシャルなテープファイルに見立てたり、非リレーショナルなファイルシステムやネットワークデータベースのポインタを真似たりする。これはいただけない。

さてそれでは、IDENTITY列の論理的な問題を詳細に見ていこう。2つの列を持つテーブルで、かつその2つの列がともにIDENTITY列であるようなケースを考えよう。もし特定のデータ型を持つ列を2つ以上宣言できないとしたら、それは定義からいっておよそデータ型と呼べるものではないのだが、Sybase/SQL Server ファミリーでは、IDENTITY 列を1 つのテーブルに1 つしか作れないという制限がある。

次に、1つのIDENTITY列と、それ以外の列a、b、cを持つテーブルFoobarを考えよう。このテーブルに対して、挿入、更新、削除を、それぞれ異なる回数行うことを考える。もしテーブルに挿入、

更新、削除を行うことができなければ、それは定義から言ってテーブルではない。

まず、次のような文を考える。

```
BEGIN
INSERT INTO Foobar (a, b, c) VALUES ('a1', 'b1', 'c1');
INSERT INTO Foobar (a, b, c) VALUES ('a2', 'b2', 'c2');
INSERT INTO Foobar (a, b, c) VALUES ('a3', 'b3', 'c3');
END;
```

これと、論理的に等価な次のSQL文を比べてほしい。

```
INSERT INTO Foobar (a, b, c)
VALUES ('a1', 'b1', 'c1'),
       ('a2', 'b2', 'c2'),
       ('a3', 'b3', 'c3');
```

あるいは次の文はどうだろう。

```
INSERT INTO Foobar (a, b, c)
SELECT x, y, z
FROM Floob; -- Floob テーブルには3行存在しているとする。
```

最初のコードにおいて、IDENTITY 列はINSERT 文が実行された順番に数を振り出していく。もし行を削除すれば、数列におけるギャップは埋められないまま、次に振り出される数は、最大値を基準に、そのテーブルでまだ使われていない数になるだろう。

2 番目と3 番目の例では、行が挿入される順番は自由であり決まっていない。3 番目のクエリの結果はテーブルであり、テーブルは定義から、いかなる順序も持たない。となると、IDENTITY 列の数はどうなるだろう？ Foobar には、一度に1 行ずつではなく、3 行揃った完璧な集合が、全体として一度に出現することになるのだ。n行を

順序づけるパターンはn!通りある。このケースなら、3!＝6通りだ。どのパターンを採用するべきだろう？ 答えは、結果セットが持つ何らか適当な物理的順序になる。ここでもまた「物理的順序」という非リレーショナルな言葉が再登場する！

だが問題はこれだけではない。もし同じINSERT文を2回実行したらどうなるだろう。ただし、統計情報を取り直したり、インデックスを作ったり削除したりしたあとに行うものとする。この場合、新しい実行計画が立てられて、同じ結果セットが異なる物理的順序で返されることになるはずだ。読者は、2 番目のクエリについて、実行するたびに各行に異なるIDENTITY 値が割り当てられることを、論理モデルから説明できるだろうか？ リレーショナルモデルにおいては、すべての行についてすべての属性が同一であれば、2つの結果セットは同じものとして扱わなければならないはずだ。

今度は次の文を見てほしい。1つのトランザクションにおいて、同じデータを削除して再び挿入しているだけだ。これは理論的に見て、データベースを同じ状態にとどめるはずである。

```
BEGIN
DELETE FROM Foobar
WHERE identity_col = 41;
INSERT INTO Foobar VALUES (<<identity_col = 41の行と同じデータ>>);
END;
```

しかしIDENTITY列の値は変化する。これと同じことは、2つの異なる行のデータを入れ替えるUPDATE文でも起こる。IDENTITY列はDMLによって変化しないからだ。

あるいは、とてもよく似た2つのデータベースにIDENTITY列が使われているケースを考えてみよう。違うのは、インデックスやキャッシュサイズといった、実行計画に影響する部分だけで、同じ文に対しては同じ結果を返す、そういうデータベースである。こういうデータベースにIDENTITY列が使われていたら、保守したり移行したいと思うだろうか？ 【訳注5】

対して、第3 章で見たCREATE SEQUENCE 文はどうだろう。これはSQL:2003 標準であり、IDENTITY列と混同しやすい。だが、シーケンスオブジェクトは次のような特徴を持っている。

特徴1 シーケンスは特定のテーブルには結び付いていないスキーマオブジェクトである。

特徴2 シーケンスは任意のSQL文の中で利用可能な数列を生成する。

シーケンスは任意のアプリケーションで利用できるオブジェクトである。特定のシーケンスにおいて、次の値を呼び出すには2つの式を使うことができる。1つはPREVVAL式で、同一セッションにおいて最も直近に生成された値を返す。もう1つのNEXTVAL式は次の値を返す。これらの式は複数のテーブルにいくつのSQLを実行していようとも、それで結果が変わることはない。

IDENTITY列とシーケンスの違いはこれだけではないが、上記の点を考慮すると、データベース設計とアプリケーションにおいて、どちらを使うべきかの判断に役立つだろう。

4.2

生成された識別子

データベース全体にわたって一意な識別子を生成するやり方はいくつかある。最も有名な方法は、Microsoft社の提唱したGUID（Global Unique Identifier）とオープンソースコミュニティの提唱したUUID（Universal Unique Identifier）である。

4.2.1 GUID

GUIDは、一意な露出した物理的ロケータであり、UTC時間とマシンのネットワークアドレスを組み合わせて作られる。16 バイトの2進値だが、一般には16 進で「1F2204F0-2A12-3212-8A1A-1214E72A2218」のようなフォーマットで表記する。Microsoft 社は、約一世紀の間は一意性を保てるはずだと言っている。Wikipediaの解説（http://en.wikipedia.org/wiki/GUID）によれば、次のとおりである。

> GUID を生成するアルゴリズムは広く批判を受けてきた。1 つには、GUID の一部の桁の生成にはユーザーのネットワークカードのMAC アドレスが使われるため、GUID から逆にマシンを追跡できるおそれがある。このプライバシーホールは、ワーム「メリッサ（Melissa）」の作成者の位置を特定するために使われた。この危険性が発見された後、Microsoft社は、MACアドレスを含まないようアルゴリズムを変更した。

露出した物理的ロケータが持つ一般的な問題に加えて、GUIDは16バイトの記憶領域を必要とするという問題もある。これに対し、単純なINTEGERならば4 バイトで済む。GUIDをもとに作られたインデックスと主キーは、列長の短い列を使ったときよりもパフォーマンスが悪い可能性がある。新米プログラマの多くは、GUIDキーを、パフォーマンスが良いという理由で正当化する。これは間違いであるうえに、そのレベルでのパフォーマンスは、最近のハードウェアをもってすれば、現実には問題にならない。64ビットのハードウェアが一般的になりつつあるいま、ハードディスクもまたどんどん高速化されている。

GUIDの本当の問題は、解釈が難しいことだ。直接意味を理解することが困難で、値の妥当性を確かめるために、ソースまでさかのぼるのも容易ではない。実際、GUIDはソートできる順序性を持っていないので、欠番があっても特定することはできないし、GUID列を結果を並べ替えるために使うことはできない。せいぜい、正規表現を使ったCHECK 制約によって、「36 桁で、4 つのハイフンによって区切られたセクションにA からFまでの文字および0 から9までの数字を含む」という条件を課しておくことができる程度だ。

GUIDはまた、集約関数を含むクエリに使うこともできない。使う場合は、まずCHAR(36)に型変換を行って、文字列として扱う必要がある。もしかするとGUIDを整数型に変換しようと思った人もいるかもしれないが、両者は変換可能ではないのも不便だ。ほかにも、GUIDには実装依存の機能があるため、Microsoft社の環境から他の環境へ移行することを難しくする。

4.2.2 UUID

UUIDはオープンソフトウェア財団（OSF）が分散コンピューティング環境の一部として標準化した。その目的は、分散システムで、中央集権的なコーディネーションがなくても、一意に情報を識別できるようにすることだ。UUIDはISO/IEC 11578:1996「Information technology—Open Systems Interconnection—Remote Procedure Call（RPC）」の一部としてドキュメント化されている。また最近では、ITU-T Rec. X.667 | ISO/IEC 9834-8:2005としてドキュメント化された。IETFはRFC 4122を公表しているが、技術的内容はITU-T Rec. X.667 | ISO/IEC 9834-8と同じだ。

UUIDには5 つのバージョンがある。UUIDを使えば、あるものを一意に識別できるし、その値が意図せずして、他のいかなる人によっても、他のいかなる目的のためにも使われることはない。UUIDは16バイト（128ビット）の数値であり、32個の16進数から構成されている。見た目上は、「8-4-4-4-12」のように、ハイフンによって区切られた5つのグループで表示される。合計は36文字である（32個の数字と4つのハイフン）。

UUID第1世代の仕組みは、これを生成するコンピュータのMACアドレスとナノ秒単位の時刻を使って計算するものだった。この仕組みだと、UUIDが作られたマシンとその時刻がわかってしまう。

そこで第2 世代は、第1 世代とよく似ているが、ローカルのPOSIXのユーザーID（UID）グループID（GID）ドメインを計算式に組み込んだ。

第3世代は、URL、完全修飾ドメイン名（FQDN）、オブジェクト識別子、その他のデータエレメントから、MD5を使って導き出す。

第3 世代のUUIDは、「xxxxxxxx-xxxx-3xxx-yxxx-xxxxxxxxxxxx」という形式の16進数の文字列である。MD5は、暗号化に広く使われているハッシュ関数で、128ビットのハッシュ値を生成する。インターネット標準（RFC 1321）でもある。

2004 年以降、研究者はMD5に多数の問題点を見つけた。今日では、米国の国土安全保障省は、MD5は「暗号理論的には破られており、今後の利用には適さない」と言っている。そのため、多くの米国政府のアプリケーションは2010 年までにSHA-2ファミリーのハッシュ関数に移行することが求められている。

第4 世代のUUIDは乱数のみに基づいて生成される。このアルゴリズムは、2つの予約されたビットにバージョン番号をセットする。残りのすべてのビットは、乱数または擬似乱数のデータソースを使ってセットされる。第4 世代のUUIDは、「xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx」という形式の16進数（x）と、同じく16進数の8、9、A、B（y）で構成される。

第5 世代のUUIDではSHA-1というハッシュ関数が使われる。それ以外の点では、第3 世代と同じである。RFC 4122は、第5世代は名前に基づいて生成する第3世代よりも好ましいと述べている。注意が必要なのは、160ビットのハッシングを行うSHA-1は、文字列長を合わせるために128ビットに切り詰められることである。

ちなみに、UUIDで重複値が発生する確率がどれくらいかというと、1 秒間に1 億個のUUID を生成することを向こう100 年間繰り返して、ようやく50%程度の確率で重複値のペアを1 つ得る、という

程度である。ただしこれは、UUIDのメカニズムが「正しく動いて」エラーがない、という前提のもとでの数値である。

4.2.3　業界標準の一意識別子

どのようなデータにおいても、妥当性確認（validation）と検証（verification）は重要な概念であるが、とりわけ識別子においては最も重要である。妥当性確認とは、データの値が正しい形式に則していることを確認することである。たとえば、私は米国の郵便番号がちょうど5桁で構成されることを知っている。そこで、郵便番号が正しい形式を取っているか確認する制約を、正規表現を使って簡単に作ることができる。

```
CONSTRAINT Valid_Zip_Code CHECK (zip_code SIMILAR TO [:DIGIT:]{5})
```

さらに私は、アラスカ州ケチカンの郵便番号が99950（ちなみにこれは最大の郵便番号だ）であり、ニューヨーク州のホルツビルが00501（こちらが最小の番号）であることを知っている。そこでさらに、妥当な郵便番号の取りうる範囲を制限する制約を追加しよう。

```
CONSTRAINT Valid_Zip_Code_Range CHECK (zip_code BETWEEN ‘00501’ AND ‘99950’)
```

チェックディジットを使えばさらに妥当性を確認する手段を追加できるが、これについては拙著『Data, Measurements and Standards in SQL』（Morgan Kaufmann、2009）で論じている。

検証は、信頼できる情報源にアクセスして識別子が本当にデータを識別できるかを確認することだ。たとえば、単純な範囲チェックによって、00000と99999は不適切な郵便番号だとわかるし、その郵

便番号が属する州を特定するための範囲も見つけることができる。しかし、たとえば75003という番号が2010年時点で発行されているかどうかを調べるには、米国郵政公社にアクセスしなければならない。

UUID、GUID、IDENTITYおよび他の自動インクリメント機能は、数式やカウンタによってローカルに生成されていた。これらの値は決まった形式を持っているため、妥当性を確認することができる。ただし、これらを検証してくれる当局は存在しない。したがって、もし業界標準識別子があるならば、その利用を検討する必要がある。

4.2.4　国防総省のUID

米国国防総省とNATOも、彼らの持つ5000ドル以上の物的資産に識別子を割り当てるというプログラムにおいて、よく似た問題に取り組んできた。彼らのアプローチは、UIDを使うことだ。もし5000ドル以上の部品であれば、それらはミッションクリティカル／予備／修理対象のいずれかに分類される。2004 年にUIDを使うことが義務化されたが、まだ対応は遅れている。この一意識別子は、UIDレジストリという巨大データベースに格納されている。

2007 年2月にサンディエゴで開かれたUIDフォーラムでは、国防総省が100 万個以上の資産に物品一意識別子（Unique Item Identifier：UII）が割り当て済みであり、UIDレジストリに登録されている、と公表した。このレジストリには最終的に1億個が登録される予定だ。このシステムは、既存の識別体系の上に構築されているが、独自のルールもある。ざっと挙げると以下のようなものだ。

- UIDの作り方1：UIDは発行する部局のコードと企業識別子、および企業内で一意なシリアル番号を含む
- UIDの作り方2：UIDは発行する部局のコードと企業識別子、および部品番号、バッチ／ロットのコードと部品番号内で一意なシリアル番号を含む
- 車両識別番号
- グローバルGlobal Returnable Asset Identifier（GRAI）
- グローバルGlobal Individual Asset Identifier（GIAI）
- 電気製品シリアル番号（典型的には携帯電話に割り当てられている）

これらの基準を満たした資産は、二次元マトリクスのバーコード（小さな格子盤に似たあれだ）で一意に識別されなければならない。すべての既存資産にUIDを割り当てる作業は、2010年までに終了する見込みである。このプログラムの詳細については、http://www.acq.osd.mil/dpap/pdi/uid/index.htmlで知ることができる。

4.2.5　検証するための情報源

一般に、識別子を検証できるのは、それを管理している団体である。こうした団体は、標準化委員会、コンソーシアム、国家機関あるいは国際機関、およびそれらのローカルなエージェントを務める企業であったりする。以下に覚えておいて損はない識別子の例を挙げよう。

DUNS

DUNS（Data Universal Numbering System、ダンズ）は米国のダンアンドブラッドストリート（D&B）社が管理する数値の企業識別コードである。これは1 つの企業に対して1 つ割り当てるコードで、1965 年から使われている。もとは同社が企業の信用報告業務を行う際の一環としてスタートしたものだが、いまでは標準的な国際基準になっている。これは国税の識別子とは違うので注意してほしい。たとえば、デルコンピュータ社は、すべての供給業者とコンサルタントに対して、DUNSを使った請求支払いを要求している。企業が米国行政管理予算局（OMB）に公共予算を新しく申請する場合は、DUNSを求められる。DUNSは国連でも使われており、米国内だけで、1億6000万個が使われている。

DUNSは9桁のランダムな数字で、2006年まではMOD 10のチェックディジットを持っていた【訳注6】。しかし、コードの桁数が増えたため、この機能はいまは廃止されている。無料で取得できるので、もし読者がほしいと思えば、オンラインで申請できるが（http://www.dnb.com/us/duns_update/index.html）、少し時間がかかる。もしすぐに払い出してほしい場合は、有料で対応してくれる。

DOI（デジタルオブジェクト識別子）

DOI（Digital Object Identifier）は、インターネット上のコンテンツオブジェクトを識別する手段である。DOIコードはネットワーク上の任意のエンティティに対して使われる。この識別子には、インターネット上のどこで見つけることができるかといっ

た、エンティティの情報が含まれている。URLなどデジタルオブジェクトについての情報は、時間とともに変わる可能性があるが、DOI名は変わらない。国際DOI財団（http://www.doi.org/）をはじめ、営利、非営利のパートナーを持つオープンメンバーシップのコンソーシアムによって管理されており、現在ISO標準になる動きがある。本書執筆時点で、米国、オーストラリア、ヨーロッパで約4000万のDOI名が割り当てられている。

ISAN（国際標準視聴覚番号）

ISAN（International Standard Audiovisual Number）は、視聴覚作品に対する標準的な識別子であり、書籍におけるISBN（International Standard Book Number）や、定期刊行物に対するISSN（International Standard Serial Number）に相当するものだ。ISO 15706:2002 およびISO 15706-2で規格化されている。

GTIN（国際取引商品番号）

GTIN（Global Trade Item Number) は、小売業の商品についている商品識別コードの総称で「ジーティン」と読む。北米では、UPCというGTIN の一種がよく利用されている。1970年に、全国フードチェーン協会（NAFC）が、食料雑貨品の識別コードであるUGPICの一部を定め、その後、米国スーパーマーケット暫定委員会が正式にUGPICを規定した。同委員会が現在のUPC体系を定義したのは1973年である。

ISIN（国際証券コード）

ISIN（International Securities Identification Number）は、ISO 6166に従って証券類を一意に識別する（「アイシン」と読む）。ISIN は、債券、商業手形、株券、ワラント債などに割り

当てられる。12 桁のアルファベットと数字から成る文字列によって構成される。ISINが識別するのはあくまで証券類であって、それが使われる取引ではないし、また銘柄のティッカーシンボルとも異なる。世界中の異なる株式市場で取り引きされる（したがって異なる通貨で値付けされている）同じ株券は、ティッカーシンボルは異なっていても、同じISINコードを持つ。株式市場を識別するコードはまた別にあって、MIC（ISO 10383, “Codes for Exchanges and Market Identification”）という4桁のコードを使う。

4.3

重複行について

　コッドのリレーショナルモデルは重複行を許していない。それは、このモデルが集合論に基づいて作られたからである。しかしSQLは常に重複行を許してきたし、重複集合モデルに基づいてきた。重複行の問題がSQL標準化委員会で持ち上がったとき、我々は、これを標準において許すことに決めた。そのとき、重複行を許すべき局面として取り上げられた実例が、「キャットフード問題」である。これはレン・ギャラガーが『Database Programming & Design』誌に送付した手紙で使われていたもので、デヴィッド・ビーチも『Datamation』誌に対する手紙の中で使っていた。これは、キャットフードの缶詰がレシートの中に繰り返し現れることをどうリレーショナルモデルで取り扱うかという、その名のとおりの問題である。

　根本的な問題は、「いったいテーブルで何をモデリングしているのか？」ということである。コッドとクリス・デイトは「テーブルは事実の集合である」という立場である。これと異なる立場としては、「テーブルはエンティティ、クラス、エンティティ間の関連を表すこともできる」というものもある。後者の立場によれば、重複行は、1つのエンティティの複数回の出現を意味する。これは前者の立場よりもオブジェクト指向的なデータの見方である。ここで、「重複」と呼ばれる現象同士の、いくつかの基本的な関連を確認しておかなければならない。

- 同一性（Identity）：「クラーク・ケントはスーパーマンだ！」という文が指し示すエンティティは1つだけである【訳注7】。つまり、ただ1人の人間である。ただ、それの表現が複数あるだけだ。こうした表現は、相互に互換可能ではない（クラーク・ケントはスーパーマンに変身するまでは空を飛べない）。
- 相等性（Equality）：「2 足す2は4 である」という文が指し示すエンティティは1つだけである。かつ、先の例と違って、常に互換可能である。
- 同値性（Equivalency）：「新しい洗剤は半分の量で従来のものと同じだけの洗浄力です！」。この文には、古い洗剤と新しい洗剤という2つのエンティティが現れている。両者はいかなる条件においても互換可能なものである。
- 互換可能性（Substitutability）：「ジンが切れてしまった。マティーニのベースはウォッカでもいいかな？」。この文は2つの異なるエンティティを指し示しているが、必ずしもあらゆる条件下でも同じ方向性を指すとは限らない。ワインがないときにウォッカを代わりに飲むのはかまわないだろうが、ワインソースを作るときにウォッカを使う人はいないだろう。

後にコッドは、重複行を扱うための方法として、「重複度（dod：degree of duplication）」演算子をモデルに追加した。重複には度合いという情報が含まれていることに気づいたからだ。重複度は、`COUNT(*)`の値や、何らかの量を表す列の値とは必ずしも一致しない。それは、数値型の列と同じようには振る舞わないからだ。たと

えば、テーブルAがあったとして、dodを各行に対する重複度を表す演算子だとする。

A

x	y
1	a
2	b
3	b

　このテーブルAに対して射影を行えば、コッドのモデルに従って重複行を排除できるが、dod 関数を使ってオリジナルのテーブルを再構築することもできる。

A

y	dod
a	1
b	2

　違いがわかるだろうか？ これは演算子であって、値ではない。コッドは次のように言っている。「私が重複行を使うのは、レガシーデータをリレーショナルデータベースにロードするときだけだ。レジの明細のような現実世界からデータを取得する場合には、重複行は非常によくある」

　私の意見では、重複行を結果に残しておくのはかまわないと思う。なぜならば、SELECT DISTINCTを使って重複行を取り除くの

は、コストが高く時間がかかる処理だし、ワークテーブルの中で結果を並べ替えてしまうと、以降の処理で利用するときにパフォーマンスが悪くなる。

コッドは、この例を「スーパーの会計問題」として取り上げている（*The Relational Model for Database Management: Version* 2, Addison-Wesley, 1990, Section 23.02.5, pp. 378–379）。またコッドはこの問題に関連して、ディヴィッド・ビーチが『Datamation』誌に寄稿した記事「テーブルにおける重複行の必要性」（*Datamation*, January 1989）を高く評価している。

4.4

シーケンス生成関数

COUNTER(*)、NUMBER(*)、IDENTITY、およびそれに類似する実装依存の機能は、関数が式の中で使われるたびに新しくインクリメントされた値を返す。これは一意な識別子を生成する方法の1つである。こうした機能は、実装によって関数呼び出しであったり、列の属性だったりする。この機能もまた、非標準で、非リレーショナルで、実装依存という感心しない機能であるため、可能な限り使わないでほしい。

実装依存のコードやハードウェアの露出した物理的ロケータに頼らず、標準SQLでシーケンスと一意な数を得る方法については、あとで詳細に見ていく。

4.4.1　一意な値の生成

一意な値を生成する方法にとって、最も重要な特性は、それが決して同じ値を二度生成しないことである。適切なキーの代わりとして、ベンダーがこの機能を実装するとき、最初に候補になるのが数列である。

要するにこの機能は、割り当てられた最後の値を調べて、その次の値を返すようなSQL のコードである。そういうプロシージャを、我々も簡単なサンプルから作ってみよう。

まずは、1行1列のGeneratorValuesというテーブルを作る。

```
CREATE TABLE GeneratorValues
(key_val INTEGER DEFAULT 1 NOT NULL -- 正数のみ
      CHECK (key_val > 0));
```

GeneratorValues：数列生成テーブル　　key_val：キー値

```
INSERT INTO GeneratorValues(key_val) VALUES (1);

-- すべてのユーザーが使えるようにする
GRANT SELECT, UPDATE(key_val)
   ON TABLE GeneratorValues
   TO PUBLIC;
```

次に、値を取り出してインクリメントする関数が必要だ。

```
CREATE  OR REPLACE FUNCTION Generator
RETURN INTEGER
IS
  intval INTEGER DEFAULT 0;
BEGIN

--SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE GeneratorValues
   SET key_val = key_val + 1;

SELECT key_val INTO intval FROM GeneratorValues;
COMMIT;

RETURN intval;
END;
```

「これでいいじゃないか」と思う人もいるかもしれないが、この単純なコードだと、複数ユーザーがいた場合、異なるユーザーに重複値を割り当ててしまう可能性がある。これを防止するために「SET TRANSACTION ISOLATION = SERIALIZABLE」を使うことで、一度に1 人だけのユーザーしかコードを実行できないように制限することが重要である。

トランザクションを分離する手段は、各DBMSの持つ同時実行制御の方法によって微妙に異なる。だがそれより問題なのは、悲観的ロックのシステムでは、トランザクションが直列に分離されると、ロ

ック競合によって深刻なパフォーマンス問題が起きることである。ユーザーはGeneratorValuesテーブルを利用するために、一列に並ぶことになる。

もしアプリケーションの仕様上、欠番が起きないナンバリングが必要ということであれば、我々は、2 つのセッションが同時に同じ値を取得しないよう保証するだけでなく、飛び石にならないことも保証しなければならない。それゆえ、GeneratorValues テーブルへのロックは、key_valが実際に使われて、トランザクションが全体がコミットされるまで保持されなければならない。これを厳密に実現するには、実装を定義する必要があるので、ここではこれ以上のコメントは控える。

4.4.2　欠番のある連番

連番に欠番があったとしても困らないアプリケーションも多い。目的が単に一意性の確保というだけであれば、欠番は問題にならないからだ（実のところ、一意性だけが問題なら、順序関係さえ保存する必要がない）。同様に、生成された値がただのソートキーであれば、やはり欠番があってもかまわない。単調増加することが保証されていれば十分だ。

というわけで、欠番を許容できるということになれば、値を生成したら、「本体の」トランザクションが開始される前に、即座にコミットを発行することができるようになる。というのも、もし「本体の」トランザクションがあとでロールバックされたり、あるいはそもそも開始されなかったとしても、それに割り当てられた番号は捨てればよいからだ。我々は、別にその番号を惜しむ必要はない。

コミットを即座に発行できることのメリットは、採番テーブルのロック時間を短くできることだ。これが最善の方法とは言いがたいが。

4.5

事前に割り当てられる値

まだ仕事で紙の様式を使っていた時代の話だ。企業や組織には、様式管理者という仕事があった。その内容は、様式の作成、発行、追跡であった。伝票、債券、株券、あるいは何であれ会計上の書類のシーケンシャルな数字に欠番があることは、会計上重要な問題であった。紙の様式は、たいてい事前に印刷され、必要なだけの数を割り当てて発行されていた。ちょっと考えれば、簡単なストアドプロシージャによって、このやり方をデータベース上でも真似することができる。

一般的には、数字の割り当て方には2つのタイプがある。1つ目は、数列が既知の場合である。2つ目はセキュリティ上の理由からランダムな数列を用いる場合である。

前者の最もポピュラーな事例は小切手帳である。数列における歯抜けは許されず、券が破れたり傷ついたりしたときは、「欠番」などの表記を備考欄に記入しておく必要がある。

システムは、どの券がどのユーザーに渡ったかを、日時やその他監査に必要な情報とともに記録しておく必要がある。

```
CREATE TABLE FormsControl
(form_nbr CHAR(16) NOT NULL,
 seq INTEGER NOT NULL CHECK(seq > 0),
     PRIMARY KEY (form_nbr, seq),
 recipient CHAR(25) DEFAULT CURRENT_USER NOT NULL,
 issue_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL);
```

FormsControl：様式管理テーブル　form_nbr：様式番号　seq：連番
recipient：受付者　issue_date：発行日

様式番号を使うテーブルには、「番号がFormsControlテーブルで発行され、そのテーブルに存在すること」という制約を付ける必要がある。次のシーケンス番号を作るのは簡単だが、一種類の様式だけを操作するストアドプロシージャによるアクセスだけに制限する、といった工夫も必要になるだろう。

```
CREATE OR REPLACE FUNCTION NextFlobSeq
RETURN INTEGER
BEGIN
INSERT INTO FormsControl (form_nbr, seq)
       VALUES ('Flob-1/R',
                  (SELECT MAX(seq)+1
                     FROM FormsControl
                    WHERE form_nbr = 'Flob-1/R'));
COMMIT;
END;
```

FormsControl テーブルへのアクセスを制限するために、ビューを使うこともできる。もしFormsControlテーブルが空の場合も考慮するとすれば、次のようなスカラ式を使うのがよいだろう。

```
(SELECT COALESCE(MAX(seq), 0)+1
   FROM FormsControl
  WHERE form_nbr = 'Flob-1/R')
```

FormsControlテーブルが空の場合、COALESCE関数は0を返す。これによって、テーブルが空の場合に、seq列を1から始められる。

4.6

特別な数列

数列は、識別子、暗号化などの用途に非常に有用な性質を持っている。本節では、基本的な数列として、素数と、順序のランダムな数列について取り上げる。

4.6.1　数列テーブル

数列テーブルを用意することは、SQLプログラミングのテクニックとしては常套手段である。しかし、使う人によってテーブルの名前は異なる。私自身、本書の以前の版では「Sequence」という名前を使っていた。しかし、この単語は標準SQL の予約語になったので、この版では「Series」という名前に変更している。また「Numbers」とか、そういった名前を使う人もいる。

```
CREATE TABLE Series
(seq INTEGER NOT NULL PRIMARY KEY
     CHECK (seq > 0));
```

このテーブルにデータを入れる方法はたくさんあるが、私は次のようなやり方が好きだ。

```
INSERT INTO Series(seq)
WITH Digits(i)
AS (SELECT i
      FROM (VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (0)) AS X(i))
SELECT (D3.i * 1000 + D2.i * 100 + D1.i * 10 + D0.i + 1) AS seq
  FROM Digits AS D0, Digits AS D1, Digits AS D2, Digits AS D3;
```

このテンプレートは拡張が簡単だし、SELECT 句の「+1」を取り除けば0 から始まる数列を作るのも簡単だ【訳注8】。

4.6.2　素数

私は、2009年7月にカーネギーメロン大学のYAPCクラスでSQLを教えたことがある（YAPC は「Yet Another PERL Conference」の略で、「ヤプシー」と読む）。正直言って、私はそれまでPERLを使ったことがなく、大学へ出向く前にGoogleでそれがどんな言語なのか調べなければならなかった。PERLはSQLとはまったく異なった言語である。生徒の1 人は、私に1000 以下の（あるいは任意の閾値以下の）素数を生成するSQL 文を書くことはできるか、そしてそれはスケールするか、とたずねてきた。彼はSQLにループがないことに戸惑っていた。

素数を生成するためのアルゴリズム「エラトステネスのふるい」を実装するのは、有名なPERLプログラミングの練習だ。その動作イメージのアニメーションをWeb 上で見ることができる（http://www.hbmeyer.de/eratosiv.htm）。また、PERL のコードサンプルも見つけられる（http://www.perlmonks.org/?node_id=276103）。

素数を生成するうえで、役に立つ事実が2つある。

事実1

与えられた数n の素因数はceiling ($\sqrt{n}$)以下である。定義から、$\sqrt{n} * \sqrt{n}$ = n である。またこれも定義から、ceiling ($\sqrt{n}$)>= floor ($\sqrt{n}$)。したがって、整数に丸めても安全だ。ここからわかることは、もし「a * b = c」（ただしa < b）という式を調べれば、「(b * a = c)」という式は見なくてもよいということであ

る。それゆえ、素因数を小さい値から探し始めればいいことがわかる。

事実2

3より大きいすべての素数は「6 * n ± 1」という数式で表される。ただし、こういう数式のすべてが素数であるわけではない。たとえば、n = 1のとき、この数式の解は5と7になり、両方とも素数である。しかし、n = 4のときは23と25になり、25 = 5 * 5であるから、素数ではない。この数式からわかることは、2や3 の倍数は素数の候補から除外してよいということだ。

それでは以上のことをSQL文で表現してみよう。次のテーブルを作ることから始めよう。

```
CREATE TABLE Primes
(p INTEGER NOT NULL PRIMARY KEY
    CHECK (p > 1));
```

さて、読者の仕事は、このテーブルに特定の上限（話を簡単にするため、1000としよう）までの素数を入れていくことだ。ここで、先ほど作ったSeriesに、すでに1000までの数列が入っていて、それを使えると仮定する。

方法1

最初に、事実2の数式を使って、Primesテーブルに素数の候補となる数を入れていく。

```
INSERT INTO Primes (p)
(SELECT (6 * seq) + 1
FROM Series
WHERE (6 * seq) + 1 <= 1000
UNION ALL
SELECT (6 * seq) - 1
FROM Series
WHERE (6 * seq) - 1 <= 1000);
```

これは事実2 の数式を正直に表現したものだが、UNION ALLを除外してパフォーマンスを改良した、テーブル定数を使う方法もある。

```
INSERT INTO Primes (p)
SELECT (6 * seq) + X.switch
  FROM Series
         CROSS JOIN
       (SELECT switch
          FROM (VALUES (-1), (+1)) AS Foo(switch)) AS X(switch)
 WHERE (6 * seq) + X.switch <= 1000;
```

だが、これだけではまだ素数以外の数もPrimesテーブルに存在する。ここから、余計な非素数を除外していこう。ここで事実1 が役に立つ。平方根よりも小さい数の集合に対して、素因数が存在するかテストするのだ。MOD関数は剰余を求める。

```
DELETE FROM Primes
WHERE EXISTS
      (SELECT *
         FROM Primes AS P1
        WHERE P1.p <= CEILING (SQRT (Primes.p))
          AND MOD (Primes.p, P1.p) = 0);
```

これでPrimesテーブルには素数だけが残る。

方法2

Primesテーブルに素数の候補をロードする別の方法を考えよう。既知のいくつかの素数を、あらかじめSQL文の中に埋め込むやり方だ。事実2の数式は、2と3の倍数を除外していたが、この方法はその一般化である。MOD関数は剰余を求めるので、戻り値が0の数を除外することで、入力数の非倍数を求められる。ただし1だけは例外なので、「seq <> 1」の条件が必要になる。

```
INSERT INTO Primes (p)
SELECT seq
  FROM Series
 WHERE 0 NOT IN (MOD(seq, 2), MOD(seq, 3), MOD(seq, 5), MOD(seq, 7), ...)
   AND seq <> 1
   AND seq <= 1000;
```

素数の候補となる集合を絞り込めれば絞り込めるほど、パフォーマンスが良くなる。極端な話、もしMOD(seq, <素数>)の式を、上限値まで増やしてしまえば、即座に解が得られる。

これは面白いトリックである。多くのSQLプログラマは、INのリストに指定できる引数は定数だけだと思っている。しかし実際は、式を書くこともできるのだ。あとはリストに指定できる引数の個数の上限値がどの程度かだが、これは読者が思っているよりも大きいと思う【訳注9】。

方法3

候補を絞り込むもう1 つのトリックは、下一桁が {2, 4, 6, 8, 0} で終わる整数は偶数、{5, 0}で終わる整数は5 の倍数という事実に基づいたものだ。これらの数は候補からはずしてかまわないというわけだ。

```
INSERT INTO Primes (p)
WITH Digits(i)
        AS (SELECT i
              FROM (VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (0)) AS X(i)),
             -- 下一桁を制限した共通表式
             Units(i)
        AS (SELECT i
              FROM (VALUES (1), (3), (7), (9)) AS X(i))
SELECT (D3.i * 1000 + D2.i * 100 + D1.i * 10 + Units.i)
  FROM Units, Digits AS D1, Digits AS D2, Digits AS D3;
```

方法4

さらに別解として、すべての非素数を生成して、それらを数列テーブルから除外するという方法もある。

```
INSERT INTO Primes (p)
SELECT seq FROM Series WHERE seq <= 1000
EXCEPT
(SELECT (F1.seq * F2.seq) AS composite_nbr
   FROM Series AS F1, Series AS F2
  WHERE F1.seq BETWEEN 2 AND CEILING (SQRT (1000))
    AND F2.seq BETWEEN 2 AND CEILING (SQRT (1000))
    AND F1.seq <= F2.seq
    AND (F1.seq * F2.seq) <= 1000);
```

EXCEPTの上のSeriesテーブルは、これまでも作ってきた、素数の候補となる整数の集合である。そこから、EXCEPTのもとで作られた整数の集合を除外している。この除外型のアルゴリズムには、「アトキンのふるい」や「車輪のふるい」など、より複雑だが高速なものもある。

4.6.3　順序のランダムな数列

アプリケーションにおいて、数列を規則的に発行したくない場合は多い。そのパターンが、我々が明かしたくない情報を与えてしまうことがあるからだ。そのような場合は、数列をランダムな順序で

発行することが望ましい。これを、乱数と混同しないでほしい。既知の値をランダムな順序で生成したいのであって、どんな値が出てくるか予測できない乱数がほしいわけではない。だが、たいていの乱数発生器は、同じ値を何度でも生成する。それでは本節の目的には沿わない。

私は極力、物理的実装には言及しないようにしているが、順序がランダムなキーの利点の1つは、ツリーインデックスのパフォーマンスを改善できることだ。Bツリーのような木構造のインデックスは、シーケンシャルなキーを繰り返し挿入すると、木のバランスが崩れて頻繁に再編成を行わなければならない。だが、同じキーの集合を挿入するにしても、ランダムな順序で挿入すると、木はバランスを保ったままで、パフォーマンスが良くなる傾向がある。

ここで示すジェネレータは、擬似ランダムな順序で値を生成する加法合同法を使った方法の1 つであり、ラショナルコマース社のロイ・ハンの考えた方法である。この方法は、シフトレジスタとXOR（排他的論理和）ゲートを基礎としていて、もともとは暗号のために考えられた。ほかにも同じことを実現する方法はあるが、このコードには以下のような利点がある。

利点1　このアルゴリズムはCで書くことができる。高速化のために他の低水準言語で書くことも可能だが、計算は10進法においてもかなり単純である。

利点2　アルゴリズムは、（たいていは）「非常にかけ離れた」一連の値を生成する。これはインデックスのパフォーマ

ンス改善に効果的である。データをストレージ内の離れた物理ページに配置することが見込めるからだ。

利点3 アルゴリズムは、生成できるすべての数を生成し尽くすまで循環しない。それゆえ、重複値の心配がない。ジェネレータが何回呼ばれたかをカウントしていればよい。

利点4 アルゴリズムは一様に分散した値を生成する。これは便利な数学的性質だ。また、このアルゴリズムでは0は生成されない。

図4.1で描かれている4ビットのジェネレータを、すべての数を生成するまで繰り返し実行してみよう。

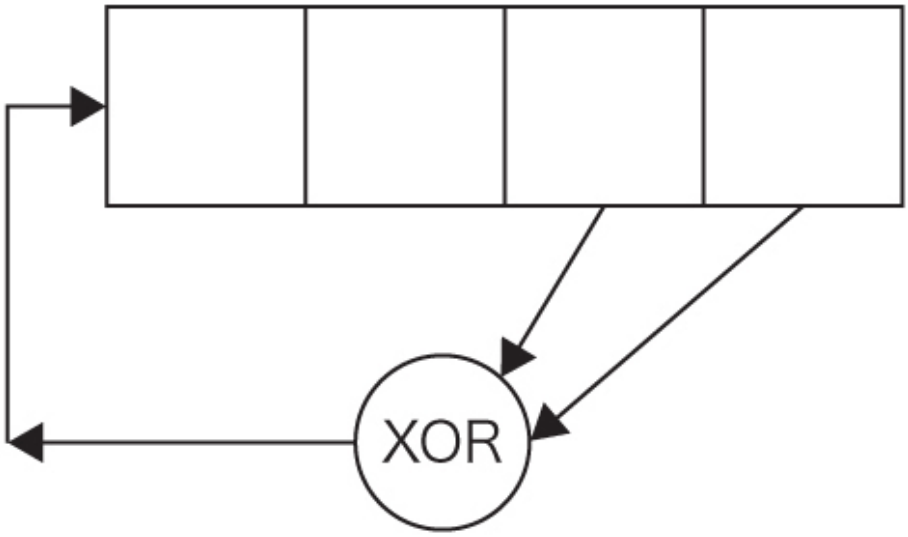


図4.1 四桁の加算合同法の生成値

シフトレジスタの初期値は「0001」となる。右端の2つのビットは、ともに排他的論理和の演算が行われ、1を返す。この結果は左端のビット位置の入力となり、最初にそこにあったレジスタは1ビット

右にシフトする。レジスタの繰り返しを、10進数表現とともに以下のテーブルに示すと次のようになる。

```
iteration 1: 0001 (1)
iteration 2: 1000 (8)
iteration 3: 0100 (4)
iteration 4: 0010 (2)
iteration 5: 1001 (9)
iteration 6: 1100 (12)
iteration 7: 0110 (6)
iteration 8: 1011 (11)
iteration 9: 0101 (5)
iteration 10: 1010 (10)
iteration 11: 1101 (13)
iteration 12: 1110 (14)
iteration 13: 1111 (15)
iteration 14: 0111 (7)
iteration 15: 0011 (3)
iteration 16: 0001 (1)  ⬅一周した！
```

この4 ビットの例だと小さすぎて、一連の値が互いに離れているかよくわからない。しかし、値が簡単にわかる順序で生成されていないことは明らかだ。0を除く16までのすべての値が生成されている。終了条件も明白で、ジェネレータが1に戻るときである。

このアルゴリズムを適当な2進の語長まで一般化し、それによってもっと長い数列を生成するようにすることは、思ったより簡単ではない。フィードバックのための「タップ」位置は語長によって変わるし、その変わり方もまったく明らかでない【訳注10】。不適切なタップ位置を選んでしまうと、たいてい非常に短い不完全なサイクルになってしまって使いものにならない。もし1から100ビットの語に対するタップ位置について詳細を知りたいならば、以下の資料を参照してほしい。

E. J. Watson, "Primitive Polynomials (Mod 2)," *Mathematics of Computation, v.*16, 1962, pp. 368–369

　下の表はそれぞれ、8ビット、16ビット、31ビット、32ビット、64ビットの語のタップ位置を示している。これは任意のハードウェアで動作するはずだ。31ビットの語が一番有用であろう。このケースでは、数百億の数を生成するのに、タップ位置は2つしか使わないので計算が簡単で、大半のハードウェアに適している。32ビットバージョンを、32ビットマシン上で実装することは容易でない。たいていはオーバーフローエラーが発生してしまうからだ。

語長

8 = {0, 2, 3, 4}
16 = {0, 2, 3, 5}
31 = {0, 3}
32 = {0, 1, 2, 3, 5, 7}
64 = {0, 1, 3, 4}

　上の表を使うと、31ビットのランダム順の数列を生成するには、0と3 のビットを必要とすることがわかる（この31ビット版が一番よく使われるだろう）。

```
UPDATE Generator31
   SET key_val = key_val/2 + MOD(MOD(key_val, 2) + MOD(key_val/8, 2),2) * 2^30;
```

　あるいは、アルゴリズムをCで書きたければ、次のようになる。

```
int Generator31 ()
{static int n = 1;
        n = n >> 1 | ((n^n >> 3) & 1) << 30;
 return n;
}
```

4.6.4　その他の数列

その他にも有用な数列がある。第6章ではフィボナッチ数列を取り上げる。この数列には多くの利用事例があり、暗号、チェックディジットなどで有用である。

最近では、自前のコンピュータで計算するよりも、インターネットから数列の表をダウンロードするほうが簡単だったりもする。身も蓋もないが、有用なものは利用させてもらおう。

【訳注1】　Codd, E. F. "Extending the Database Relational Model to Capture More Meaning, " *ACM Transactions on Database Systems*, 4(4), 1979, pp.409-410.

【訳注2】　Oracleの公式ドキュメントによれば、ROWIDについては以下のような注意が喚起されている。

> 表の主キーとしてROWIDを使用しないでください。たとえば、インポートユーティリティとエクスポートユーティリティで行を削除してから再挿入する場合、ROWIDが変わる場合があります。行を削除した場合、Oracleは、後から新しく挿入される行にそのROWIDを再度割り当てる可能性があります。
>
> 出典：Oracle Database SQL言語リファレンス, 11gリリース2（11.2）

> http://docs.oracle.com/cd/E16338_01/server.112/b56299/pseudocolumns008.htm

【訳注3】 コッドはリレーショナルデータベースが満たすべき12のルールを定めた。物理的データ独立はそのうちの1つで、データベースを扱うアプリケーションは、データの物理的レベルにおける表現形式に依存せず、物理的変更があってもアプリケーションは影響を受けない、というルールである。

【訳注4】 SQL Serverでは、テーブルの行を削除した場合にも数列にギャップが生じる場合がある。以下のMicrosoft 社のドキュメントを参照。

> ID 列を持ったテーブルに対して頻繁に削除を行うと、ID 値間でギャップが生じる可能性があります。これが問題ならば、IDENTITYプロパティは使用しないでください。
> 出典：SQL Server 2012 IDENTITY (プロパティ) (Transact-SQL)
> http://msdn.microsoft.com/ja-jp/library/ms186775.aspx

【訳注5】 このような「よく似た2つのデータベース」を、読者の多くも実際に保守した経験があると思う。たとえば「試験環境」と「本番環境」など。これに「開発環

境」まで含めれば、データベース環境は2つに限らず、もっと増える。こういうレプリカ環境間の移行などを行うときに、IDENTITY列やシーケンスの引き起こす問題を経験した人もいるかもしれない。

【訳注6】 MOD 10はIBMに勤めていたハンス・ピーター・ラーン（Hans Peter Luhn）が発明したチェックサム方式で、Luhnアルゴリズムとも呼ばれる。クレジットカード番号やカナダの社会保障番号で使われており、チェックディジットを含む数が正しいかどうかをある程度検証できる。

【訳注7】 この例は、「クラーク・ケントはクラーク・ケントだ！」という文とセットで考えたほうがよい。私たちは、この文が正しいとわかる。クラーク・ケントが誰だか知らなくても、彼がスーパーマンに変身する新聞記者であることを知らなくても、正しいとわかる。それは、エンティティの状態によらず同一のエンティティを指す文だからだ。一方、「クラーク・ケントはスーパーマンだ！」が正しいとわかるには、彼がスーパーマンに変身できることを突き止めたうえで、彼がスーパーマンになっても人格的にクラークと連続した人間であると確認できたあとである。

【訳注8】 このSQLを最初見ると驚くかもしれないが、やっていることは、0～9までの数字を使った直積である。「D0.i」が1の位、「D1.i * 10」が10の位…というよ

うに、それぞれの「桁」の数字の網羅的な組み合わせを作っている。

【訳注9】 INの引数の個数の上限値は実装依存だが、たとえばOracleでは1000個である。以下のOracle社のドキュメントを参照。

- Oracle Database SQL言語リファレンス 11gリリース2 (11.2)
  http://docs.oracle.com/cd/E16338_01/server.112/b56299/conditions013.htm

【訳注10】 タップとは、次の入力ビットに影響を与えるビットのこと。タップは順次XOR演算が行われる。

5

基底テーブルとそれに関連する要素

Base Tables and Related Elements

かつて、SQLにはたった1つのデータ構造しか存在しなかった。テーブルである。以降、標準SQLと各ベンダーは、他の構造も追加していったが、それらは滅多に使われなかったり、他の環境に移植できないようなものだ。テーブルの形はさまざまだが、いずれも同じように振る舞う。概念的には、テーブルは0 行以上の行の集合であり、行は1 つ以上の列の集合である。各列は特定のデータ型と制約を持っており、属性の値に対する抽象的なドメインを構成している。テーブルの物理的な実装がどうなっているかはあまり重要ではない。なぜなら、テーブルにアクセスする唯一の手段は、キーを使ったSQLによるものだからだ。細かいことはすべてデータベースエンジンがやってくれるので、物理ファイルを扱うときのように、内部について気にかける必要はない。実際、どのDBMSも内部構造の実装はバラバラである。

ファイルシステムに慣れたプログラマが犯しがちな概念的な間違いが2つある。1つ目はテーブルをファイルだと考えることである。2つ目はテーブルをスプレッドシートだと考えることである。テーブルはこのどちらとも異なった振る舞い方をする。このテーブルという基本的な概念を理解しておかないと、のちのち驚くことになるだろう。

テーブルをファイルだと想像することは簡単だ。行はレコードで、列はフィールドに相当する。どちらも馴染み深い概念だし、データがSQLからホスト言語に渡されるときは、データをホスト言語のデータ型とデータ構造に変換しないと、表示したり編集したりできない。

ファイルシステムを扱うときと、SQLを扱うときの大きな違いは、SQL がホスト言語に合わせて動くときの動作にある。ファイルシステムを使うときは、個々のプログラムでファイルのオープンとクローズを行う必要がある。だがSQLでは、スキーマ全体を単一の単位として、プログラムから接続されたり切断されたりする。ホストのプログラムは、すべてのテーブルや他のスキーマオブジェクトを参照したり操作する権限を与えられていないこともある。しかし、そうした権限設定もコネクションの一部として行われる。

プログラムがファイルの中のフィールドを定義するのに対して、SQL はスキーマの中の列を定義する。FORTRAN は、ファイルからデータを取得するときに`FORMAT` 文と`READ` 文を使う。同様に、COBOLプログラムは`DATA DIVISION`をフィールド定義のために使い、`READ`でフェッチする。他の3GL言語も、構文とオプションが異なるだけで考え方は皆同じである。

ファイルシステムでは、同じデータを別々のプログラムで異なる名前で参照できる。もしファイルのレイアウトが変更されれば、そのファイルを使うプログラムはすべて書き直しになる。ファイルが空の場合、それは他の空のファイルとまったく同じである。空のファイルを読もうとすると、EOF（end of file）フラグが立ち、プログ

ラムは何らかの動作をする。一方、テーブルのカラム名とデータ型は、データベーススキーマの内部で定義される。それらは決して、フィールド名のようにプログラム内でしか使用できないわけではない。一定の制限内で、テーブルはホスト言語のことを知らずに変更することができる。

ホスト言語はただ、データベースから自身の持つ変数へ、正しく値を受け渡せるかどうかだけを心配すればよい。ところで、空集合がどういう扱いを受けるか、高校のときの数学を覚えているだろうか？ 空集合は正当な集合である。テーブルが空の場合であっても、なおテーブルは列と0 個の行を持つ。例外を発生させるEOFフラグは、テーブルにはない。なぜなら、テーブルに「最終行」はないからだ。

ファイルとテーブルのもう1つの大きな違いは、テーブルと列には制約を付けることができることだ。制約は、個々のトランザクションの終了後に、データベースか満たさなければならないルールだ。この意味で、データベースは、従来の受動的なファイルシステムより、オブジェクトの集合に似ている。

テーブルはスプレッドシートでもない。たしかに両者は、ディスプレイや紙の上で見るとよく似ている。スプレッドシートでは、カーソルを操作することで行、列、セル、セルの集合にアクセスできる。だがテーブルには、こうしたカーソル操作の概念はない。スプレッドシートにおけるセルは、データだけでなく見出しを保持することもできるし、スプレッドシートでは、行と列に本質的な違いは

ない。行列をそっくり入れ替えることもできるし、それでも妥当なデータが得られる。これはSQLのテーブルではあり得ないことだ。

テーブルは行から構成される。行はいかなる順序も持たず、キーによって識別される（つまり、1つ以上の列の値の組み合わせで一意になる）。行には、「最初」とか「最後」とか「次の」といった概念は存在しない。

行は列から構成される。列はいかなる順序も持たず、各行において一意な名前によって識別される。`SELECT *`や`INSERT INTO <table name>`では、データベースエンジンは、テーブル定義で決められたデフォルトの順序で列をリストするため、このケースは「列に順序がない」ことに対する例外に見えるかもしれない。しかし、これらはあくまで列名をずらずら書き並べる代わりの省略形にすぎない。

列はスカラ値から構成される。列はテーブル定義で宣言された固定的なデータ型を持たねばならない。またそれ以外にも、列には制約、デフォルト値といったものを定義して、データがビジネスルールを満たすことを保証させることができる。

5.1

CREATE TABLE 文

CREATE TABLE 文には、たくさんの情報が詰め込まれている。この文に制約と文を詰め込めば詰め込むほど、SQLは速く、簡単になるだろう。基本構文は次のとおりである。しかし、実際はたくさんのオプションが存在するので、それらについては後述する。

```
CREATE TABLE <table name> (<table element list>)
<table element list> ::=
<table element> | <table element>, <table element list>
<table element> ::=
<column definition> | <table constraint definition>
```

テーブル定義では、列定義によってデータを、制約によってそのデータを制御するルールを定義する。これはつまり、テーブルというのは、ファイルのような単純で受動的な存在よりも、（データとメソッドを持つという点で）オブジェクトに似ているということである。

5.1.1　列制約

新米のSQLプログラマは、しばしば自分たちに与えられた利点を十分に活用できない。そのせいで、エラーに見舞われたり、アプリケーション側で余計な仕事をしなければいけなくなる。列は、ファイルシステムにおけるフィールドのような単純で受動的な存在ではない。列は、それに紐付けられたデータ型以上のものである。以下にBNF記法の列定義を示す。

```
<column definition> ::=
<column name> <data type>
[<default clause>]
[<column constraint> ...]
[<constraint attributes>]
<column constraint> ::= NOT NULL
| <check constraint definition>
| <unique specification>
| <references specification>
[<constraint attributes>]
```

最初に注目すべき重要な点は、すべての列がデータ型を持たなければならないことだ。`ALTER` 文でテーブル定義を変更しない限りデータ型は不変だ。標準SQLは、多くのデータ型を用意している。なぜなら、SQLは多くの異なるホスト言語とともに働かなければならないからだ。

データ型は、大きく3つに分類できる。数値、文字、時間である。データ型とこれを操作するときのルールについては別章で詳しく見る。この主要な3つの型はわかりやすいので、詳細を知らなくても以下のサンプルを読む際に支障はない。

列制約は、列に付与されるルールである。行制約は同じ行の複数の列に付与される。テーブル制約は、複数の行に（普通は集約された形で）適用される。テーブルに含まれるすべての行は、これらの制約を満たさなければならない。ファイルシステムにはこういう仕組みはない。データの整合性チェックはアプリケーションプログラムで行われるからだ。制約は、SQL の機能の中で最も利用されていないものの1つだ。だから、これをマスターすれば本物の魔法使いのように見えるだろう。

制約には、名前といくつかの属性を与えることができる。制約名はデータベースエンジンが制約を変更したりエラーメッセージを表

示するときに使われる。

```
<constraint name definition> ::= CONSTRAINT <constraint name>
<constraint attributes> ::=
<constraint check time> [[NOT] DEFERRABLE]
  |[NOT] DEFERRABLE [<constraint check time>]
<constraint check time> ::= INITIALLY DEFERRED | INITIALLY IMMEDIATE
```

DEFERRABLEオプションを指定すると遅延制約となり、これによってトランザクションの実行中は制約を「オフ」にしておくことができる（DEFERRABLEを指定しただけで制約が遅延制約になるわけではない。遅延「可能」にするだけである）。また、初期状態を指定するINITIALLY DEFERREDとINITIALLY IMMEDIATEは、DML文が実行されたとき即座に制約を施行するか、トランザクションが終了するとき、つまりコミットの前に施行するかを選べる。これらのオプションについては、意味を持つ組み合わせは限られる。

- もしINITIALLY DEFERREDが指定されていれば、制約はDEFERRABLEでなければならない。
- もしINITIALLY IMMEDIATEが指定されているか暗黙に含意されていれば、DEFERRABLE もNOT DEFERRABLEも指定できない。この場合は、NOT DEFERRABLE が暗黙に前提とされる。

必要に応じて、トランザクション単位に制約を遅延させるか即座に施行するかを制御することもできる。次のような文を実行すればよい。

```
<set constraints mode statement> ::=
SET CONSTRAINTS <constraint name list> {DEFERRED | IMMEDIATE}
<constraint name list>
::= ALL | <constraint name> [{<comma> <constraint name>} ...]
```

この機能はSQL-92 のフルバージョンで取り入れられたが、大規模向けのデータベース製品を除くと、まだ実装されている製品は多くない【訳注1】。

実際、多くの製品では、すべての制約を`NOT DEFERRABLE INITIALLY IMMEDIATE`で使用している。

5.1.2 DEFAULT 句

`DEFAULT`句はあまり利用されていない機能だ。その構文はこうである。

```
<default clause> ::=
[CONSTRAINT <constraint name>] DEFAULT <default option>
<default option> ::= <literal> | <system value> | NULL
<system value> ::= CURRENT_DATE | CURRENT_TIME | CURRENT_TIMESTAMP |
  SYSTEM_USER | SESSION_USER | CURRENT_USER
```

SQL:2003 では、`CURRENT_PATH`と`<implicitly typed value specification>`も付け加えることができる。だがこれらはあまり使わないので、気にしなくてよい。

`INSERT` 文の実行時に、SQL エンジンが明示的な値を見つけられなかった場合、`DEFAULT` 句を探してそこで指定されている値を使用する。デフォルト句は、列に指定されているデータ型のリテラル値か、あるいは現在のタイムスタンプ、現在の日付、現在のユーザー識別子などを指定することができる。もしデフォルト句を指定しな

い場合で、列にNULLを許可している場合、データベースはNULLをデフォルトとして使う。もしこうした動作が失敗した場合、失われたデータについてのエラーメッセージが返ることになる。

アプリケーションでコードを書いて実行させる作業を、データベース側に行わせるのは良いことである。NULLを避けるためにしばしば使われるトリックとして、数値型の列で値が不明な場合のデフォルト値にゼロを使う、文字列型の場合には'{{unknown}}'とか'same address'のような明示的な文字列を使う、時刻であればトランザクションを識別するシステムのタイムスタンプを使う、といったものがある。なお、デフォルト句に設定できるのは定数や単純な関数のみである。生成列（第7章）などを指定することはできないので注意してほしい。

5.1.3 NOT NULL 制約

最も重要な列制約がNOT NULL制約である。これは、その列にNULLを使用することを禁じる。書く順番としては、DEFAULT句の後ろの位置なのだが、最近のたいていの実装では、この2つの句の順序にこだわらないことが多い。

NOT NULLは、原則として付けるべき制約だと考えてほしい。これを取り除いてよいのは、しかるべき理由がある場合だけである。NOT NULL制約は、NULLを許可したときのクエリの複雑さを回避するのに効果的な方法だ。裏を返すと、NULLが生成される場合には、DEFAULT値で代替するべきだ、ということである。

SQLにおけるNULLは、すべてのデータ型に属する特別なマーカーであり、こうした機能を持つ言語はSQL以外にない。NULLがどのように作用するかを理解すれば、SQLを理解したと言っても過言ではない。

NULLに関するアルファにしてオメガは、NULLは値ではないということだ。NULLは「ここには値はない」ということを示す目印にすぎない。ただ、物理ストレージ上で格納するには、何らかのデータ型に変換しなければならないだけだ。NULLにはさまざまな意味がある。たとえば、以下の4つが考えられる。

（1）失われた値
（2）未知の値
（3）雑多な値
（4）適用不能な値

上記以外の意味を持つ場合もあるが、当面、この4 パターンを考えてほしい。NULL がこの4 つのどの意味であるかは、その使われ方から判断する。この点を説明するために、私がイースターエッグを探していて、その色を知りたがっている、という例を考えてみよう。穴の中に卵が落ちてしまっている場合、卵は失われてしまっている（1）――あとで取り戻せればよいのだが。次に、卵がアルミフォイルでくるまれている場合、卵はあるのだが何色かはわからない（2）。さまざまな色が混合している卵の場合、その色は一意な値に決まりきらない雑多な値を持つ（3）。もし私が見つけたのが卵では

なくビリヤードの玉だった場合、その色は適用不能である（4）。それぞれの状況に対処する手段は、微妙に異なる。

NULLを算術計算の中で使用すると、NULL の伝播（propagation）という現象が起きる【訳注2】。つまり、式の中にNULLが含まれていると、結果までNULLになるのだ。またNULLを比較の中で使うと、真理値UNKNOWNが返されることもある。すると、SQLの奇妙な3 値論理の体系に迷い込むことになる。ORDER BY 句では、NULLS [FIRST|LAST] オプションを付けることで、NULL をソート順序において最大として扱うか最小として扱うかを制御できる。もし何の指定もない場合は、ベンダーは独自の順序を自由に決めている。また、NULLはいくつかの操作においては一緒にグループ化されるが、別の操作ではグループ化されない。一言で言えば、NULLを使うということは、SQLの多くの変則的なロジックと付き合わなければならない、ということなのだ。このNULL のもたらす諸問題については以降の章で論じる。ただ、1 つ言っておきたいのは、もし読者が新米のSQLプログラマであるならば、NULL の動作パターンについて十分な経験を持つまでは、NULLに関するルールをよく覚えておくことである。

5.1.4　CHECK 制約

CHECK 制約はテーブルの値が論理式を満たすかテストする。この論理式は、SQLで一般に検索条件と呼ばれる式だ。そして、その検索条件がFALSEを返すような行を拒否する。逆に、条件の戻り値がTRUE かUNKNOWN の場合は、行を受け付ける。これはUNKNOWN の行

を拒否するWHERE句や結合のON句における条件評価のルールとは異なる。この違いが生じる理由はCHECK 制約が「疑わしきは罰せず」の原則を採用しているからだが、これは、NULLを許可している列に対する条件を書きやすくするためである。

CHECK制約は(order_qty >= 0)のように簡単な形で書くことができる。こういうシンプルな検索条件を使うことは多いだろう。こうした式は、仮想的な手続き型言語における論理式の形をしている。簡単な式はプログラマにとっても馴染み深いので、新米プログラマでもSQL的なCHECK制約を十分に活用できる。

SQLはコードを簡潔にして読みやすくするための簡便な記法をいくつか持っている。たとえば、CHECK (rating BETWEEN 1 AND 10)と書けば、2つの不等式を使った比較を代用できる。同様に、CHECK (sex IN (0, 1, 2, 9))と書けば、ORで比較式をずらずら並べなくて済む。もちろん、こうした機能を利用するしないは自由だが、制約に名前を付けることはやっておいたほうがいい【訳注3】。制約に名前を付けないと、たいていのDBMSは非常に長く読みづらいランダムな文字列を制約名として自動生成する。これは、当該のスキーマにおいて制約を識別する名前が必要になるからだ。自分で読める名前を制約に与えておけば、制約を削除したり変更したいときに簡単に見つけられる。また、制約名は制約違反を知らせるエラーメッセージにも現れる。

たとえば、ある会社が「バーテンダーとして21 歳以下の従業員は雇うことができない」というルールを強制するCHECK 制約を作ろう。これは、応募者の誕生日と入社日を比較することで実現可能

だ。このとき、現在のシステム日付をCHECK 制約に使ってはいけない。それは常に変わり続けるからだ。

CHECK 制約の真の実力は、他の行や他のテーブル、あるいは他の制約との関係を検証するために複雑な式を記述する場合に発揮される。標準SQLが制定される以前、CHECK制約はそれが宣言されたテーブルの列しか参照することができなかった。だが標準SQLにおいて、CHECK 制約は任意のスキーマオブジェクトを参照できることとされている。ただし、この拡張はまだそれほど広く実装されていない。

CHECK 制約がいかに複雑なことを実現できるかを示すため、映画のデータベースを例に考えてみよう【訳注4】。まずは、いかなる国も10本以上の作品は輸出できないというルールを施行する。

```
CREATE TABLE ExportMovies
(movie_title CHAR(25) NOT NULL,
 country_code CHAR(2) NOT NULL, -- ISO-3166 のコード体系を使用
 sales_amt DECIMAL(12,2) NOT NULL,
   PRIMARY KEY (movie_title, country_code),
   CONSTRAINT National_Quota
     CHECK (-- 同じテーブル内の参照
             10 > ALL (SELECT COUNT(movie_title)
                         FROM Exported_Movies AS E1
                        GROUP BY E1.country_code)));
```

ExportMovies：輸出映画テーブル　　movie_title：作品名　　country_code：国コード
sales_amt：興行収入

では次に、いかなる作品も、たとえ国が違っていても、同じタイトルではいけないというルールを施行しよう。このルールを実装するには自己結合を使うが、その際は、基底テーブルの名前（ExportMovies）と相関名（E1）を結合の対象としなければならない。

```
CREATE TABLE ExportMovies
(movie_title CHAR(25) NOT NULL,
 country_code CHAR(2) NOT NULL,
 sales_amt DECIMAL(12,2) NOT NULL,
   PRIMARY KEY (movie_title, country_code),
   CONSTRAINT National_Quota
     CHECK (NOT EXISTS -- 自己結合
                (SELECT *
                   FROM ExportMovies AS E1
                  WHERE ExportMovies.movie_title = E1.movie_title
                    AND ExportMovies.country_code <> E1.country_code));
```

さらに、製作された国には輸出できないというルールを施行するには、次のようにする。

```
CREATE TABLE ExportMovies
(movie_title CHAR(25) NOT NULL,
 country_code CHAR(2) NOT NULL,
 sales_amt DECIMAL(12,2) NOT NULL,
   PRIMARY KEY (movie_title, country_code),
   CONSTRAINT Foreign_film
     CHECK (NOT EXISTS -- 別のテーブル「Movies」への参照
                (SELECT *
                   FROM Movies AS M1
                  WHERE M1.movie_title = ExportMovies.movie_title
                    AND M1.country_of_origin = ExportMovies.country_code)));
```

こうしたテーブル全体に対する制約では、NOT EXISTS述語を使うことが多い。このように、1つの制約の中で非常に多くの機能を持たせることは可能だが、何が悪くてエラーになったのかを明確に切り分けられるようにするには、シンプルな制約を複数記述するほうがよい。

制約名はスキーマ内で一意である必要がある。テーブル内で一意ではないので、注意してほしい。これには2 つの理由がある。第1に、もし2 つの異なるテーブルにおける2 つの異なる制約が同じ名前を持っていた場合、エラーメッセージにおいて、どちらの制約が破

られたのかわからない。第2に、スキーマレベルの宣言を行う文として、CREATE ASSERTION文があるからだ。これについては後述する。

5.1.5　UNIQUE 制約とPRIMARY KEY 制約

一意制約は、ある単一の列または複数列の組み合わせにおいて、重複値の存在を排除するためのルールである。

```
<unique specification> ::= UNIQUE | PRIMARY KEY
```

ファイルシステムに慣れたプログラマは、主キーをシーケンシャルファイルにおけるレコード番号と混同しがちだ。レコード番号は、ファイルにおいてレコードを特定するための手段である。これを目印にして、ディスクや磁気テープの読み込み／書き込みのヘッドを動かして行を探す。明白なことだが、シーケンシャルファイルは、1つの物理的順序しか持つことができない。初期のSQLは、その当時に既存のファイルシステム上で作られたので、主キーの構文はレコード番号を真似る形で設計された。後にコッドは、これが間違いであることに気づいた。テーブルにはいかなる順序もない。だが、そのときすでに主キーはSQLの一部になっていた。今日では、主キーは行の一意性を保証するために使われ、参照整合性制約では、被参照列の指定がなければデフォルトで主キーが使用される。

UNIQUE制約とPRIMARY KEY制約の間には、微妙な違いがある。1つのテーブルにはPRIMARY KEYは1 つしか存在してはならないが、UNIQUE 制約は複数付与できる。また、主キーの列は自動的にNOT

NULL制約が付与されるが、UNIQUE制約の列は（明示的にNOT NULL制約を付けない限り）NULLが存在してもよい。一意性はNULL以外の値についてチェックされる【訳注5】。もちろん、付けられる場合には常にUNIQUE制約の列にもNOT NULL制約を付けておくことが望ましい。そうすれば、その列をリレーショナルモデルにおける適切なキーとして扱うことができる。

一意制約は、複数列の組み合わせに対して付与することもできる。この場合の定義は、普通、列定義の最後にUNIQUE(列1, 列2, ...)のように書く。これによって、この列の組み合わせが一意であるという意味になる。たとえば、PRIMARY KEY(city, department)あるいはUNIQUE (city, department)という制約を宣言したとしよう。すると、たくさんの都市（city）にオフィスがあって、違うオフィスであっても同じ名前の部署（department）がありうることになる。しかし、都市と部署の組み合わせは一意でなければならないので、シカゴには同じ名前の部署は1 つしかない。たとえば、シカゴに「広報部」という名前の部署が2 つあってはならない、ということだ。このバリエーションを次のテーブルを使って考えよう。

```
CREATE TABLE Job_Assignments_1
(emp_name VARCHAR(10) NOT NULL,
 city_name  VARCHAR(15),
 dept_name CHAR(5));
```

Job_Assignments_1：仕事割当てその1テーブル　　emp_name：社員名
city_name：都市名　　dept_name：部署名

このCREATE TABLE 文は一応SQL でも通るのだが、Job_Assignments_1テーブルはまったくキーを持っていない。つまり、このテーブルはリレーショナルなテーブルではない。テーブル定義においてPRIMARY KEYを必須とすべきかどうかは、データベース標準化委員会でも議論があったのだが、非正規化データをデータベース内で扱えるよう配慮して、結局必須とはされなかった。

以下に、主キー制約と一意制約のパターンをいくつか示す。

```
CREATE TABLE Job_Assignments_2
(emp_name VARCHAR(10) NOT NULL PRIMARY KEY,
 city_name VARCHAR(15) UNIQUE,
 dept_name CHAR(5));

CREATE TABLE Job_Assignments_3
(emp_name VARCHAR(10) NOT NULL PRIMARY KEY,
 city_name VARCHAR(15),
 dept_name CHAR(5),
   UNIQUE (city_name, dept_name));

CREATE TABLE Job_Assignments_4
(emp_name VARCHAR(10) NOT NULL,
 city_name VARCHAR(15) NOT NULL,
 dept_name CHAR(5) NOT NULL,
  UNIQUE (emp_name, city_name, dept_name)); -- またはPRIMARY KEYでもよい
```

これらのテーブルに、表5.1に示すデータを上の行から順に挿入することを考える。

表5.1　主キー制約、一意制約、NOT NULL制約の施行例

emp_name	city_name	dept_name	説明

Jack	Chicago	Acct	1、2、3、4に妥当
Jack	Chicago	Acct	1に妥当、2、3、4に非妥当--主キー違反
Jill	Chicago	NULL	1に妥当、2に非妥当（Chicago）、3に妥当、4に非妥当
Hilary	NULL	Acct	1、2、3に妥当、4に非妥当
Walt	NULL	NULL	1、2、3に妥当、4に非妥当
Fred	NULL	NULL	1、2、3に妥当、4に非妥当

あとで、5.2 節においてオーバーラップする一意制約について見る。これを使うと、テーブルレベルでさらに複雑なルールを施行することができる。

5.1.6 REFERENCES 句

<references specification>は参照整合性制約を実装する最も単純な方法だが、ややトリッキーなところがある。一番単純なケースを考えてみよう。

```
<references specification> ::=
[CONSTRAINT <constraint name>]
REFERENCES <referenced table and columns>
[MATCH <match type>]
[<referential triggered action> ]
<match type> ::= FULL | PARTIAL
```

これは2つのテーブルを関連付けるという点で、これまで見てきた制約とは異なる。この制約を付与するテーブルは、参照する側であり、REFERENCES句に含まれるテーブルが、参照される側である。普通は、この2つは異なるテーブルだが、同じテーブルを使う自己参照もできないわけではない。自己参照の場合は、参照するテーブルの列の値が、同じテーブルの別の列にも存在していなければならない、という意味になる。

さらに、参照される側の列（複数列もありうる）には、一意制約が必要とされる。たとえば、注文テーブルは在庫テーブルに存在する商品についての注文しか登録できないというルールの場合、「商品」列は在庫テーブルにおける主キーか、あるいは一意制約の付与された列でなければならない【訳注6】。

オプションのMATCH FULL は、列の2 つのリストは、いずれも非NULL という意味である。MATCH PARTIALは、列の2つのリストは、NULLを許可し、かつNULLとNULLは等しいという扱いになる。この違いは、SQLではグルーピングと同等性が対立するとき、たびたび問題になる。もし参照先の列が明示されていない場合は、暗黙に参照先テーブルの主キー列が参照対象とされる。これが、主キーを宣言しておくべき理由の1つでもある。だが、安全のため常に参照先の列名は明示しておくべきだろう。また、複数のREFERENCES句が同じ列を参照してもかまわない。たとえば、飛行機の「操縦士」列と「副操縦士」列は、ともにパイロットテーブルの「パイロット」列を参照してもよい。

参照のタイプに、循環参照というものがある。ある列が、別のテーブルの列を参照する。その列もまた3 番目のテーブルの列を参照する。さらにその列が4 番目のテーブルの列を参照し……、ついに最初のテーブルの列が参照されて、円環が閉じる。まるで「未経験では仕事は見つからない。だが仕事を得ないと経験を積めない」という古いギャグのようだ。このような循環参照はデータの妥当性が保証されないため、良い設計とは言えない。

なお、外部キーの列は、主キーまたは一意制約の列と一致しなければならない。構文は以下のとおりである。

```
[CONSTRAINT <constraint name>]
FOREIGN KEY (<column list>)
REFERENCES <referenced table name> [(<reference column list>)]
```

REFERENCES 句は参照先のテーブルが変更された場合に、どのような動作を行うかを制御する2つの補助的な句を付け加えることができる。テーブル変更には、更新と削除があり、補助句もそれに対応している。

```
<referential triggered action> ::=
<update rule> [<delete rule>] | <delete rule> [<update rule>]
<update rule> ::= ON UPDATE <referential action>
<delete rule> ::= ON DELETE <referential action>
<referential action> ::= CASCADE | SET NULL | SET DEFAULT | NO ACTION
```

参照先テーブルが変更されると、SQLエンジンによってここで指定された動作が実行される。

1. CASCADEオプションは、参照先テーブルの新しい値に参照元テーブルの値も変更する。特定の信頼できる1つのテーブルを、識

別子のソースとして設定するのは、DDLの書き方としてよくやるものだ。そうすることで、システムは自動的に更新を伝播させられる。これによって、アプリケーションに何千行ものコードを書く必要がなくなり、クエリの最適化を改善し、更新から取り残されるデータを防止できる。

最近のDBMS は、スキーマを全体として扱う。いったん、参照先テーブルにおける列の値が現れ、参照元テーブルはそれをシステムが保持するポインタチェーンやハッシュテーブルによってスキーマ内に出現させる。これによって、更新のカスケードは速くなり、物理ストレージの消費も小さくなり、結合や集約も速くなる。

2. `SET NULL`オプションは、参照元テーブルの値を`NULL`にセットする。もちろん、これを使うには参照元の列が`NULL`を許可している必要がある。
3. `SET DEFAULT`オプションは、参照元テーブルの値を列のデフォルト値に変える。もちろん、これを使うには、参照元の列がデフォルト句を持っている必要がある。
4. `NO ACTION`オプションは名前のとおりである。参照元テーブルの値は何も変わらず、参照違反によるエラーメッセージが生じる。もし参照整合性制約が`ON UPDATE`も`ON DELETE` も明示していない場合、暗黙に`NO ACTION`が前提とされる。

5.2

入れ子のUNIQUE 制約

SQLにおける基本的なトリックの1つは、2つ（またはそれ以上の）テーブル間の1対1や多対多の関連を、それぞれのテーブルの主キーを含むテーブルによって表現することである。この第三のテーブルは、リレーショナルデータベース以前のデータベースにおける用語を使って「連結テーブル」とか「結合テーブル」と呼ばれることもある。しかし我々は、これが関連を表すテーブルであるがゆえに、正しくは「関連テーブル（junction table）」と呼ぶことを知っている。このタイプのテーブルには、関連を正しく保証するための制約が必要になる。例に次の2つのテーブルを考えよう。

```
CREATE TABLE Boys
(boy_name VARCHAR(30) NOT NULL PRIMARY KEY,
 ...);
```

Boys：少年テーブル　　boy_name：少年の名前

```
CREATE TABLE Girls
(girl_name VARCHAR(30) NOT NULL PRIMARY KEY,
 ...);
```

Girls：少女テーブル　　girl_name：少女の名前

もちろん、名前をキーに使うのはバッドノウハウであることは承知している。しかしここでは、サンプルの読みやすさを優先している。この2つのテーブルの間に成り立つ関連は、たくさんのものが考えられる。もし信じられないと言うなら、テレビ番組『ジェリー・

シュプリンガー・ショー』をしばらく鑑賞してみることだ【訳注7】。最も単純な関連テーブルは次のようになる。

```
CREATE TABLE Couples
(boy_name VARCHAR(30) NOT NULL
   REFERENCES Boys (boy_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 girl_name VARCHAR(30) NOT NULL
   REFERENCES Girls(girl_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE);
```

Couples：カップルテーブル　boy_name：少年の名前　girl_name：少女の名前

Couplesテーブルには、たとえば次のような行を挿入できる。

```
('Joe Celko', 'Hilary Duff')
('Joe Celko', 'Lindsay Lohan')
('Tobey Maguire', 'Lindsay Lohan')
('Joe Celko', 'Hilary Duff')
```

おっと、私が'Hilary Duff'と付き合っているという行が二度も現れてしまった！　これが起きるのは、Couplesテーブルがキーを持っていないからだ。これはイージーミスだが、修正するのは見かけほど簡単ではない。

```
CREATE TABLE Orgy
(boy_name VARCHAR(30) NOT NULL
   REFERENCES Boys (boy_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 girl_name VARCHAR(30) NOT NULL
   REFERENCES Girls(girl_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
   PRIMARY KEY (boy_name, girl_name)); --複合キー
```

Orgy（「乱交」）テーブルは、重複行を排除することで、適正なテーブルになった。主キーは複数のキーから構成されている。こういうキーを複合キーと呼ぶ。これで、次のように正しく行をINSERTできる。

```
('Joe Celko', 'Hilary Duff')
('Joe Celko', 'Lindsay Lohan')
('Tobey Maguire', 'Lindsay Lohan')
```

だが、このテーブルにおける唯一のルールは、カップルとして一度しか現れてはならない、というものなので、すべての男の子がすべての女の子とカップリングできてしまう。このような性の乱れは、良識ある大人にとっては深刻な事態である。そこで私が提案したいルールは次のようなものだ。

> 男の子は何人でも好きなだけ女の子と付き合えるが、女の子は1人だけの男の子に貞節を捧げなければならない。

これを実現するには、NOT NULL UNIQUE 制約を「girl_name」列に付与すればよい。これによってこの列はキーとなる。1列しかないので、実にシンプルなキーであるが、これは複合的な主キーの部分集合であるため、入れ子のキーにもなっている。

```
CREATE TABLE Playboys
(boy_name VARCHAR(30) NOT NULL
   REFERENCES Boys (boy_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 girl_name VARCHAR(30) NOT NULL UNIQUE --入れ子のキー
   REFERENCES Girls(girl_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
   PRIMARY KEY (boy_name, girl_name)); --複合キー
```

Playboysテーブルは重複行を持たないのでやはり適正なテーブルだ。だが、このルールの元では、次のように私は複数の女性と遊ぶことができる。

```
('Joe Celko', 'Hilary Duff')
('Joe Celko', 'Lindsay Lohan')
```

女性の側としては、むしろこの逆を主張するであろう。つまり、女性が複数の男性と遊べるようなテーブルだ。

```
CREATE TABLE Playgirls
(boy_name VARCHAR(30) NOT NULL UNIQUE --入れ子のキー
   REFERENCES Boys (boy_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 girl_name VARCHAR(30) NOT NULL
   REFERENCES Girls(girl_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
   PRIMARY KEY (boy_name, girl_name)); --複合キー
```

Playgirlsテーブルは、もともとのテーブルのデータのうち、次のような行を許可する。

```
('Joe Celko', 'Lindsay Lohan')
('Tobey Maguire', 'Lindsay Lohan')
```

さて、検討してきたキーのパターンを少し考えてみよう。実は複合的な主キーは冗長だ。このようなキーのことをRDBMSの用語で「スーパーキー」と呼ぶ。もし男性が1人につき1行しか現れず、かつ、女性も1人につき1行しか現れないのだとしたら、（boy_name,girl_name）のペアも1 行しか現れないのは自明である。いま、この条件はboy_name 列とgirl_name 列の一意制約で満たされている。とはいえ、このような冗長なキーが役に立つこともある。テーブルを検索するとき、（boy_name,girl_name）のペアを条件にすることは珍しくない。そういう場合、この2 列をカバーするインデックスはパフォーマンス改善に有効だ。もし運良くこの2 列だけしかSELECT 文で使わないなら、クエリエンジンはインデックスだけにアクセスすればよく、基底テーブルには触らずに済むのでI/O量が削減される【訳注8】。

さて、良識ある大人はこのようなハリウッドスキャンダルに眉をひそめて、1 人を伴侶とした落ち着いた生活を作ってほしいと思うだろう。その望みを叶えるテーブルは次のようなものだ。

```
CREATE TABLE Marriages
(boy_name VARCHAR(30) NOT NULL UNIQUE --入れ子のキー
   REFERENCES Boys (boy_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
 girl_name VARCHAR(30) NOT NULL UNIQUE --入れ子のキー
   REFERENCES Girls(girl_name)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
   PRIMARY KEY(boy_name, girl_name)); --冗長な複合キー
```

このMarriagesテーブルによって、もともとのCouplesテーブルのうち、許される行は以下の2つだけになる。

```
('Joe Celko', 'Hilary Duff')
('Tobey Maguire', 'Lindsay Lohan')
```

これで道徳を重んじる方々も満足だろう。同性婚を認めるテーブルの作り方は、読者の練習問題としよう。

多くの製品では、主キーはデータモデルにおいて特別な意味を持つと想定しており、このキーを使ったテーブルアクセスが最も頻繁に行われると想定している。REFERENCES句がデフォルトでは参照先テーブルの主キーを参照することを考えても、主キーについて特別な仕組みを設けることは、悪いアイデアではない。

新米のSQLプログラマは、外部キー制約が、同じテーブルまたは異なるテーブルの任意の一意制約を参照できることを意識していない。次のコードは参照だらけだが、この句が持つ可能性について理解するには良いサンプルだ。なお、複数列を参照する場合の構文も基本的にこれと同じだ。

```
CREATE TABLE Foo
(foo_key INTEGER NOT NULL PRIMARY KEY,
   ...
 self_ref INTEGER NOT NULL
   REFERENCES Foo(foo_key),
 outside_ref_1 INTEGER NOT NULL
   REFERENCES Bar(bar_key),
 outside_ref_2 INTEGER NOT NULL
   REFERENCES Bar(other_key),
   ...);

CREATE TABLE Bar
(bar_key INTEGER NOT NULL PRIMARY KEY,
 other_key INTEGER NOT NULL UNIQUE,
   ...);
```

5.2.1　オーバーラップするキー

さて、入れ子のキーに話を戻そう。これを使うと、どの程度のことができるようになるだろう？ 私のお気に入りのサンプルは、先生のスケジュールを管理するテーブルだ（簡単にするため、参照整合性制約とCHECK制約は除外した）。

```
CREATE TABLE Class_Schedule
(teacher_name VARCHAR(15) NOT NULL,
 class_name CHAR(15) NOT NULL,
 room_nbr INTEGER NOT NULL,
 class_period INTEGER NOT NULL,
   PRIMARY KEY (teacher_name, class_name, room_nbr, class_period));
```

Class_Schedule：クラススケジュールテーブル　teacher_name：教師名
class_name：クラス名　room_nbr：教師番号　class_period：クラス期間

上記のように、主キーとしてすべての列を選択するのは、最も安直な方法である。このテーブルに登録される行の典型は、次のようなものだ。

```
('Mr. Celko', 'Database 101', 222, 6)
```

さて、このテーブルに施行したいルールは以下の4つである。

ルール1　1人の先生が1つの学期で使用できる教室は1つだけである。

ルール2　1人の先生が1つの学期で受け持つクラスは1つだけである。

ルール3　1つの教室は1つの学期で1つのクラスにしか使われない。

ルール4　1つの教室は1つの学期で1人の先生しか使わない。

よく見ると、ルール1とルール4 は同じことを言っている。ただ表現のしかたが違うだけだ。このように、一意性の条件を検討するときはよく注意しなければならない。ここでは、それぞれ別々のルールとして両方を残しておく。さて、読者の頭の中に浮かぶ回答はどのようなものだろうか。おそらく、次のような一意制約をずらっと並べる方法ではないだろうか。

```
CREATE TABLE Class_Schedule_1 --案1:間違い!
(teacher_name VARCHAR(15) NOT NULL,
 class_name CHAR(15) NOT NULL,
 room_nbr INTEGER NOT NULL,
 class_period INTEGER NOT NULL,
   UNIQUE (teacher_name, room_nbr, class_period),      --ルール1
   UNIQUE (teacher_name, class_name, class_period),    --ルール2
   UNIQUE (class_name, room_nbr, class_period),        --ルール3
   UNIQUE (teacher_name, room_nbr, class_period),      --ルール4
   PRIMARY KEY (teacher_name, class_name, room_nbr, class_period));
```

列の順序はインデックスを作る際にはパフォーマンス上の考慮が必要だが、いまの目的においては無視してよい（インデックスのチューニングについては第39章で取り上げる）。

もしこの4つの制約をすべて付与するなら、それで主キー制約も実質的にカバーされるので、PRIMARY KEYは削除してかまわない。しかし、PRIMARY KEYと制約のうちの1 つを削除すると何が起きるだろう？ たとえばこうだ。

```
CREATE TABLE Class_Schedule_2 -- 案2:間違い
(teacher_name VARCHAR(15) NOT NULL,
 class_name CHAR(15) NOT NULL,
 room_nbr INTEGER NOT NULL,
 class_period INTEGER NOT NULL,
   UNIQUE (teacher_name, room_nbr, class_period),   --ルール1
   UNIQUE (teacher_name, class_name, class_period), --ルール2
   UNIQUE (class_name, room_nbr, class_period));    --ルール3
```

このテーブルには、次の2つのレコードを登録することができる。

```
('Mr. Celko', 'Database 101', 222, 6)
('Mr. Celko', 'Database 102', 223, 6)
```

このようなデータが登録されてしまうと、私の第6学期はたいへん忙しいものになる。同時に2つの異なる教室にいなければならないからだ。このスケジュールに別の教師が加わると、さらにややこしくなる。

```
('Mr. Celko', 'Database 101', 222, 6)
('Mr. Celko', 'Database 102', 223, 6)
('Ms. Shields', 'Database 101', 223, 6)
```

シールズ（Shields）氏と私は、どちらも223教室で、同時に別の教科を教えることになる。マシュー・バールは、この制約を分析して、次のようなルールを思いついた。

```
CREATE TABLE Class_Schedule_3 --案3:正しい
(teacher_name VARCHAR(15) NOT NULL,
 class_name CHAR(15) NOT NULL,
 room_nbr INTEGER NOT NULL,
 class_period INTEGER NOT NULL,
   UNIQUE (teacher_name, class_period), --ルール1 および ルール2
   UNIQUE (room_nbr, class_period));    --ルール3 および ルール4
```

もし教師が一学期に1つの教室にしかいないならば、学期と教師がわかれば、教室も1つに決まる。つまり、教室は教師と学期の組み合わせに関数従属しているのだ。同様に、もし教師が一学期に1 つだけのクラスしか教えないならば、クラスは教師と学期の組み合わせに関数従属する。残りの2つのルールについても同様だ。クラスは教室

と学期の組み合わせに関数従属し、教師は教室と学期の組み合わせに関数従属する。

案1 の制約では、このような従属性のルールが施行されない。たとえば、次のような行をINSERTできてしまう。

```
('Mr. Celko', 'Database 101', 222, 6)
('Mr. Celko', 'Database 102', 223, 6)
```

この行は、ルール1とルール2 に違反する。Class_Schedule_2 テーブルで与えられていた一意制約でも、この違反を検知することができず、行の挿入を許してしまう。

UNIQUE (teacher_name, room_nbr, class_period)という制約は、教師、教室、学期の完全な組み合わせをチェックするため、('Mr. Celko', 222, 6)が('Mr. Celko', 223, 6)は異なるとみなし（教室が違っている）、この2行を挿入しても何の問題もない。実際には、この2 行が意味することは、セルコ氏（Mr. Celko）が同時に2 つの異なる教室にいることを意味するのだが。

UNIQUE (teacher_name, class_name, class_period) は、これが表現すべきルールを正しく表していないため、('Mr. Celko','Database 101', 6)が('Mr. Celko', 'Database 102', 6)と異なること、およびセルコ氏が一学期に2 つの教室で教えられてしまうことから、ルール2を破っている。この状態だと、

```
('Ms. Shields', 'Database 103', 222, 6)
```

という行も挿入することができてしまい、そうするとルール3とルール4を破ることになる。

5.2.2　一列の一意性と複数列の一意性

2007年8月、ライオネル・クラークは、シンプルなパズルを考えついた。このパズルは一意制約がどのように機能するかを示す良い例題だ。問題はこうだ。移行元のテーブル（Source）から移行先のテーブル（Destination）へデータを移したい。ただし1 つ条件があって、1 回のINSERT文しか使ってはならない。

```
CREATE TABLE Source
(a INTEGER NOT NULL,
 b INTEGER NOT NULL,
   PRIMARY KEY (a, b));

INSERT INTO Source (a, b) VALUES (1, 1), (1, 2),          -- a=1 グループ
                                 (2, 3), (2, 4),          -- a=2 グループ
                                 (5, 5), (5, 1), (5, 3), -- a=5 グループ
                                 (7, 2),                  -- a=7 グループ
                                 (9, 0),                  -- a=9 グループ
                                 (11, 2);                 -- a=11 グループ

CREATE TABLE Destination
(a INTEGER NOT NULL UNIQUE,
 b INTEGER NOT NULL UNIQUE,
   FOREIGN KEY (a, b)
   REFERENCES Source(a, b));
```

Source：移行元テーブル　　Destination：移行先テーブル

ポイントは、Destinationテーブルに適合する部分集合は複数考えられることだ。もし(1, 1)を使えば、列「a」に対する一意制約によって、もう(1, 2)を移行することはできない。これはデータを列

「a」から見た場合の話で、同様に、(5, 1)を使えば、列「b」に対する一意制約によって、もう(1, 1)は使えない。

　列「b」を基準にINSERT文のリストを並べると、次のようになる。

```
INSERT INTO Source (b, a) -- 列を入れ替えた
            VALUES (0, 9),                   -- b=0 グループ
                   (1, 1), (1, 5),           -- b=1 グループ
                   (2, 1), (2, 7), (2, 11), -- b=2 グループ
                   (3, 2), (3, 5),           -- b=3 グループ
                   (4, 2),                   -- b=4 グループ
                   (5, 5);                   -- b=5 グループ
```

　ちょっと考えれば、部分集合を最大にするためには、bの値を基準にした各グループから、最低1 つのペアを選択することが必要条件であることがわかる。1 つのグループから複数のペア、たとえば(1, 1)と(1, 5)を選んでしまうと、その時点で列「b」の一意制約に違反してアウトである。ただし、これだけではまだ、その部分集合の中に列「a」の一意制約を満たさない行が存在する可能性がある。たとえば、(1, 1)を選んだあとに(2, 1)を選ぶことはできない。もし読者が学生時代に数学専攻だったなら、これは選択公理の有限集合に対する適用であることに気づくだろう【訳注9】。

　まず考えられる解は次のようなものだ。

```
INSERT INTO Destination
WITH X (a, b, pair_nbr)
AS
(SELECT a, b,
        ROW_NUMBER() OVER(ORDER BY a ASC, b DESC)
   FROM Source)
SELECT a, b
  FROM X AS P1
 WHERE NOT EXISTS
            (SELECT *
               FROM X AS P2
              WHERE P2.pair_nbr < P1.pair_nbr
                AND (P1.a = P2.a OR P1.b = P2.b));
```

・・・

このSQLは、1つの正しい集合を返す。考え方としては、ROW_NUMBER関数で行をソートして連番を振り、テーブルP1のカレント行より前のP2テーブルのすべての行をスキャンして、aやbの値が重複する行がないかを調べて、そのような行を除外するというものだ【訳注10】。それゆえ、カレント行は自動的に答えから排除される。結果は、表5.2のようになる。

表5.2： 一意制約のパズル：正しい答え バージョン1

a	b
1	2
2	4
5	5
9	0

マイク・グッドは、これは最善解ではないと指摘してきた。バージョン1の結果には4 行しか含まれないが、実際は表5.3に示すよう

に、5行選び出すことも可能だからだ。バージョン1は1 行無駄にしてしまっている。

表5.3： 一意制約のパズル：正しい答え バージョン2

a	b
1	1
2	4
5	3
7	2
9	0

　母集合を変えてみると、バージョン1のさらなる欠点が明らかになる。私のアルゴリズムでは、特定の母集合においては、4 行返すべきところ、2 行しか返さない。たとえば、次のような母集合を考えよう。

```
INSERT INTO Source (a, b) VALUES (1, 1), (1, 2),
                                 (2, 1), (2, 2), (2, 3), (2, 4),
                                 (3, 3), (3, 4),
                                 (4, 1);
```

　このSourceテーブルに対して、妥当な解の1つは次のようになる。

a	b
1	2

2	3
3	4
4	1

だが、バージョン1では、次のようになってしまう。

a	b
1	2
2	4

また本来、どのようなデータを含むSource テーブルが与えられようと、Source.a とSource.bを入れ替えたとしても同じだけの行数をDestinationテーブルにINSERTできるべきだが、最初のデータについて、Sourceテーブルのaとbを入れ替えると、次のように3行しか返さなくなってしまう。

a	b
0	9
1	5
2	11

このような非対称性は、論理的に何か間違っている証拠である。OVER (ORDER BY ...)句のソート順序を変えることで、異なる答えを得ることも可能だが、そのどちらかが最適解である保証はない。

バージョン1が最善解ではないというマイクの指摘は正しい。この解の目的は、最初のとっかかりとして、受け入れられる方法を見つけることだった。一般に、NP完全問題をSQLで解こうとすると、考えうる答えのすべての組み合わせを作ることになり、クエリの実行には非常に時間がかかる【訳注11】。そのような、正しいが、しかし「激遅」なクエリの一例をお見せしよう。

```
--オリジナルのソースデータを作る
CREATE TABLE Source2
(pair_nbr INTEGER NOT NULL UNIQUE,
 a INTEGER NOT NULL,
 b INTEGER NOT NULL,
  PRIMARY KEY (a, b));

INSERT INTO Source2
VALUES (1, 1, 1), (2, 1, 2), (3, 2, 3), (4, 7, 2), (5, 2, 4),
       (6, 5, 5), (7, 5, 1), (8, 5, 3), (9, 9, 0), (10, 11, 2);

--10個のサンプルペアからすべての部分集合を作る共通表式
WITH Flags (I)
AS (SELECT I FROM (VALUES ('t'), ('f')) AS X(I)),

--すべての部分集合とランダムなsubset_nbrとの最低なクロス結合
Subsets (subset_nbr, f01, f02, f03, f04, f05, f06, f07, f08, f09, f10)
AS
(SELECT ROW_NUMBER() OVER () AS subset_nbr,
         F01.I, F02.I, F03.I, F04.I, F05.I,
         F06.I, F07.I, F08.I, F09.I, F10.I
   FROM Flags AS F01, Flags AS F02, Flags AS F03,
        Flags AS F04, Flags AS F05, Flags AS F06,
        Flags AS F07, Flags AS F08, Flags AS F09, Flags AS F10),

--順列からペアを除外する
Candidates(subset_nbr, a, b)
AS
(SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 1 AND P.f01 = 't'
 UNION ALL
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 2 AND P.f02 = 't'
 UNION ALL
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr =3 AND P.f03 = 't'
 UNION ALL
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 4 AND P.f04= 't'
 UNION ALL
```

```
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 5 AND P.f05 = 't'
 UNION ALL
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 6 AND P.f06 = 't'
 UNION ALL
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 7 AND P.f07 = 't'
 UNION ALL
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 8 AND P.f08 = 't'
 UNION ALL
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 9 AND P.f09 = 't'
 UNION ALL
 SELECT subset_nbr, S.a, S.b
   FROM Subsets AS P, Source2 AS S
  WHERE S.pair_nbr = 10 AND P.f10 = 't')
SELECT subset_nbr, a, b
  FROM Candidates AS C1
 WHERE subset_nbr
           IN (SELECT C2.subset_nbr -- 一意性制約を満たすすべてのペアを見つける
                 FROM Candidates AS C2
                GROUP BY C2.subset_nbr
               HAVING COUNT(a) = COUNT(DISTINCT a)
                  AND COUNT(b) = COUNT(DISTINCT b)
                  AND COUNT(*) >= 5) --ペアが5つ以上のもの
ORDER BY subset_nbr, a, b;
```

結果

```
 subset_nbr   a    b
====================
        110    1   1
        110    2   4
        110    5   3
        110    9   0
        110   11   2
        206    1   1
        206    2   4
        206    5   5
        206    9   0
        206   11   2
        218    1   1
        218    2   3
        218    5   5
        218    9   0
        218   11   2
        614    1   1
        614    2   4
        614    5   3
        614    7   2
        614    9   0
        710    1   1
        710    2   4
        710    5   5
        710    7   2
        710    9   0
        722    1   1
        722    2   3
        722    5   5
        722    7   2
        722    9   0
```

この解の問題点として、以下の2つが気になるところだ。

1. `subset_nbrs`が異なるだけで、重複する答えが含まれることがある。重複を除去するために関係除算を行うこともできるが、ますます読みにくくなるだろう（関係除算については第27章参照）。
2. スケーラビリティも気になるだろうが、思ったよりも悪くないはずである。リチャード・レムレーは、数独パズルの解法を見つけるために、すべての可能なマス目の組み合わせを計算する

ために81回の自己結合を行うクエリを書いたが、解が数百個になっても、非常に高速だった。それに比べれば、このクエリはずっと単純だ。

これでようやく、いかなる母集合に対しても、Sourceテーブルの「a」列と「b」列を入れ替えたとしても、同じ数だけのDestinationテーブルの行を得ることができるようになった。

5.3

CREATE ASSERTION 文

標準SQL では、CREATE ASSERTION 文によって同じスキーマ内のテーブル間の表明（ASSERTION）を作ることができる。表明は、CHECK 制約などと違って、特定のテーブルに付与されるものではない。構文は次のとおりである。

```
<assertion definition> ::=
CREATE ASSERTION <constraint name> <assertion check>
[<constraint attributes>]
<assertion check> ::= CHECK (<search condition>)
```

ご想像のとおり、DROP ASSERTION文もある。だがALTER文はないので注意してほしい。表明は、特定のテーブルに付与されるCHECK制約ではできなかったことができるようになる。なぜなら、表明はテーブルの外側に存在するからだ。

たとえば、ある会社の社員の総数は、複数の保険プランを含むテーブルの社員数の合計と一致しなければならない、というルールをCHECK 制約で実装することは原理的に難しい。だが、表明を使えば簡単だ。

```
CREATE ASSERTION Total_health_Coverage
CHECK (SELECT COUNT(*) FROM Personnel) =
    + (SELECT COUNT(*) FROM HealthPlan_1)
    + (SELECT COUNT(*) FROM HealthPlan_2)
    + (SELECT COUNT(*) FROM HealthPlan_3);
```

Total_health_Coverage：保険加入者総数　　Personnel：社員テーブル
HealthPlan_1：保険プラン1テーブル　　HealthPlan_2：保険プラン2テーブル
HealthPlan_3：保険プラン3テーブル

5.4

一時テーブル

標準SQLでは、一時テーブルは基底テーブルとほぼ同じ機能を持つ。その構造はスキーマの中で永続的だが、自動的に行が削除されるので、ユーザーはいつ削除するか意識する必要がない。一時テーブルには、複数のユーザーでデータを共有するグローバル一時テーブルと、1人のユーザーのみがデータを利用可能なローカル一時テーブルがある。どちらのテーブルにも、基底テーブルと同じようにユーザー権限を設定できる。

5.4.1　一時テーブルの宣言

一時テーブルの利用方法として、ストアドプロシージャとともに使って、中間的な結果セットを保持し、何度もクエリを実行したり再計算する手間を省く、というものがある。一時テーブルを作る構文は以下のとおりである。

```
CREATE [GLOBAL | LOCAL] TEMP[ORARY] TABLE <table name>
(<table element list>)
ON COMMIT [PRESERVE | DELETE] ROWS;
```

<table element>は、基底テーブルと同様、列、制約、参照整合性制約の定義である。ここまでは通常のCREATE TABLE文と同じだが、違うのはGLOBALとLOCALという2つのオプションが付け加えられていることだ。

GLOBALオプションは、テーブルの1 つのコピーが、一時テーブルを利用するすべてのアプリケーションプログラムのモジュールから

利用可能なことを意味する。グローバル一時テーブルは、一般にセッション間でデータを引き継ぐために使われる。

LOCALオプションは、テーブルの1つのコピーが、一時テーブルを利用する1つのアプリケーションプログラムのモジュールでのみ利用可能なことを意味する。ローカル一時テーブルは、一般に単一モジュール内における「使い捨てテーブル」として使われる。もし複数のユーザーが同一のローカル一時テーブルにアクセスした場合、それぞれがテーブルのコピーを取得する（初期状態では空である）が、そのコピーの有効範囲は、セッションまたはモジュールのスコープ内である。

一時テーブルについて、同じ名前のテーブルが同一スキーマ内に複数存在することに違和感を持つ人もいるかもしれない（実際、これはスキーマオブジェクトの一意性というSQL の基本的なルールに反しているように見える）。だが、作られたテーブルはあくまで1つであり、それにユーザーを識別する見えない列を裏で持っていると考えればわかりやすいだろう。ユーザーから見えているのは、いわばローカル一時テーブルをもとに作られた更新可能なビューであり、この見えない列の値がユーザー識別子と一致する行だけが、それぞれのユーザーには見えている。新しい行がローカル一時テーブルに追加されるときは、カレントユーザーの名前がデフォルトでこの見えない列に設定される。

SQLにおけるモジュールの概念については、ジム・メルトン『Understanding SQL's Stored Procedures』（Morgan Kaufmann、1998）に詳しい。しかし、ここでは、プログラム、プロシージャ、

関数、サブルーチン、あるいは読者の使う手続き型言語で書いたひとかたまりのコード、ぐらいの意味で理解してもらえればかまわない。

一時テーブルはスキーマに存在するテーブルなので、DROP TABLE文で削除することができるし、通常のINSERT、DELETE、UPDATE の各文で変更できる。通常のテーブルと違うのは、セッションまたはモジュールの終了時における動作である。ON COMMIT [PRESERVE | DELETE] ROWS 句は、COMMIT 文が成功したときの動作を定義する。PRESERVE オプションを指定すると、次にテーブルが使われるときにもデータを保存しておき、セッションの終了時のみにデータを削除する。DELETEオプションを指定すると、セッションが継続中であっても、COMMITが実行されるたびに行が削除される。どちらの場合でも、セッションまたはモジュールの終了時にテーブルがクリアされるのは共通である。

一時テーブルを利用することで、ときに重大な問題をもたらすことがあるので注意が必要だ。たとえば、プログラマがこっそり一時テーブルを作ることができると、その設計はすぐにシーケンシャルなファイルシステムを扱うときのやり方になってしまう。彼らは、一時「テーブル」ではなく一時作業用の「ファイル」を扱っているような気になってしまう。また、一時テーブルは実体のあるテーブルなので、物理ストレージ領域を消費する。もし何百というユーザーが同じプロシージャを呼び出せば、同じデータの何百というコピーのためのテーブルが割り当てられることになり、深刻なパフォーマンス低下を招くことになる。特に、一時テーブルと特定の物理領

域に作成するような実装においては、こうしたアクセス集中による I/O負荷は無視できない。

5.5

テーブルを操作する

DDLにおける3つの基本的なテーブル操作の文は、

```
CREATE TABLE
DROP TABLE
ALTER TABLE
```

である。これらの文の役割は、名前どおりなのでとてもわかりやすい。一言で言えば、`CREATE TABLE`でテーブルを作り、`DROP TABLE`で削除し、`ALTER TABLE`で構造を変更する。テーブルを作り、命名する際には、守るべきシンプルなルールがある。

ルール1

テーブル名はスキーマ内で一意でなければならない。列名はテーブル内で一意でなければならない。SQLではテーブル名と列名に同じ名前を付けることが許されているが、混乱するのであまりやらないほうがよい。ルール4と6も参照。

ルール2

SQLにおける名前に使ってよいのは、アルファベット、アンダースコア（_）、数字である。ベンダーによってはほかにも特殊文字を許していることもあるが、避けるのが賢明だ。そうした文字は移植できないリスクがあるし、製品が変わるとソート順序が変わる原因にもなる。

ルール3

標準SQLでは、ダブルクォーテーションで囲めば、スペースや予約語を名前に使うことが許されている。しかしこれも可能な限

り避けるべきだ。

ルール4

テーブルに集合名詞や複数形を使うことで、テーブルが集合であることをイメージしやすくなる。たとえば、本当に社員が1 人しかいない会社でない限り、「Employee」というテーブル名を使うべきではない。代わりに「Employees」や「Personnel」という名前にするべきだ。

ルール5

スキーマ内の同じ属性については、どのテーブルでも同じ名前を使うべきだ。たとえば、1 つのテーブルで「sex」という名前を作ったら、それと同じ意味の列を別のテーブルで「gender」と名付けてはならない。こうしたルールを開発者に守らせるための辞書を用意しておくべきだろう。

ルール6

列やスキーマオブジェクトには、単一の属性名を使うこと。この点については、拙著『SQL programming style』で詳しく論じているので、あまり深入りはしない。テーブルは最低1 つの列を持たなければ作れない。列を定義するときの並び順は、（別に強制ではないのだが）慣習的な順序で並べるのがよいだろう。というのも、デフォルトでは、列はテーブル定義で並べた順序で画面に表示されるからだ。たとえば住所であれば、名前、番地、市町村、州、郵便番号、というのが社会一般で通用している順序であるから、テーブル定義においてもこの順番で並べる

のがいいだろう【訳注12】。そうすれば、画面上に表示したときもこの順番で並ぶので読みやすい。

本書では、予約語は大文字、テーブル名は先頭文字を大文字、列名は小文字で表記するルールを採用している。また、テーブルの相関名で数字の前に来る文字についても、大文字を使う（たとえばPersonnel テーブルの相関名はP0、P1、… 、Pn）である。この場合の数字は、SQL文の中でテーブルが出現した順番を意味する。

5.5.1 DROP TABLE 文

DROP TABLE 文はテーブルをデータベースから削除する。これはテーブルを空にするのとは違う。スキーマオブジェクトが削除されるとは、それが完全になくなるということだ。構文は次のとおりである。

```
<drop table statement> ::= DROP TABLE <table name> [<drop behavior>]
<drop behavior> ::= RESTRICT | CASCADE
```

<drop behavior> 句は2 つのオプションを持っている。もしRESTRICT が指定されている場合、削除対象のテーブルは他のいかなるビューや制約においても参照されていてはならない。このオプションの目的は、知らないうちにテーブルが削除されて、それを使っているビューなどを使ったときに思わぬエラーを引き起こさないための予防措置である。一方、CASCADE が指定されている場合は、そのような参照元のオブジェクトもテーブルとあわせて削除される。

デフォルトでは、ほとんどのDBMSで実質的にRESTRICTモードで動いており、テーブルが他のオブジェクトで参照されている場合は、エラーメッセージを表示する。そうでないと、削除されたテーブルに依存するオブジェクトを利用しようとしたときに、テーブルがないことに気づきあわてることになる。

DROPという予約語や<drop behavior>句は、テーブル以外のスキーマオブジェクトを削除する場合にも使われる。DROP VIEW、DROP SCHEMA、DROP CONSTRAINT 等々。普通これらはDBAだけが使用可能な文であり、当然のことだが開発者が勝手に使うことは許されない。

5.5.2 ALTER TABLE 文

ALTER TABLE 文は、列や制約をテーブルに付け加えたり、削除したり、変更したりする。この文もいまでは標準SQLに存在しているが、大半のDBMSが、標準化される前からサポートしていた。標準化が遅かったせいで、いまでも製品間で統一性がない。詳しい構文については製品のマニュアルを見てほしい。これもまた、DBAだけが使えるコマンドであり、開発者が許可なく使ってはならない。標準SQLの構文は次のとおりである。

```
ALTER TABLE <table name> <alter table action>
<alter table action> ::=
| DROP [COLUMN] <column name> <drop behavior>
| ADD [COLUMN] <column definition>
| ALTER [COLUMN] <column name> <alter column action>
| ADD <table constraint definition>
| DROP CONSTRAINT <constraint name> <drop behavior>
```

DROP COLUMN句は列をテーブルから削除する。標準SQLでは <drop behavior>オプションを指定することが可能だが、現在の多くの製品はこれをサポートしていない。RESTRICT とCASCADEのオプションは、テーブルの場合と同じである。RESTRICTが指定されていると、他のスキーマオブジェクトで参照されている列を削除することはできない。CASCADE が指定されていると、参照元のスキーマオブジェクトもあわせて削除する。ALTER TABLEを使うときは、まずはRESTRICTオプションを指定して参照するオブジェクトが存在しないか確認することを強く推奨する。

ADD COLUMN 句は、名前のとおりテーブルに新たな列を追加する。新しい列はテーブル内で一意な名前を持たねばならないし、妥当な列宣言としてのルールにも従わなければならない。新しい列の位置は、普通は、既存の列のリストの最後である。

ALTER COLUMN 句は列の定義を変更する。実際のところ、この句でできることは製品によってバラバラであるが、データ型を他の互換可能なデータ型に変えるために使うことが多い（たとえば CHAR(n) のサイズを長くすることはできるが、短くすることはできない。またINTEGER型をREAL型に変えることはできる）。

ADD <table constraint definition>句は、テーブルに制約を追加できる。ここで注意が必要なことは、読者の使うDBMS が、新たに追加した制約によって既存データの妥当性をチェックするかどうかだ。古い製品だと、既存データが制約を満たさなくても放置しておく仕様になっているものもあるかもしれない。

DROP CONSTRAINT句を使うには、制約の名前がわからないといけない。この点でも、制約を作る際に名前を付けておくのは良いプログラミング規約である。もし名前を付けていないと、削除しようとしたときにそれを探し出さなければならないのだが、なにしろSQLエンジンが自動的に割り振った機械的な記号なのでわかりにくい。標準SQL では、そのような名前をどう構成するかのルールまでは決めておらず、スキーマ内で一意であればよい、とだけ定めているだけだ。実際の製品では、ランダムで長大な文字列が割り振られるので、人間には理解しがたい。制約名は、エラーや警告のメッセージにも表示されることを考えれば、デバッグのためにも読みやすい名前を付けておくべきだ。

5.6

属性分割はしてはいけない

属性分割にはいろいろな種類がある。これが起きるのは、単一の属性があって、その値をスキーマ内の複数の場所に保持する場合である。一番ありふれたパターンとしては、値によってテーブルを分割する場合と、同じ値に対して行を分割する場合である。定義で説明するよりサンプルを見たほうが早いだろう。

5.6.1　テーブルレベルの属性分割

社員のテーブルを作るとき、男性社員テーブルと女性社員テーブルの2つを作ったとしたら、読者は即座に「それは1つのテーブルにまとめて、性別コードの列を設けて管理するべきだ」と思うだろう。つまりこれは、性別という属性についてテーブルを分割した例である。このパターンは実に明白だが、もう少し微妙なケースもある。

雑誌の購読者を管理するテーブルを考えよう。ただし、購読者には個人と団体がある。同じ構造を持ったテーブルを2 つ作り、3 つ目として、購読タイプを管理するテーブルを作る。これが分割された属性を持つテーブルだ。

```
CREATE TABLE OrgSubscriptions
(subscr_id INTEGER NOT NULL PRIMARY KEY
   REFERENCES SubscriptionTypes(subscr_id),
 org_name CHAR(35),
 last_name CHAR(15),
 first_name CHAR(15),
 address1 CHAR(35) NOT NULL,
 ...);
```

OrgSubscriptions：団体購読テーブル　subscr_id：購読ID　org_name：組織名
last_name：姓　first_name：名　address1：住所1

```
CREATE TABLE IndSubscriptions
(subscr_id INTEGER NOT NULL PRIMARY KEY
   REFERENCES SubscriptionTypes(subscr_id),
 org_name CHAR(35),
 last_name CHAR(15),
 first_name CHAR(15),
 address1 CHAR(35) NOT NULL,
 ...);
```

IndSubscriptions：個人購読テーブル　subscr_id：購読ID　org_name：組織名
last_name：姓　first_name：名　address1：住所1

```
CREATE TABLE SubscriptionTypes
(subscr_id INTEGER NOT NULL PRIMARY KEY,
 subscr_type CHAR(1) DEFAULT 'I' NOT NULL
   CHECK (subscr_type IN ('I', 'O'))); -- Iは個人(Individual)、Oは団体(Organization)を意味する
```

SubscriptionTypes：購読タイプ　subscr_id：購読ID　subscr_type：購読種別

団体購読の登録は、代表者の名前だけで行うこともできるし（last_name, first_name）、単純に団体名（org_name）で登録してもいい。もし個人購読において特定の人物名が不明の場合は、個人名の変わりに団体名に「現在お住まいの方（'{Current Resident}'）」とでも割り当てておけばよい【訳注13】。

このオリジナルの仕様では、「subscr_id」列はスキーマで一意になる（SubscriptionTypesテーブルには個人も団体も登録されるため）。

このテーブル構造を改善するには、まず3 つのテーブルを1 つにまとめて、「subscr_type」列を本来あるべき場所に戻す。そして次

に、個々の登録に対する制約を付与する。

```
CREATE TABLE Subscriptions
(subscr_id INTEGER NOT NULL PRIMARY KEY
   REFERENCES SubscriptionTypes(subscr_id),
 org_name CHAR(35) DEFAULT '{Current Resident}',
 last_name CHAR(15),
 first_name CHAR(15),
 subscr_type CHAR(1) DEFAULT 'I' NOT NULL
   CHECK (subscr_type IN ('I', 'O')),
   CONSTRAINT known_addressee
     CHECK (COALESCE (org_name, first_name, last_name) IS NOT NULL),
   CONSTRAINT junkmail
     CHECK (CASE WHEN subscr_type = 'I' AND org_name = '{Current Resident}'
                 THEN 1
                 WHEN subscr_type = 'O' AND org_name = '{Current Resident}'
                 THEN 0
                 ELSE 1 END = 1),
 address1 CHAR(35) NOT NULL,
 ...);
```

known_addressee制約は雑誌を購読するには受取人が必要だという意味で、junkmail制約は、既知の個人に対する購読ではない場合は、法人購読に分類するという意味だ【訳注14】。

5.6.2　行レベルの属性分割

社内システムに対するサインイン／サインアウトを管理するこんなテーブルを考えてみよう。

```
CREATE TABLE RegisterBook
(emp_name CHAR(35) NOT NULL,
 sign_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
 sign_action CHAR (3) DEFAULT 'IN' NOT NULL
   CHECK (sign_action IN ('IN', 'OUT')),
   PRIMARY KEY (emp_name, sign_time));
```

RegisterBook：登録者テーブル　　emp_name：社員名　　sign_time：記録時刻
sign_action：アクション種別

ある社員がサインインした時間とサインアウトした時間のペアを得るためには、2つの行を必要とするので、必然的に自己結合を行わなければならない。これは面倒である。正しいテーブルは次のようにするべきだ。

```
CREATE TABLE RegisterBook
(emp_name CHAR(35) NOT NULL,
 sign_in_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
 sign_out_time TIMESTAMP, -- NULLならば現在サインイン中を意味する
   PRIMARY KEY (emp_name, sign_in_time));
```

期間という単一の属性を標準SQLでモデル化するには、2 つの列を使うべきである。だが先のテーブルではこれを2 行に分割していた。もしこれが、緯度と経度だったら、よもや2 行に分割しようなどと考える人はいなかっただろう。

5.7

DDL におけるモデリングクラスの階層

オブジェクト指向モデルにおけるクラスシナリオは、すべてのクラスに共通の属性を持ったルートクラスと、それの下位に位置する特定のサブクラスから成る。たとえば、デヴィッド・ポータスから提供されたサンプルを使ってみよう。これは製品（`Products`）テーブルをルートとして、書籍、CD、DVDをサブクラスとするモデルだ。

```
CREATE TABLE Products
(sku CHAR(17) NOT NULL PRIMARY KEY,
 product_type CHAR(2) NOT NULL
   CHECK (product_type IN ('B', 'C', 'D')), -- Book, CD, DVD の頭文字
 product_title VARCHAR(50) NOT NULL,
   UNIQUE (sku, product_type));
```

Products：製品テーブル　sku：製品ID　product_type：製品種別　product_title：製品名

```
CREATE TABLE Books
(sku CHAR(17) NOT NULL PRIMARY KEY,
 product_type CHAR(2) DEFAULT 'B' NOT NULL
   CHECK (product_type ='B'),
 page_cnt INTEGER NOT NULL,
   FOREIGN KEY (sku, product_type)
     REFERENCES Products (sku, product_type)
       ON UPDATE CASCADE
       ON DELETE CASCADE);
```

Books：書籍テーブル　sku：製品ID　product_type：製品種別　page_cnt：ページ数

```
CREATE TABLE CDs
(sku CHAR(17) NOT NULL PRIMARY KEY,
 product_type CHAR(2) DEFAULT 'C' NOT NULL
   CHECK (product_type ='C'),
 track_cnt INTEGER NOT NULL,
   FOREIGN KEY (sku, product_type)
     REFERENCES Products (sku, product_type)
       ON UPDATE CASCADE
       ON DELETE CASCADE);
```

CDs：CDテーブル　sku：製品ID　product_type：製品種別　track_cnt：曲数

```
CREATE TABLE DVDs
(sku CHAR(17) NOT NULL PRIMARY KEY,
 product_type CHAR(2) DEFAULT 'D' NOT NULL
   CHECK (product_type ='D'),
 play_time INTERVAL HOUR TO SECOND NOT NULL,
   FOREIGN KEY (sku, product_type)
     REFERENCES Products (sku, product_type)
       ON UPDATE CASCADE
       ON DELETE CASCADE);
```

DVDs：DVDテーブル　sku：製品ID　product_type：製品種別　play_time：再生時間

「sku」は最小在庫管理単位（stock-keeping unit）を指す。ここでは製品IDと同じだと考えていい。さて、Productsテーブルの候補キーがオーバーラップしていることに注意してほしい。「sku」列のみで主キーとなるため、本来は一意制約を付けなくても(sku, product_type)が一意になるのは保証されている。それでもなお(sku, product_type)に一意制約を付与したのは、他のテーブルからこの複合キーを参照する整合性制約を付与するには、一意制約が必要だからだ。ほかにもたとえば車を管理するVehiclesテーブル

などを追加するときにも、同様の参照整合性制約を付与すれば済む。

　このようなテーブル間の階層は、さらに深くしていくことができる。たとえば、もしBooksテーブルをペーパーバックとハードカバーにブレイクダウンするならば、次のようなスキーマが適しているだろう。

```
CREATE TABLE Books
(sku CHAR(17) NOT NULL PRIMARY KEY,
 product_type CHAR(2) NOT NULL
   CHECK(product_type IN ('PB', 'HB')),
   UNIQUE (sku, product_type),
   FOREIGN KEY (sku, product_type)
     REFERENCES Products (sku, product_type)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
...);
```

Books：書籍テーブル　　sku：製品ID　　product_type：製品種別

```
CREATE TABLE Paperbacks
(sku CHAR(17) NOT NULL PRIMARY KEY,
 product_type CHAR(2) DEFAULT 'PB' NOT NULL
   CHECK(product_type = 'PB'),
   UNIQUE (sku, product_type),
   FOREIGN KEY (sku, product_type)
   REFERENCES Books(sku, product_type)
     ON UPDATE CASCADE
     ON DELETE CASCADE,
...);
```

Paperbacks：ペーパーバックテーブル　　sku：製品ID　　product_type：製品種別

```
CREATE TABLE Hardbacks
(sku CHAR(17) NOT NULL PRIMARY KEY,
 product_type CHAR(2) DEFAULT 'HB' NOT NULL
   CHECK(product_type = 'HB'),
   UNIQUE (sku, product_type),
     FOREIGN KEY (sku, product_type)
       REFERENCES Books (sku, product_type)
         ON UPDATE CASCADE
         ON DELETE CASCADE,
...);
```

Hardbacks：ハードカバーテーブル　　sku：製品ID　　product_type：製品種別

このモデリング方法のトリックは、識別子（sku）と製品タイプ（product_type）に一意制約を付与しておけば、参照整合性制約でツリーをさかのぼっていける、という点にある。他のクラス構造についても応用の利くトリックである。最近では、こうした階層構造をビューの中に隠して、それらのビューにINSTEAD OFトリガーを使うという方法も使える。

5.8

その他のスキーマオブジェクト

さて、少し定義にこだわろう。データベースとは、DBMS の制御下にあるデータのことだ。DBMSはスキーマ、ルール、データベースに適用する演算子を持っている。スキーマはデータベースにおけるオブジェクトの定義を含む。しかし我々は通常、ただ「データベース」と言うだけで、まるでそれには個々のパートがないかのようだ。

最初に定義された標準であるSQL-89では、ユーザーがSQLでアクセスできる唯一のデータ構造はテーブルとされていた。ここでいうテーブルとは、永続的テーブル（基底テーブル）と仮想テーブル（ビュー）のことである。標準SQL はまた、DBA が他のスキーマオブジェクトを定義することを許しているが、それらの多くは新しい機能で、実際の製品では利用できないか、実装依存のものだ。こうした新機能についても本節で概観しておこう。なお、こうしたオブジェクトのうち、カーソルとシーケンスについては、第7 章で詳しく取り上げるのでそちらを参照してほしい。

5.8.1　スキーマテーブル

SQLエンジンが、スキーマについて必要な情報を保存する通常の方法は、それをテーブルに持つことである【訳注15】。こうしたスキーマテーブルの名前や構造については、ベンダーによってバラバラである。標準SQLも一応、スキーマテーブル群について定義しているが、実装しているベンダーはない。今後もそれを実装するDBMSは現れないと思うが、データ交換用のフォーマットとしてスキーマ情報を生成する機能を実装するベンダーはあるかもしれない。

どんなDBMSにおいても、スキーマテーブルに対しては普通にSQLで問い合わせを行うことができる。DBMSの有志のユーザーグループは、こうしたスキーマテーブルから有益な情報を引き出すためのクエリのライブラリを用意しているものだ。こうしたライブラリは、是非コピーして使わせてもらうべきだろう。

標準SQLはまた、時間関数、照合順序、キャラクタセットなどの情報を持つためのテーブルも定義している。しかし、このテーブルもまた製品によって実装のしかたは不統一である。

5.9

CREATE DOMAIN 文

ドメインは標準SQLにおける新しいスキーマ要素である。ドメインは、一種のインラインマクロである。これを宣言することで、頻繁に使われる列の定義をスキーマ内の1か所にまとめることができる。この機能はそれほど実装が難しくないので、利用できるようになるまで、それほど時間はかからないだろう。構文は次のとおりである。

```
<domain definition> ::=
CREATE DOMAIN <domain name> [AS] <data type>
[<default clause>]
[<domain constraint> ...]
[<collate clause>]
<domain constraint> ::=
[<constraint name definition>]
<check constraint definition> [<constraint attributes>]
```

```
<alter domain statement> ::=
ALTER DOMAIN <domain name> <alter domain action>
<alter domain action> ::=
<set domain default clause>
| <drop domain default clause>
| <add domain constraint definition>
| <drop domain constraint definition>
```

重要なことは、ドメインは基本的なデータ型を使って定義する必要があることだ。別のドメインを使ってドメインを定義することはできない。いったん宣言されると、ドメインは、列に対して通常のデータ型と同じように使うことができるようになる。ドメイン定義におけるCHECK 句には、データ項目が満たすべきチェックディジット、範囲、リストなどの条件を記述する。ドメインはスキーマの1か

所に存在するので、コーディングを簡潔にできる。たとえば、次のようなCHECK制約を考える。

```
CONSTRAINT valid_state_code
CHECK (VALUE IN ('AL', 'AK', 'AZ', ...));
```

これは次のようにドメインを使って書くこともできる。

```
CREATE DOMAIN StateCode AS CHAR(2)
DEFAULT '??'
CONSTRAINT valid_state_code
CHECK (VALUE IN (SELECT state FROM StateCodeTable));
```

この2つのコードを比較すると、2番目のほうがよい。もしドメインがなければ、CHECK句を複数のテーブルで繰り返し使わなければならない。ドメインを使うことで、値の集合を1か所にまとめて定義でき、変更するときもその箇所だけを修正すればよい。

5.10

CREATE TRIGGER 文

トリガーと呼ばれる機能は、多くのSQLのバージョンに存在する。トリガーは、テーブルにあるイベントが起きたことを契機にして、手続き的なコードをデータベースに実行する。参照整合性制約のアクションの一般化と考えてもらえばいいだろう。

手続き型のコードは、通常は実装依存の言語によって記述されるが、中には標準的な手続き型言語のプログラムを使える製品もある。トリガーを使うと、たとえば、取り扱いが終了した製品を自動的に在庫から削除するといった動作を実装できる。

また、標準的なSQL/PSM（Persistent Stored Modules）を使ってトリガーを実装することもできる。ただし、この標準の定義はかなり複雑で、完全にサポートした製品は存在しない。したがって読者は、自分の使っている特定ベンダーの製品がどんな形でトリガーを実装しているか、調べておく必要がある。

宣言的な参照整合性制約に対してトリガーを使用する利点は、参照整合性制約ではできないようなことも、トリガーなら自在にできることだ。欠点は、オプティマイザが手続き型のコードからは何のデータも得られないため、トリガーのパフォーマンスが悪いことと【訳注16】、実装依存のコードになるので、他の環境に移植できないことだ。

私は、参照整合性制約で事足りる場合はトリガーを避けることをすすめる。もしトリガーを使うときは、コードを注意深くチェックして、パフォーマンスが悪化しないようなるべく単純なコードにとどめるよう努力してほしい。

5.11

CREATE PROCEDURE 文

プロシージャは手続き型のコードを使うためのスキーマオブジェクトだ。その点でトリガーやホスト言語のコードと似ているが、2つ大きな違いがある。1つは、プロシージャはパラメータを受け取り、返すことができる。もう1つは、ユーザーセッションから明示的に呼び出すことができる。

トリガーと同様、プロシージャの構文も製品によってバラバラだ。そのため、プロシージャを使うときは、コードを注意深くチェックして、パフォーマンスが悪化しないようなるべく単純なコードにとどめるよう努力してほしい。

SQL/PSMはISO標準である（詳細については『Understanding SQL's Stored Pro cedures』（Melton, 1998）を参照）。しかしまだ、各製品は実装依存の構文を多くの点で残している。

【訳注1】　この機能は、Oracle、PostgreSQLなどで実装されている。なお、SQL-92には、「フル」「標準」「部分」の3つのレベルがあり、どの程度の機能まで実装するかによって対応のレベルが分かれる。あるDBMSが「SQL-92対応」を謳っていたとしても、その程度には実は3段階ある。

【訳注2】　たとえば、以下の演算はすべて結果がNULLになる。

```
1 + NULL = NULL
1 - NULL = NULL
1 * NULL = NULL
1 / NULL = NULL
```

```
NULL / 0 = NULL
```

驚くかもしれないが、プログラミング言語一般でエラーとされる0除算の式においても、被除数がNULLの場合は結果がNULLになる。このようなNULLの伝播は、算術以外でも、文字列や日付の操作においても同様に生じる。たとえば、`'a' || NULL`という文字列連結の結果は、NULLである。

このように、NULLを1か所でも許すとSQLの演算の中で次々にNULLが繁殖する。「propagate」は「（雑草が）生い茂る」という場合にも使う動詞だが、NULLの厄介者ぶりがよく表されている。

【訳注3】 制約に名前を付けることは、主キーや参照整合性制約においても同様に推奨したい。

【訳注4】 CHECK制約の強力さを嬉しそうに語る著者に水を差すようで悪いのだが、ここで1つ「夢のない」話をしておかなくてはならない。こうした高度なCHECKを使うにあたっては、注意すべき点が1つある。それは、このような複雑な制約は更新SQLのパフォーマンスを相当悪くする危険があることだ。もしここで紹介されているような結合や集約を駆使した制約が、INSERT 文やUPDATE 文が実行されるたびに実行されることになれば、大きく性能劣化することは容易に想像がつく。5.1.1 項で見た遅延制約を使えば、制約を施行するSQLの実行をトランザクションが終了するときの一度に減らすことも可

能だが、それでも十分な性能リスクと言える。著者は、本書全体を通じてこうした、主に物理的制約から来る「夢のない」部分についてはあまり言及しない。それは、SQLとRDBのような抽象水準の高いシステムの利点を語るうえでは正しい態度ではあるが、現場のエンジニアとして、訳者は今後もたびたび、訳注において、物理的制約に起因する夢のない話を補いたい。ご理解いただければ幸いである。

【訳注5】 SQL Serverのように、NULLも1つの値とみなして一意性制約のチェック対象とする実装も存在する。もちろん標準違反である。

【訳注6】 なぜ参照される側の列に主キーまたは一意制約が必要とされているのか、その理由は明言されていないが、簡単に推測できる。値の存在をチェックするとき、もしこうした制約がなければ、テーブルをフルスキャンしなければならない。これは性能的に不利である。しかし、主キーと一意制約はインデックスを必要とするため、値の存在チェックの際も、インデックスを検索することで高速化できる。

【訳注7】 『ジェリー・シュプリンガー・ショー』は、米国のワイドショー的な番組。一般視聴者から選ばれたゲストの恋人、夫婦、親兄弟をスタジオに呼んで、ホストのジェリー・シュプリンガーがドロドロした関係を暴露する。浮気と不倫はこのショーの定番ネタである。

【訳注8】　SQL文に含まれる列のすべてをカバーするインデックスが存在する場合、そのSQLはインデックスのみを使用して、テーブルにアクセスしないという実行計画を立てることができる。この機能を「インデックスオンリースキャン」と呼び、Oracle、DB2、PostgreSQLなどが実装している。これがパフォーマンス改善に有効なのは、概してテーブルに比べてインデックスのほうがサイズがとても小さく、データの読み込み量が少ないからである。ローベーストアにおいて、擬似的にカラムベーストアを表現する方法、と考えてもいいかもしれない。

【訳注9】　選択公理は1904年に数学者エルンスト・ツェルメロが定式化した集合論の公理の1つ。その定義は、空集合ではない集合を要素とする集合、つまり「集合の集合」（これを集合論で「集合族」と呼ぶ）があった場合、各集合から要素を1つずつ選び出して新しい集合を作ることができる、というもの。集合族が有限集合だと前提とすれば、当たり前のことを言っているように思えるが、無限集合だった場合、果たしてそのような手続きが可能かは自明とは言えない。

今回のケースでは、次のような集合S1～S6を要素とする集合族を考える。

```
  S1 ={(0, 9)}
S4 ={(3, 2), (3, 5)}
```

S2 ={(1, 1), (1, 5)}　　S5 ={(4, 2)}

S3 ={(2, 1), (2, 7), (2, 11)}　　S6 = {(5, 5)}

ここから、たとえば、以下のようにS1～S6の要素を抜き出して新たな集合を作ることができる。これが、移行先テーブル（Destination）に挿入する候補となる。

{(0, 9), (1, 1), (2, 1), (3, 2), (4, 2), (5, 5)}

もちろん、こうした作られた集合は、まだ列「a」に対する一意制約を満たすことは保証されているわけではない。

【訳注10】　サブクエリ内の「P2.pair_nbr < P1.pair_nbr」と「(P1.a = P2.a OR P1.b = P2.b)」という条件の意味を疑問に思う読者もいるかもしれない。前者は、著者も言うように、それぞれの行の可能なすべてのペアを（順列ではなく組み合わせで）得る方法である。これはSQLで組み合わせを得る常套手段である。

後者は、NOT EXISTSによって全称量化（すべての行が満たすべき条件）を表現する、これもSQLで頻繁に使うテクニックである。SQLには、存在量化子に対応するEXISTS 述語はあるが、全称量化子に対応する述語がな

いので、これをNOT EXISTS で代用するというテクニックを多用する（詳細は第22 章参照）。NOT EXISTSは、「(P1.a = P2.a OR P1.b = P2.b)」に合致しない行を選択するが、これは裏を返すと、全行について「(P1.a <> P2.a AND P1.b <> P2.b)」を満たす行を選択する、という意味である。この条件が、「a」列と「b」列の一意制約を表していることがわかるだろう。

【訳注11】　NP完全のNPは、Non-deterministic Polynomial time（非決定性多項式時間）の略で、非決定性チューリングマシンによって多項式時間で解くことができる問題の集合を意味する。NP 完全問題とは、総当たり方式などで答えをしらみつぶしに探せば答えが求まる、ということはわかっているが、問題の大きさが膨らむほど計算量が爆発的に増えてしまい、現実的な時間で解くことが難しい（と予想されている）問題を指す。代表的な問題に「ナップザック問題」などがある。SQL のアルゴリズムは、まさに総当たり方式で解を探すものになりがちなため、NP完全問題を解くには速度が遅く向かない。

【訳注12】　日本の場合、住所の並び順はこの逆で、郵便番号、都道府県、市町村、番地、というように大きな地域から並べるのが一般的なので、日本ではこの順番で並べるのがよい、というように読み替えてほしい。

【訳注13】　Current Residentという表現は、住所と居住者の関連が定かでない場合（この例では、購読者が住所変更を知

らせないまま引っ越してしまったような場合）の郵便物の宛先によく使われる。

【訳注14】　一見すると、junkmail制約のELSE句は不要に見えるかもしれないが、subscr_type列などがNULLの場合、UNKNOWNとなり、WHEN句の評価が失敗し、ELSE句まで到達することがある。

【訳注15】　こうしたスキーマテーブルの実装例としては、Oracleの「データディクショナリ」、PostgreSQLの「情報スキーマ」などがある。いずれもテーブル、インデックス、プロシージャなどスキーマオブジェクトについてのメタ情報を含む。

【訳注16】　DB2などでは、トリガーで使用されるSQLに対しても、オプティマイザが統計情報から実行計画を作成することがある。

6

手続き型プログラミング、半手続き型プログラミング、宣言型プログラミング

Procedural, Semiprocedural, and Declarative Programming

本章では、プログラミングとプログラミングを行う際の考え方（マインドセット）について論じる。SQLは手続き型のコードをスキーマ内に保持することができるし、SQL自身、手続き型のホスト言語の内部で動作する。だが、ファイルシステムに対する手続き型のプログラミングと同じように、SQL をプログラミングするのは避けるべきだ。SQL は宣言型言語であり、手続き型のプログラミングとは異なる考え方を必要とするからだ。

6.1

ソフトウェア開発の基礎

とはいえソフトウェア開発の基礎は、手続き型言語でもSQLでも変わらない。まずはAlgol-60 の子孫である手続き型言語から学び始めることにする（LISP、APL、FP、Haskellといった毛色の変わった言語から始める人は例外的だろう）。手続き型のコードに適用できる原則は、それ以外の原理に基づいた言語にも適用できる。

1970年代の後半、ローカルスコープの原則と、1つの入り口と1つの出口しかないモジュールに限定することで、より良い（つまりより速く、間違いが少なく、保守が簡単な）コードが書けることがわかった。我々はGOTOを排除し、単純な制御構造を使うことにした。これが構造化プログラミング革命だった。

6.2

凝集度

凝集度（cohesion）は、モジュールがどれだけ1つのまとまりになっていて、1つだけのタスクを行うかを示す。モジュールはなるべく高い凝集度を持つべきである。モジュールの名前は<verb> <object>というフォーマットで名付けるべきである。ここで、<object>はデータモデルにおける明確な論理的単位であり、<verb>は単一の明確な行為である。凝集度にはいくつかのタイプがある。これを最低から最高まで順位をつけてみよう。

1. **偶発的凝集**：モジュールが恣意的に集められており、モジュール同士はまったく関係のない機能を持つ。たとえば、{ブリトニー・スピアーズ、イカ、自動車}というモジュールは、でたらめな単語の寄せ集めである。偶発的凝集のSQL における好例は、OTLTという設計のアンチパターンである。これについては第8章で論じる。
2. **論理的凝集**：モジュールが論理的に同じものであるためにグループ化されている。SQLでのサンプルは、任意のテーブルに対する更新、挿入、削除を行う汎用モジュールである。これはブリトニー・スピアーズにも、イカにも、自動車にも適用される。動的SQLによって実装されることが多い。
3. **時間的凝集**：モジュールの実行タイミングによってグループ化されている。たとえば、システム全体の初期化を行うモジュールをまとめるような場合だ。
4. **手続き的凝集**：モジュールが常に特定の実行順序に従うためにグループ化されている。たとえば、ユーザーがデータベースに

ログオンする際に、権限をチェックし、サインインのログを記録する。

5. **通信的凝集**：モジュールが同じデータ要素に操作を行うためにグループ化されている。あるプロシージャの中で、同じ列を更新する複数のUPDATE文を想像してほしい。
6. **逐次的凝集**：ちょうど組み立てラインのように、あるモジュールのアウトプットが他のモジュールのインプットになることからグループ化されている。次々にデータを受け渡していく一時テーブル群を想像してほしい。
7. **機能的凝集**：モジュールが常に明確に定義された単一のタスクを行う。ちょうど数学の関数のように。これがモジュールに最も望ましい性質であり、関数的プログラミング言語の基礎である。

もし読者がプログラミング入門編の授業を受講したときにこれらの勉強をサボっていたなら、詳細な定義をいま一度復習することをすすめる。

6.3

結合度

結合度は、複数のモジュールが互いにどのように依存しているかを示す概念である。もしモジュールが特定の順序に従って実行される必要があるならば、両者は非常に強い結合度を持っていることになる。反対に、もし順序に関係なく実行することができるならば、その結合はゆるく弱い。今度は結合度を悪い順から良い順に並べてみよう。

1. **内容的結合**：あるモジュールが別のモジュールの内的なデータに依存する、または変更する。
2. **共通的結合**：複数のモジュールが同じグローバルデータを共有する。
3. **外的結合**：複数のモジュールが外部から与えられたデータフォーマット、通信プロトコル、デバイスインターフェイスを共有する。
4. **制御的結合**：モジュール間でフラグの受け渡しによって実行を制御する。これはSQLプログラミングにおいてビットフラグを使うのがよくない理由の1つである。
5. **スタンプ結合（データ構造結合）**：複数のモジュールが複合的なデータ構造（たとえば構造体）を共有し、その一部だけを使用する。SQLでは、この結合はビューによって実現される。
6. **データ結合**：複数のモジュールが単純なデータ要素を共有する。モジュール間でパラメータを受け渡す場合を想像してほしい。この場合、共有されるデータだけが受け渡される。

7. **メッセージ結合**：複数モジュールが互いに依存しておらず、パラメータを持たないメッセージを交換するための共通インターフェイスだけを使う。これは最もゆるい結合である。すぐれてオブジェクト指向的な結合だが、SQLにおいても、トリガーや例外ハンドラといった機能で見ることができる。

結合度については、拙著『SQL Programming Style』（Morgan Kaufmann、2005）のストアドプロシージャについての章でも取り上げている。また、あわせてデマルコ、ヨードン、コンスタンティン、マイヤーズといったソフトウェア開発のパイオニアたちの仕事を知ることも重要である。こうした知識はSQLプログラミング以前に、エンジニアにとっての基礎教養と言ってよい。どのような言語を習得するにせよ、知っておくべき内容である。

6.4

大いなる跳躍

多くのプログラマは、手続き型の考え方から宣言型の考え方へ移行する際に、大きな断絶を感じる。そのため、移行は急に起こるものではなく、徐々に起こる。まずは第一段階として、手続き型のパラダイムから半手続き型とでも言うべきスタイルへの進化が起こる。

これは、私が考え方を変えるときの方法そのものである。一足飛びに成長するのではなく、一歩一歩着実に階段を上っていく。史上最初の映画は、カメラを固定して舞台を撮影するという方法で撮られた。これは、人々が何千年もの間、舞台演劇を鑑賞してきた方法だ。W.D.グリフィスが最初に2本のリールで撮影を行ったのは、1910年のことである【訳注1】。彼以前には、誰一人としてリールを2 つ使って撮影しようなどとは考えなかった。1911 年から12 年にかけて、彼とカメラマンのビリー・ビッツァーは、撮影中にカメラとレンズを動かすという試みを始めた。クローズアップやソフトフォーカスという撮影技術はこうして生まれた。グリフィスは、さしずめ映画界のE. F. コッドである。

6.4.1　ありがちな間違い

Web上のニュースグループというのは悪いサンプルの宝庫である。あるとき、整数型の列a、b、c、dを持つ簡単なサンプルテーブルを使った質問が投稿されていた。質問者のやりたいことは、d =(a + b + c)を保証するようなトリガーを書くことだった。そもそもなぜ彼がトリガーを使うことに決めたのかという点を無視すると、この問題を解く方法には以下のようなものがある。

方法1

CREATE PROCEDURE：手続き型の答え。ストアドプロシージャの中に`UPDATE`文を書き、必要なタイミングで呼び出す。もし呼び出すのを忘れたら、テーブルの状態は不整合になる。

方法2

CREATE TRIGGER：手続き型の答え。投稿者のオリジナルの解であり、常にテーブルの状態を整合的に保てるという点で、ストアドプロシージャよりも優れている。ただし、テーブルが更新されるたびにトリガーが実行されることになり、物理的なディスクアクセスが多く、パフォーマンスが悪いのが欠点だ。

方法3

SELECT 文：宣言的な答え。単純に、`SELECT`文の中に`(a + b + c) AS d`という形で列`d`を記述すれば、クエリが実行されるタイミングで`d`が計算されるため、常に正しい値が保証される。

方法4

CREATE VIEW：宣言的な答え。ビューもまた列`d`を必要なタイミングで計算する。そう、ユーザーに見えていないだけで、計算は行われているのだ。この点を忘れないように。

方法5

計算列：宣言的な答え。計算列も基本的なアイデアはビューと同じだが、違うのは基底テーブルに列`d`を持つことだ。投稿者のケースでは、これが最適解のように思えた（常にそういうわけではないが）。

ところで、この事例における本当の問題はこうである。そもそも、なぜ彼はトリガーが解になると考えたのだろうか？ おそらくそれは、トリガーがdの値を物理的に実体化するからだろう。ファイルシステムでは、（計算列のような）仮想フィールドが存在しない。フィールドは常に物理的実体である。計算列が作られるのは、あくまでホスト言語のローカル変数においてである。プロシージャは呼び出しを忘れるかもしれないし、そうするとデータが不正な値になる。そのため、トリガーのほうがプロシージャより優れているように思えたのだろう。この投稿者は、手続き型の考え方から一歩も抜け出せていなかったわけである。

6.4.2　一歩を踏み出せ

2010 年2月、私はニュースグループの議論で、手続き型の考え方から一歩を踏み出すサンプルに出くわした。コードの詳細は論点と関係ないので、重要なところだけをかいつまんで話すと、その議論のスレッドは、あるユーザー定義関数がうまく動作しない、という質問から始まっていた。

> 以下のような整数を受け取って、その組み合わせによってVARCHAR のデータを返すコードがあります。しかし、6行目のCASE が始まる行でエラーが発生します。CASE 文は大量にあって、これをデバッグするのは私の手に負えません。何が悪いのか途方に暮れています。

そのコードというのは、BETWEENとORを使って、CASEの中にCASEを入れ子にする複雑なものだった。整数のペア(x, y)を引数にとって、3 つの値、たとえば {'a', 'b', 'c'} から1 つの値を返す、というものだった。繰り返すが、コードの詳細はどうでもいい。ただ、その関数は非常に複雑な数式で組み立てられていた、ということである。

これに対し、ある人が、SQLではCASEは文ではなく式なので、構文的に間違っているという点を指摘した。また、返す文字列は常に1文字であったので、VARCHAR(1)ではなくCHAR(1)のほうが適当だ、という指摘も出た。CASE式は最初に手続き型言語を学んでIF-THEN-ELSE文を使っていた人には、常に紛らわしいものだ（CASE式については第18章で詳しく論じる）。

こうした点を修正したコードサンプルは次のようになる。

```
CREATE FUNCTION Find_Foobar (IN in_x INTEGER, IN in_y INTEGER)
RETURNS CHAR(1)
WITH EXECUTE AS CALLER
AS
BEGIN
RETURN (<< xとyを使った複雑なCASE式 >>);
END;
```

さらに別の人から、あらかじめCASE式の計算を行っておいて、結果をテーブルに保持しておいてはどうか、という提案があった。これは良いアドバイスである。というのも、整数(x, y)の組み合わせは、たかだか数千パターンぐらいのケースしかないということがわかっていたからである。数千行程度のサイズならば、参照用のテーブルを作ってもテーブルへのアクセスコストは十分に小さく、実用

的だ。テーブルはおそらく読み取り専用であろうから、これをメモリキャッシュに保持しておけばさらにパフォーマンス向上が見込めるし、多くのセッションで共有できる。これは、ユーザー定義関数の中にロジックを閉じ込めてしまう解よりも大きな利点を持っていると言える。もしこの程度のテーブルすら乗せられないぐらいメモリが足りないというなら、いまやメモリ価格は年々下落するようなご時世だ。メモリを買い足すことを考えたほうがよいだろう。

テーブルとインデックスは、オプティマイザによってパフォーマンス最適化の恩恵を受けることもできる。また、DBMSによっては、クエリに必要な列をすべてカバーするインデックスを作ることで、テーブルへのアクセスをスキップすることすらできる【訳注2】。

反対に、もしデータをプロシージャのコードに閉じ込めしてしまえば、それを共有することはできないし、呼び出されるたびに計算しなければならない。インデックスを使うこともできない。このように、ユーザー定義関数というのは極めて閉鎖的な方法である。

標準SQL/PSMは、`[NOT] DETERMINISTIC`オプションをプロシージャ宣言に付けることができる。これはコンパイラに、当該関数が同じ引数には常に同じ答えを返すかどうかを知らせるためのものだ（標準SQLでは、パラメータはプロシージャ宣言の形式的なプレースホルダであり、引数（argument）は、関数が呼ばれたときにそのプレースホルダに渡される具体的な値を意味する）。

非決定的な関数であれば、呼び出されるたびに計算しなければならない。もしクエリオプティマイザが、特定のプロシージャが決定的かどうかを知らなければ、とにかく毎回計算するしかない。

テーブルを利用する解のサンプルは以下のとおりだ。

```
-- 参照テーブルを作る
CREATE TABLE Foobar
(x INTEGER NOT NULL,
 y INTEGER NOT NULL,
 foobar CHAR(1) NOT NULL,
     PRIMARY KEY (x, y));

--テーブルにデータを入れる
INSERT INTO FooBar (x, y, foobar)
SELECT X_CTE.x, Y_CTE.y, << xとyを使った複雑なCASE式 >> AS foobar
  FROM (SELECT seq
          FROM Series
         WHERE seq BETWEEN 100 AND 300) AS X_CTE(x)
       CROSS JOIN
       (SELECT seq
          FROM Series
         WHERE seq BETWEEN 1 AND 100) AS Y_CTE(y);
```

Series テーブルは、非常によく使う補助テーブルである。seq列には適当な整数n までの1～nの整数値が格納されている。BETWEENでseq列の範囲を指定しているのは、次ページで見るように、xとyの範囲に合わせたためだ。経験を積んだSQLプログラマは、新しいプロジェクトが始まると同時に、この数列用のテーブルとカレンダー用のテーブルを作成するものだ。

数列を作る手段としては再帰共通表式を使う方法もあるが、これは実行コストが高いうえに手続き的な道具である。これは半手続き的な考え方だ。

さて、テーブルが用意できたら、残りのコードも修正しよう。

```
CREATE FUNCTION Find_Foobar
(IN in_x INTEGER, IN in_y INTEGER)
RETURNS CHAR(1)
WITH EXECUTE AS CALLER
AS
BEGIN
RETURN
COALESCE
((SELECT foobar
    FROM Foobar
   WHERE x = in_x
     AND y = in_y), 'A');
END;
```

COALESCE関数を使っている理由は、スカラクエリはWHERE句の条件に合致する行が見つからなかった場合、NULLを返すため、関数Find_FoobarがNULLを返さないよう、NULLを'A'に変換するためだ。もしこのクエリが常に行を見つけることができるならば、COALESCE関数は不要である。

ここで、宣言型プログラミングについて考えてみよう。SQL ではそれは、DDLにおける制約を活用するということだ。このサンプルでは、主キー以外にテーブルは何の制約も持っていない。これが雑誌記事やニュースグループの投稿の問題点だ。サンプルを提示するとき、皆いともあっさり制約を省略してしまう。制約は、ファイルを宣言するときには使わないものだったので、いまだにその考え方をひきずっている。だが忘れがちなことだが、SQLは、DDL、DML、DCLという3 つのサブ言語の集合なのであり、この3 つは一緒に使って初めて効果的なのだ。特に、DDL制約はDMLのオプティマイザがより良いアクセスパスを考えるために利用するもので、非常に重要なのだ。

もともとのサンプルコードにおいて、プロシージャの中にCASE式が記述されていたということは、(x, y)のペアは、よくあるパターンが予想できる、ということだ。実際、下限についてはx が100、y が1と決められていたが、上限は特になかった。おそらく、質問者は、大半のペアは、狭い範囲内に収まると期待しており、そこから外れる入力値に対してはエラーを返す想定だったのだろう。これはG.ワインバーグが「データの合理性チェック」と呼んだ現実的な態度である。この原則はまた、ジップの法則とか、医療分野における「馬を探せ、シマウマを探すな」の法則として知られている【訳注3】。

そのようなわけで、まずは単純に、xとyには上限と下限があると想定してみよう。テーブル定義はこうなる。

```
CREATE TABLE FooLookup
(x INTEGER NOT NULL
     CHECK (x BETWEEN 100 AND 300),
 y INTEGER NOT NULL
     CHECK (y BETWEEN 1 AND 100),
 foobar CHAR(1) DEFAULT 'A' NOT NULL
     CHECK (foobar IN ('A', 'B', 'C')),
     PRIMARY KEY (x, y));
```

`DEFAULT 'A'`は、`foobar` 列に明確な値が存在しなかった場合はとりあえずAとするためだ。これによって、プロシージャのCOALESCE関数を削除できる。もしxかyのどちらかが範囲外の値を取りうるならば、`CHECK` 制約を削除するしかない。あるいは、上限か下限のどちらかが決まっていない、というケースならば、`CHECK (y >= 1)`のような単純な不等式でCHECK制約を記述する。

オリジナルのコードでは、xとyの組み合わせに対するfoobarの値は、CASE式の入れ子で表現されていた。これは実物を見たほうがわ

かりやすいだろう。

```
CASE
WHEN x BETWEEN 100 AND 200
THEN CASE
    WHEN y IN (2, 4, 6, 8) THEN 'B'
    WHEN y IN (1, 3, 5, 7, 9) THEN 'C'
END
WHEN x BETWEEN 201 AND 300
THEN CASE
    WHEN y IN (2, 4, 6, 8, 99) THEN 'C'
    WHEN y IN (3, 5, 7, 9, 100) THEN 'B'
END
ELSE 'A'
END
```

これは言ってみれば、制約をDML で記述しているようなものだ。このため、UPDATE、INSERT、SELECTといった、この式が記述された文が実行される間のみ有効となる。だが我々が本当にやりたいのは、DDLにおける制約によって、現在および未来にわたってこのテーブルを利用するすべての文に有効な制約を実現することのはずだ。これを実現するトリックは、テーブルにそれぞれのパラメータの範囲に対して下限と上限の値を保持することだ。

```
CREATE TABLE FooLookup
(low_x INTEGER NOT NULL,
 high_x INTEGER NOT NULL,
     CHECK (low_x <= high_x),
 low_y INTEGER NOT NULL,
 high_y INTEGER NOT NULL,
     CHECK (low_y <= high_y),
 foobar CHAR(1) NOT NULL
     CHECK (foobar IN ('A', 'B', 'C')),
     PRIMARY KEY (low_x, high_x, low_y, high_y));
```

そうすると、このCASE式の対応表は表6.1（次ページ参照）のようになる。

安全のため、`default 'A'`を範囲外の値として設定しておく。私は–9999と9999を上限と下限の値として使ったが、この値は適宜変更してほしい。

このテーブルに対するクエリは、下限と上限に対する`BETWEEN`を使うことになる。

```
SELECT F.foobar, ...
  FROM FooLookup AS F, ...
 WHERE my_x BETWEEN F.low_x AND F.high_x
   AND my_y BETWEEN F.low_y AND F.high_y
   AND ...;
```

こうした宣言型の解が常に最良の方法かどうかは、読者が各自テストしてほしい。

表6.1　関数参照テーブル

low_x	high_x	low_y	high_y	foobar
100	200	2	2	B
100	200	6	6	B
100	200	8	8	B
100	200	1	1	C
100	200	3	3	C
100	200	5	5	C
100	200	7	7	C
100	200	9	9	C

201	300	2	2	C
201	300	4	4	C
201	300	6	6	C
201	300	8	8	C
201	300	99	99	C
201	300	3	3	B
201	300	5	5	B
201	300	7	7	B
201	300	9	9	B
201	300	100	00	B
301	9999	101	9999	A
-9999	99	-9999	0	A

6.5

半手続き型への書き換え

手続き型の考え方から宣言型の考え方への移行は、多くのプログラマにとっては大ジャンプである。前述のように、この移行は一度に起きるのではなく、まずは半手続き型スタイルへ、それから宣言型へと、一歩一歩進む。

手続き型のコードはプロシージャの中に記述される。だがその逆、つまりプロシージャの中に記述されるのは常に手続き型のコードだろうか？ SQLにおいてプロシージャは、BEGINとEND で囲まれたブロックにすぎない。極端な話、IF-THEN-ELSEやWHILE-DOループといった制御構造が一切なく、SQL文が順番に並べられているだけのプロシージャを書くこともできる。このようなプロシージャは、手続き型だろうか宣言型だろうか。私は、これは宣言型のコードだと思う。手続き型のコードには、IF-THEN-ELSEやWHILE-DOループが必要だというのが私の考えだ。

IF-THEN-ELSE文の大半はCASE式で置き換えることができる。SQLにCASE式が取り入れられる前は、手続き型のコードをSQLに変換するのは一苦労だった。その古典的な一例が、書籍の値引き問題だ。25ドル以上の本は一律10%オフとし（これは大々的に宣伝する）、25ドル未満の本は一律15%値上げとする（これは宣伝しない）。単純に考えると次のように書きたくなる。

```
BEGIN
UPDATE Books
   SET price = price * 1.15
 WHERE price < 25.00;

UPDATE Books
   SET price = price * 0.9
 WHERE price >= 25.00;
END;
```

だがこれはうまくいかない。現在の価格が24ドルの本は、最初のUPDATE 文で27.6ドルに上昇し、2 番目のUPDATE 文で24.84ドルに下落する。これはやりたかったこととは違う。UPDATE 文の順序を入れ替えてもうまくいかない。

かつては、この問題を解決する手段はカーソルだった。1 行ずつループして、IF-THEN-ELSE 文で値下げするか値上げするか決める。このときのテーブルは、あたかもファイルのようだ。しかし今日ではこんな方法を使う必要はない。我々には宣言型の道具であるCASE式がある。

```
UPDATE Books
SET price = CASE
            WHEN price < 25.00 THEN price * 1.15
            ELSE price * 0.9
            END;
```

ループを置き換える手段はいろいろあるが、一般的にはテーブルに対する集合指向的な操作によって可能だ。RBAR（「リバー」と読む。これはジェフ・モダンが考えた言葉で、Row By Agonizing Row、つまり「1行ずつ処理する」の略である）は絶対に避けなければならない。ループを回避する別の手段は、ROW_NUMBER 関数を使って行に連番を割り振ることである。ROW_NUMBERや広くウィンド

ウ関数を使ってループを回避するサンプルとしては、第31章の中央値を求めるクエリが好例である。

6.5.1　データテーブル対ジェネレータコード

数列テーブルを再帰によって作るのは、頻繁に行われている。これは手続き型の考え方だ。もっとよいのは、参照テーブルを使って置き換えることだ。この点を考えるために、簡単な問題を使おう。フィボナッチ数列の計算である。手続き型、半手続き型、宣言型のそれぞれの考え方で、フィボナッチ数列を作ってみよう。

問題は、n 番目までのフィボナッチ数列のテーブルを作り、n+1 番目のフィボナッチ数を計算することだ。フィボナッチ数列がどのような数列であるかについては、節末の参考文献リスト（『不思議な数列フィボナッチの秘密』など）を参照してほしい。この数列の定義は、通常は再帰を用いて次のように記述される。

```
CREATE FUNCTION fib(n IN INTEGER)
RETURN INTEGER
IS
BEGIN
  IF n = 0
    THEN RETURN 0;
  ELSIF n = 1
    THEN RETURN 1;
  ELSIF n > 1
    THEN RETURN (fib(n-2) + fib(n-1));
  END IF;
END;
```

6.5.2　参照を計算で置き換える

私が覚えているなかで、参照を計算で置き換える最も極端な例は、数十年前にジョージアの技術フォーラムで見たものだ。我々

は、CDC Cyberシリーズのコンピュータを持っていて、そのハードウェアは当時としては浮動小数点計算で最高性能を誇っていた【訳注4】。IBMのマシンすら凌駕していたのだ。

我々のセールスチームは、FORTRAN の中でテーブル参照をするよりは、チェビシェフ多項式の曲線にデータを当てはめていくほうがよいと考えた【訳注5】。そこで、チェビシェフ多項式を生成するソフトウェアパッケージを作った。数式は複雑すぎて人間には理解しがたいものだが、たしかにディスクからデータを読み出すよりも高速で、データもうまく多項式のパターンにフィットした。こういう真似は、読者がスーパーコンピュータを所有してでもいない限りやってはいけない。ここには、パフォーマンス以外に、保守性の問題もあるのだが、その点についてはいまは立ち入らない。

6.5.3　フィボナッチ数列

この数列の問題は、読者の考え方を変化させる格好の例題だ。この数列を(n = 0)から始めるか、(n = 1)から始めるかには少し議論がある。だが、要するにこの数列はfib(n) = (fib(n - 2)＋fib(n - 1))という意味だ。数列は以下のようになる。

```
0, 1, 1, 2, 3, 5, 8, ...
```

フィボナッチ数列のテーブルを計算するには、まず1つの方法として、ループを使う完全に手続き型の方法がある。

```
CREATE TABLE Fibonacci
(n INTEGER NOT NULL);

CREATE PROCEDURE ProcFib
IS
  a INTEGER;
  b INTEGER;
  fib INTEGER;
BEGIN
  a := 0;
  b := 1;
  fib := 0;
  WHILE (fib < 1000) LOOP
    fib := a + b;
    a := b;
    b := fib;
    INSERT INTO Fibonacci VALUES (a);
  END LOOP;
END;
```

これを再帰共通表式で置き換えると、次のようになる。

```
WITH RECURSIVE Fib(a, b) AS
(SELECT a, b FROM (VALUES (0, 1)) AS Foobar(a,b)
UNION ALL
SELECT b, a+b
  FROM Fib
 WHERE b < 100)
SELECT a FROM Fib;
```

これはまだプログラマ的な考え方にとらわれており、数学者の考え方ではない。フィボナッチ数列は、いわゆる閉式である【訳注6】。つまり、その値をループや再帰を使わずに計算できる。黄金比（ファイ）を使った単純な数式を使えば、次のようにループなしで書ける。

```
CREATE FUNCTION FuncFib(n IN INTEGER)
RETURN INTEGER
IS
    phi DECIMAL(35, 25);
    intval INTEGER DEFAULT 0;
BEGIN
    phi := 1.61803398874989484820458683;
    intval := ROUND (((POWER (phi, n) - POWER (1.0 - phi, n)) / SQRT (5.0)), 0);
    RETURN intval;
END;
```

ファイの100万桁までの近似値は、以下のWebサイトでダウンロードすることができる。

http://www.goldennumber.net/phi-million-places/

ファイについては、以下の資料も参照。

- *The Golden Ratio: The Story of PHI, the World's Most Astonishing Number*, Mario Livio （『黄金比はすべてを美しくするか?最も謎めいた「比率」をめぐる数学物語』マリオ・リヴィオ著、斉藤隆央訳、早川書房、2005［文庫版2012］）
- *The Golden Section: Nature's Greatest Secret*, Scott Olsen（『黄金比　自然と芸術にひそむもっとも不思議な数の話』スコット・オルセン著、藤田優里子訳、創元社、2009）
- *The Divine Proportion*, Herbert Edwin Huntley
- *The Fabulous Fibonacci Numbers*, Alfred S. Posamentier（『不思議な数列フィボナッチの秘密』アルフレッド・S・ポザマンティエ／イングマル・レーマン著、松浦俊輔訳、日経BP社、2010）
- *A Mathematical History of the Golden Number*, Roger Herz-Fischler

- *The Golden Section (Spectrum)*, Hans Walser（『黄金分割』H.ヴァルサー著、蟹江幸博訳、日本評論社、2002）

6.6

述語で使うための関数

半手続き型の考え方に基づく非常に奇妙なコードサンプルの1つは、SQL Serverのプログラマの間に見出せる。彼らは、制約の中にユーザー定義関数を作ることを好む。バスケットから卵を取り出して箱（カートン）に詰めるパズルを考えよう。1つの箱に入る卵は12個以下である。

```
CREATE TABLE Baskets
(basket_id INTEGER NOT NULL PRIMARY KEY);
```

Baskets：バスケットテーブル　　basket_id：バスケットID

```
CREATE TABLE Cartons
(carton_id INTEGER NOT NULL PRIMARY KEY);
```

Cartons：箱テーブル　　carton_id：箱ID

```
CREATE TABLE Eggs
(basket_id INTEGER NOT NULL
     REFERENCES Baskets(basket_id),
 carton_id INTEGER NOT NULL
     REFERENCES Cartons(carton_id));
```

Eggs：卵テーブル　　basket_id：バスケットID　　carton_id：箱ID

```
CREATE FUNCTION CartonCount(in_carton_id IN INTEGER)
RETURN INTEGER
IS
    intval INTEGER DEFAULT 0;
BEGIN
    SELECT COUNT(*) INTO intval FROM Eggs
     WHERE carton_id = in_carton_id;

    RETURN intval;
END;

ALTER TABLE Eggs ADD CONSTRAINT carton_limit
    CHECK (CartonCount(carton_id) BETWEEN 1 AND 12);
```

オリジナルのEggsテーブルにはキーがなく、オプティマイザはまったくキーについての情報を使うことができない。ユーザー定義関数CartonCountは、コードをさらに複雑なものにしているし、第一このコードは実装依存である【訳注7】。宣言的なDDLは次のようなものであるべきだ。

```
CREATE TABLE Eggs
(basket_id INTEGER NOT NULL,
 carton_id INTEGER NOT NULL,
     PRIMARY KEY (basket_id, carton_id),
 egg_cnt INTEGER DEFAULT 1 NOT NULL
     CONSTRAINT carton_limit CHECK (egg_cnt BETWEEN 1AND 12));
```

Eggs：卵テーブル　basket_id：バスケットID　carton_id：箱ID　egg_cnt：卵の数

主キーによって、バスケットと箱のIDで一意になることがわかるし、制約carton_limitの中にあるBETWEEN述語も、オプティマイザが利用することができる。

6.7

手続き VS 論理的分解

集合という観点から考えることをしてこなかったSQLプログラマは、しばしば分解を問題解決技法として誤用する。もちろん、問題を相互に独立の下位問題へ分解し、それぞれを各個撃破したあとで再統合することは、優れた経験則だ。だが、万能薬ではない。SQLでは、再統合をUNION ALL や使い捨てテーブルへの一連のINSERTによって行うことが珍しくないが、これは推奨できない。それは手続き的な分解／統合なのだ。我々がほしいのは、論理的な分解／統合の技法である。簡単なオフィスを管理するテーブルを使って、これを考えよう。

```
CREATE TABLE Offices
(duns CHAR(9) NOT NULL
     CHECK (    duns BETWEEN '000000000' AND '999999999'
       AND UPPER(duns) = LOWER(duns)),
 office_id INTEGER NOT NULL
     CHECK (office_id > 0),
 office_type CHAR(1) DEFAULT 'N' NOT NULL
     CHECK (office_type IN ('C', 'N')),
 dba_name VARCHAR(15) NOT NULL,
     PRIMARY KEY (duns, office_id));
```

Offices：オフィステーブル　duns：ダンズ　office_id：オフィスID
office_type：オフィス種別　dba_name：DBA名

ダンズ（DUNS）は企業を一意に識別する業界標準の識別子である（第4 章参照）。オフィスIDは、企業内のいろいろな所在地から作られる整数である。我々が気にすべきオフィス種別は、法人（'C'）と、非法人（'N'）だけである。このテーブルから、次の条件のいずれかを満たす行を返すクエリを作りたい。

条件1　ある企業内で1つしかオフィスが存在せず、そのオフィスは非法人である。

条件2　ある企業内で複数のオフィスが存在し、少なくともそのうち1つは法人である。

サンプルデータは表6.2のとおりである。最終的に取得したい結果に矢印を付けている。

表6.2　オフィスとDBA

duns	office_id	office_type	dba_name
132345678	1302	N	Fred
132345678	1303	N	Sam
132345678	1306	N	Mary
132400000	304	C	Bill ⬅条件2
132400000	305	N	Melvin ⬅条件2
132456885	9907	N	Ned ⬅条件1
139824328	2001	C	Irving

6.7.1　手続き的分解による解法

以下に示すのは、トム・クーパーの考えた解を書き直したものだ。これはまさに分解／統合による解法だ。

```
SELECT O.duns,
       MAX(O.office_id) AS office_id,
       MAX(O.office_type) AS office_type,
       MAX(O.dba_name) AS dba_name
  FROM Offices AS O
 GROUP BY O.duns
HAVING COUNT(*) = 1
   AND MAX(O.office_type) = 'N'
UNION ALL
SELECT O1.duns, O1.office_id, O1.office_type, O1.dba_name
  FROM Offices AS O1
 WHERE EXISTS
        (SELECT O2.duns
           FROM Offices AS O2
          WHERE O1.duns = O2.duns
          GROUP BY O2.duns
         HAVING MAX(O2.office_type) <> MIN(O2.office_type));
```

このクエリは、UNION ALLの前後で2つのパートに分けられる。最初のパートはこうだ。

```
SELECT O.duns,
       MAX(O.office_id) AS office_id,
       MAX(O.office_type) AS office_type,
       MAX(O.dba_name) AS dba_name
  FROM Offices AS O
 GROUP BY O.duns
HAVING COUNT(*) = 1
   AND MAX(O.office_type) = 'N'
```

これは条件1 を満たす行を返す。`HAVING COUNT(*)= 1` の条件は、1 つの企業につきオフィスが1 つしかないという条件を保証している。`duns`をキーにグループ化すると、他の列をそのまま`SELECT`句には書けないので、ダミーの集約関数を使う必要がある。これが`MAX`関数を使っている理由だ。`HAVING` 句の条件によって、オフィスは各グループに1 行しかないことが保証されているので、`MAX(office_id)`は`office_id`と同じ値になる。同じことが、

office_type列とdba_name列にも当てはまる。こうして、企業について1 行のみを選択することができる。しかし、我々が欲しいのは(office_type ='N')の行だけなので、HAVING 句の(AND MAX(O.office_type)='N')でその条件を記述している。

一方、UNION ALLのあとのクエリは、条件2に該当する。

```
SELECT O1.duns, O1.office_id, O1.office_type, O1.dba_name
  FROM Offices AS O1
 WHERE EXISTS (SELECT O2.duns
                 FROM Offices AS O2
                WHERE O1.duns = O2.duns
                GROUP BY O2.duns
               HAVING MAX(O2.office_type) <> MIN(O2.office_type));
```

こちらのパートは、少なくとも一企業について2 行以上持ち、それらの行のoffice_typeの値が異なる（つまり、最大値と最小値が異なる）行を選択している。だが、これは厳密に見ると条件2を正しく実装していない。というのも、これはすべてのオフィスが法人のケースを選択できないからだ（「少なくとも1つは法人」という条件は、すべてが法人のケースも含む）。

6.7.2　論理的分解による解法

このクエリの最も重要なパートは、条件を満たす会社のDUNSを取得することだ。それさえわかれば、会社に紐付けられた情報はすべてわかる。

クエリのスケルトンは次のような形である。

```
SELECT O1.duns, O1.office_id, O1.office_type, O1.dba_name
  FROM Offices AS O1
 WHERE O1.duns IN (<条件を満たすduns>);
```

我々は、同じ抽象レベルにおける特殊例の代わりに、入れ子のトップダウンアプローチを使っている。条件を満たすDUNSを見つけるという問題の本質は、2つのパートに分解して、それらをもう一度統合することにはない。この問題はまた、比較的小さいサンプルでもある。これが集約を使う問題であることは明らかなので、GROUP BYかPARTITION BYをサブ問題のどこかで使うことになる。GROUP BYを使うなら、スケルトンは次のようになる。

```
<qualified duns> ::=
SELECT O2.duns
  FROM Offices AS O2
 GROUP BY duns
HAVING (<条件1>)
    OR (<条件2>)
```

<条件1>に着目しよう。この条件というのは、行数が1で、オフィス種別が'N'ということだ。すなわち、結果は常に1 行になるということなので、この条件は、最小値と最大値が一致する、ということだ。この条件をスケルトンに肉付けするとこうなる。

```
<条件1> ::=
SELECT O2.duns
  FROM Offices AS O2
 GROUP BY duns
HAVING (COUNT(*) = 1 AND MIN(O2.office_type)= 'N')  --条件1
    OR (<条件2>)
```

今度は<条件2>を見よう。今度は、行数は2以上で、オフィス種別の最小値は'C'である（いまは'C'がoffice_type の取りうる値の集合の中で最小値なので）。この条件は、単純にMIN 関数を使って記述できるが、もし条件に指定する値が最小値ではない場合は、

MIN(CASE WHEN office_type =<target value> THEN 0 ELSE 1 END)のようにCASE式を使えば一般的に対応できる（このような式を特性関数と呼ぶ。詳しくは第18章で解説する）。これで、スケルトンは完全に血肉を備えた答えになる。

```
SELECT O1.duns, O1.office_id, O1.office_type, O1.dba_name
  FROM Offices AS O1
 WHERE O1.duns IN (SELECT O2.duns
                     FROM Offices AS O2
                    GROUP BY duns
                   HAVING (COUNT(*) = 1 AND MIN(O2.office_type)= 'N')
                       OR (COUNT(*) > 1 AND MIN(O2.office_type)= 'C'));
```

このクエリはもはやUNION ALLを必要としないことに注目してほしい。当然、パフォーマンスも改善されるだろう。

【訳注1】 D.W.グリフィス（1875–1948）は米国の映画監督。さまざまな撮影技法を生み出したことから、「映画の父」と呼ばれる。代表作に『國民の創生』、『イントレランス』など。

【訳注2】 インデックスオンリースキャンと呼ばれる機能である。第5章の訳注7を参照。

【訳注3】 ジップの法則は、出現頻度がn 番目に大きい要素が、全体に占める割合が1/nに比例するという経験則。もともとは、米国の言語学者G. K.ジップが、文章中に登場する英単語の頻度に偏りがあることから発見した。さまざまな分野でこの法則が当てはまる現象が確認されているが、なぜこの法則が成立するのか、確たる理論的説明はない。

また、「馬を探せ、シマウマを探すな」とは、ある症状を示す病気として、ありふれた（風邪のような）病気と、非常に稀な奇病が考えられた場合、まずはありふれた病気から疑え、という医療現場の知恵を表現した原則。

【訳注4】　コントロールデータコーポレーション（CDC）は、スーパーコンピュータで有名な企業だった。Cyberシリーズは1970年代の同社の主力製品。

【訳注5】　チェビシェフ多項式は、ロシアの数学者パフヌティ・チェビシェフ（1821–1894）の考えた式で、関数を特定の区間内で近似するために使われる。

【訳注6】　閉式（closed form）とは、無限級数や極限などを含まず、基本的な算術関数の有限解の適用で表現できる式のこと。

【訳注7】　Oracleなどでは`CHECK`制約の中でユーザー定義関数を呼び出すことはできない。

7

手続き型の遺産

Procedural Constructs

SQL は宣言型言語だが、手続き型の機能も備えている。ストアドプロシージャ、トリガー、カーソルである。ストアドプロシージャ、より正確にはPersistent Stored Modules（SQL/PSM）は、SQLを拡張し、独自の4GLプログラミング言語を付け加えたものだ。トリガーは1つのテーブルに結び付けられたストアドプロシージャであり、挿入、更新、削除といったデータベースイベントによって制御される。カーソルは、検索、更新、削除のSQL 文をシーケンシャルなファイル構造に変換することで、一度に1行ずつアクセスすることができるようになる。

いまからそれぞれについて詳しく見ていくが、あらかじめ言っておくと3 つのうちで特に有用なのはストアドプロシージャである。トリガーが存在しているのは、昔の標準SQL が参照整合性制約による更新機能を持っていなかったからであり、徐々に不要になりつつある。後に、DB2 for z/OSのようにトリガーを実装するより先に参照整合性制約をサポートするデータベース製品も登場した。トリガーは極力使うべきではないし、使うのは、参照整合性制約では実現できない機能を補う目的に限定すべきである。カーソルの存在意義は、シーケンシャルなファイル構造を持つアプリケーションプログラミング言語とSQLのインターフェイスを作ることにあるが、今日では、

そのようなインターフェイスのためのメカニズムはほかにもあるし、適切にスキーマ設計を行えば、カーソルを使う必要はない。今日もしカーソルを見る機会があるとすれば、レガシーなCOBOLプログラムに埋め込まれているものだろう。

7.1

CREATE PROCEDURE 文

SQL のANSI/ISO 標準における4GLプログラミング言語は、SQL/PSM（Persistent Stored Modules：永続格納モジュール）である。定義文書は、「ISO/IEC 9075-4:2003, Information technology -- Database languages -- SQL -- Part 4: Persistent Stored Modules（SQL/PSM）」だ。

一応標準はあるものの、多くのSQL 製品は、独自の手続き型言語を持っており、詳細はベンダーのドキュメントを確認する必要がある。たとえば、OracleはPL/SQLという名前のプロシージャを持っている。これはSQL/PSMに非常に近い。DB2はPL/SQLとSQL/PSMの両方をサポートしている。

T-SQLはMicrosoft SQL ServerとSybaseが持つ独自の4GL言語だ。これは1回の処理でコンパイル可能なシンプルなコンパイラを備えており、もともとアプリケーション開発の作業に使うことを意図していなかった。もともとは、SQL Serverを最初に実装した16ビットUNIXのC 言語コンパイラである。

Informix 4GL は、Informix のための独自のアプリケーション言語である。Algol をもとにしており、多くのメインフレームのソフトウェアパッケージで使われている。

SQL/PSMはプログラミング言語ADAをもとにしているが、より現代的な、構造化されたプログラミング言語に近い。そのモジュールは、SQL/PSMコンパイラにコードについて教える情報から始まり、本体には、戻り値を戻す`return` 文を書くことができる。また、ブロックは入れ子で記述できる。

ブロックの先頭ではローカル変数を宣言し、真ん中にコードを記述し、最後にエラーハンドラを書く。ローカル変数は、通常のSQLのデータ型を使うが、注意が必要なのは、スキーマオブジェクトをSQL/PSMの中で作ることはできないことだ。制御構造の記述は、ブロックを入れ子にしたり（`BEGIN-END`と`BEGIN ATOMIC-END`）、分岐（`IF-THEN-ELSE`、`CASE`）、ループ（`WHILE` など）を使って行う。

エラーハンドラは割り込み駆動（interrupt-driven）モデルで動く。ブロック内のどこであれ例外が発生すると、制御は適切なエラーハンドラにジャンプする。本書でも簡単なSQL/PSMプロシージャを使うが、それ自身の解説は本書の目的ではないので、詳しくは専門の書籍を読んでほしい。

7.2

CREATE TRIGGER 文

1 つのトリガーは1 つの基底テーブルに属している。トリガーはスキーマ内で「データベースイベント」が起きたときに実行されるプロシージャである。トリガーイベントは、テーブルの行に対する削除、行の挿入、あるいは行に対する更新である。

トリガーの動作が起きるタイミングは、2つ選べる。トリガーイベントが起きる直前か、起きた直後である。これは、BEFOREまたはAFTERオプションによって指定する。

トリガーは動作の種類によって、削除トリガー、挿入トリガー、更新トリガーの3つに分類される。トリガーの動作にはINSERT INTO、UPDATE、DELETE FROM、MERGEなどがあり、SQLそのものだ。参照整合性を維持するためのアクション、たとえばCASCADE、SET NULL、SET DEFAULT もまたトリガーが起動するイベントになりうる。

もしCASCADEによってトリガーが起動してほしくなければ、オプションを指定して起動しないようにすることもできる。トリガーによって削除、挿入、更新される対象のテーブルは、遷移表（transition table）と呼ばれる。削除トリガーの場合、遷移表はOLDで指定する1 つしかない。挿入トリガーの場合も遷移表は1 つだが、こちらはNEW で指定する。更新トリガーでは、OLDとNEWの両方のテーブルが存在する（どちらも行数は同じである）。

トリガーの動作を概念的なモデルとして説明すると、OLDの行が集合として削除され、NEWの行が集合として挿入されるイメージだ。こ

うした遷移表に対して、REFERENCING句で別名を指定することもできる。

もしこれがAFTERトリガーであれば、イベントが起きた直後に実行されるし、BEFOREトリガーなら、イベントの直前に実行される。なおBEFOREトリガーには、INSTEAD OFトリガーという特殊なものがある。これは、VIEWと一緒に使い、基底テーブルへの更新イベントは見ない。このトリガーは、ビューが作られるもとになった基底テーブルに対して作用するコードを記述する。これによって、ユーザーから見ると、ビューがあたかも基底テーブルであるかのように見える。

INSTEAD OFトリガーの簡単な例を見せよう。国別の都市名を表示するGeographyビューを考える。これは結合を使っているため、更新不可能なビューである。

```
CREATE TABLE Countries
( country_code CHAR(2) NOT NULL PRIMARY KEY,
  country_name VARCHAR(64));
```

Countries：国テーブル　　country_code：国コード　　country_name:国名

```
CREATE TABLE Cities
( country_code CHAR(2) NOT NULL,
  city_code CHAR(2) NOT NULL,
  city_name VARCHAR(64),
     PRIMARY KEY(country_code, city_code));
```

Cities：都市テーブル　　country_code：国コード　　city_code：都市コード
city_name：都市名

```
CREATE VIEW Geography (country_name, city_name)
AS
SELECT country_name, city_name
  FROM Countries, Cities
 WHERE Countries.country_code = Cities.country_code;
```

以下のINSTEAD OFトリガーは、Geographyビュー経由で、基底となっている2つのテーブルCountriesとCitiesに行を挿入する。これによって、あたかもユーザーはビューを更新しているかのように錯覚する【訳注1】。

```
CREATE TRIGGER Add_New_Town
INSTEAD OF INSERT
ON Geography
REFERENCING NEW AS CP -- Current_Places
BEGIN
    INSERT INTO Countries (country_name)
    SELECT DISTINCT CP.country_name
     FROM CP
     LEFT OUTER JOIN
     Countries AS C
     ON CP.country_name = C.country_name
     WHERE C.country_name IS NULL; -- すでにテーブルに存在する国は除外

    INSERT INTO Cities (city_name, country_code)
    SELECT DISTINCT CP.city_name, Countries.country_code
     FROM CP
     INNER JOIN
     Countries AS C
     ON CP.country_name = C.country_name
     LEFT OUTER JOIN
     Cities
     ON CP.city_name = Cities.city_name
     WHERE Cities.city_name IS NULL; -- すでにテーブルに存在する都市は除外
END;
```

トリガーには、行レベルと文レベルの2 種類がある。文レベルのトリガーは、テーブル全体がアクションの対象となる。行レベルのトリガーは、アクションの影響を受ける行に限定される。

トリガーの参照整合性制約に対する利点としては、参照整合性制約ができることはトリガーでもできるし、参照整合性制約にはできないこともトリガーならできることだ。欠点は、オプティマイザは手続き型のコードからは何の情報も引き出せないため、最適化の恩恵を受けられず、トリガーの処理時間は遅いことだ。処理をシーケンシャルにしか実施できず、パラレル化できないこともパフォーマンス上の欠点だ。しかも実装依存なので移植性がない。

私からのアドバイスは、何度も言っているが、トリガーを避けて、参照整合性制約を使うことだ。どうしてもトリガーを使わざるを得ないときは、コードを注意深くチェックして、パフォーマンスを悪くさせないよう簡単な処理を行うにとどめてほしい【訳注2】。

7.3

カーソル

本書ではカーソルについてはあまり取り上げない。だが、カーソルが何かについてはよく理解しておいてほしい。というのも、実際のコードにおいて頻繁に見られるからだ。カーソルにも標準構文はあるのだが、例によって製品ごとに構文は不統一である。これはカーソルが低レベルの概念で、製品の物理的実装に近いレベルで動作するため、どうしても物理的制約の影響を受けやすいからだ。

カーソルの役割は、SQLの結果セットを、ファイルのようなシーケンシャルなデータ構造（レコードが順序を持つ）に変換して、手続き型のホスト言語がデータを扱えるようにすることである。実際のところ、カーソルの動作は昔ながらの磁気テープそのものだ！読者はすでに気づいているかもしれないが、SQLにおいて「CREATE～」は、永続的なスキーマオブジェクトを作るときに使う。一方「DECLARE～」は、セッションの終了とともに消える一時的なオブジェクトを作るときに使う。これが、DECLARE CURSOR文はあってもCREATE CURSOR文はない理由だ。

カーソルを使うには、まずホスト言語のBEGIN DECLARE...END DECLAREセクションで囲まれたコードが動作するためのワーク領域を割り当てる。この領域は、SQL 変数をホスト言語のデータ型に変換したり、その逆をするときに使われる。NULLを扱うには、ホスト言語のBEGIN DECLAREセクションでINDICATOR変数を宣言する。この変数はホスト言語側の適切な変数と対応している必要がある。インジケータ（INDICATOR）は整数のデータ型であり、ホスト言語ではまさに一種の整数値として扱われる。

7.3.1 DECLARE CURSOR 文

標準SQLにおけるDECLARE CURSOR 文は次のとおりである。SQL-92 構文は、実際の製品で使われる構文にかなり近い。近いのだが、正確なところは製品のマニュアルを調べる必要がある。

```
<declare cursor> ::=
DECLARE <cursor name> [INSENSITIVE] [SCROLL] CURSOR
FOR <cursor specification>
<cursor specification> ::=
<query expression> [<order by clause>]
[<updatability clause>]
<updatability clause> ::= FOR {READ ONLY | UPDATE [OF
<column name list>]}
<order by clause> ::= ORDER BY <sort specification list>
<sort specification list> ::=
<sort specification> [{<comma> <sort specification>} ...]
<sort specification> ::= <sort key> [<collate clause>]
[<ordering specification>]
<sort key> ::= <column name>
<ordering specification> ::= ASC | DESC
```

この構文について少し説明しておきたい。まず第1に、ORDER BY 句をSELECT 文の一部だと思っている人もいるかもしれないが、実際はカーソルの一部である。SQL Server やSybase のような製品では、ユーザーが暗黙にカーソルを作ることができるため、多くの新米プログラマがこの点を誤解する。シーケンシャルなファイルシステムから進化したDBMS 製品にとっては、そのほうが実装しやすく、いまだにそのアーキテクチャをユーザーに使わせている。だがもちろん、これはコッドのルール（第9章参照）に違反している。

INSENSITIVE、SCROLL、ORDER BY のいずれかが指定されているか、ワークテーブルが読み取り専用の場合、<updatability clause>は暗黙にREAD ONLYとされる。そうでない場合は、

<updatability clause>は<column name list>なしのFOR UPDATEが暗黙に前提される。

7.3.2 ORDER BY 句

ORDER BY 句は一般にSELECT文の一部だと信じられている。だが実際には、これはカーソル定義の一部である。これがSELECT文の一部であるように思える理由は、クエリの結果セットをホスト言語に渡す唯一の方法が、カーソル経由だからである。ベンダーツールが裏でカーソルを使うときは、クエリにORDER BY 句を含められるのが普通だ。たいていのオプティマイザは、クエリの必要とする結果が、ソート済みの状態でフェッチされていないか調べる。もし見つかれば、ソート処理を二重に行うことを避けられるからだ。

残念なことに、多くのプログラマは特定のSQL製品の特定のバージョンが結果を表現する方法に依存するコードを記述している。もしインデックスが削除されたり定義を変更された場合や、データベースが新しいバージョンにアップグレードしたり、異なる製品に移行したりした場合、自動的なソートは行われなくなる可能性がある【訳注3】。

カーソルの一部として、ORDER BY 句は読者の知らないいくつかのプロパティを持っている。ORDER BY句の標準構文はこうである。

```
<order by clause> ::=
ORDER BY <sort specification list>
<sort specification list> ::=
<sort specification> [{ <comma> <sort specification> } ...]
<sort specification> ::=
<sort key> [<collate clause >] [<ordering specification>]
<sort key> ::= <column name> | <scalar expression>
<ordering specification> ::= [ASC | DESC] {NULLS FIRST | NULLS LAST}
```

最初に注目すべきは、ソートキーはSELECT句に現れる列名を使う必要があることだ。列位置の番号の使用は標準SQLから削除が予定されている機能なので、使うべきではない。

次に、違法なコードの例をお見せしよう。

```
SELECT a, (b+c) AS d
  FROM Foobar
 ORDER BY a, b, c; -- 違法!!
```

列bとcはSELECT句に存在せず、カーソルの結果には含まれていない。それゆえこの列を使ったソートはできない。ただし、SQL-99ではORDER BY句に計算列を含むことができるようになった。

```
SELECT a, b, c -- 違法!!
  FROM Foobar
 ORDER BY a, b, (b+c)
```

上のコードを正しくするには、SELECT 句にも(b+c)という式を追加して、その式に列名を与え、その列名をORDER BY 句で使うことだ。これによってユーザーは、どのような値でソートが行われているかがわかる。レポートや帳票がどんな順序でソートされているかわからないというのは恐ろしいことではないだろうか？

さらに、ソートキーの列は中間層（アプリケーションサーバーなど）に情報を渡すことで、フロントエンドのクライアントに結果が返される前にデータを再ソートするのにも利用できる。

ソート順序は、列の定義に付属している照合順序に基づいて決められる。照合は文字列の列に対してスキーマ内で定義される。

ORDER BY とNULL

もしソートキーにNULLが含まれていたら、NULLは他の非NULL値と比べて大きいのだろうか小さいのだろうか。この扱いはSQL製品によって異なる。次のようになっている。

- PostgreSQL ：大きい
- DB2 ：大きい
- SQL Server ：小さい
- MySQL ：小さい
- Oracle ：大きい

SQL-99 はこうした状況に対応するため、<ordering specification>句に{NULLS FIRST | NULLS LAST}というオプションを追加した。これはもともと、DB2とOracleに実装されていた機能だ。ここには1つエピソードがある。1999年5月、クリス・ファーラーはある疑問を提示した。彼は、一般的に理解されている常識と、実際のSQL標準の仕様書の記述との間に差異があることに気づいたのだ。たとえば、Sortableというテーブルを考える。このテーブルには2 つの整数型の列alpha、betaがあり、2 行のデータが含ま

れている。その物理的な格納順序は、たまたまalpha、betaの昇順である。

ここで、次のような擬似的なクエリを考える。

```
SELECT alpha, beta
  FROM Sortable
 ORDER BY alpha, beta;
```

最初の疑問は、これはカーソルが表7.1に示す順序の結果を生成しうる適法なSQLだろうか、というものだ。betaの値に注目してほしい。昇順にソートしたならば、むしろ4が先に来るべきだが、それはalpha列のNULLを互いに等しいとみなした場合の話である。

表7.1 カーソル結果順序**バージョン1**

alpha	beta
NULL	8
NULL	4

問題は、標準SQL のルールでは、NULLはまとめて非NULL 値よりも大きいか小さい値として扱われることになっているが、NULL同士が等しい値と解釈されるべきかどうかは決められていない、という点だった。ORDER BY 句でNULLとNULLを比較したとき、NULLとNULLが等しいかどうかは、標準SQLでは言及されていないのだ（GROUP BY句でも同じだ）。

SQL標準化委員会で得られた合意は、NULLとNULLは等しいとみなすべきだ、というものだった。つまり、期待される結果の表示順序は表7.2である。

表7.2　カーソル結果順序バージョン2

alpha	beta
NULL	4
NULL	8

だが、委員会の初代代表であったIBMのフィル・ショウは、SQL-89のドキュメントを掘り返して、この問題に対する答えを見つけた。SQL-89では、ORDER BY 句での比較ルールは以下のようになっていた。

> 「x = y」の値は、xとy がともにNULLの場合は知りえないが、GROUP BY、ORDER BY、DISTINCTにおいては、2つのNULLは互いに等しく、重複しているとみなされる。

これがいわゆる「集約対同値性」問題である。すべてのNULLは、GROUP BYなどにおいて同じグループにまとめられる。このルールをORDER BYに適用すると、2行のalpha列はともにNULL なので、同値とみなされる。したがって、ソートはbeta 列の値を使って行われる。ソートキーはリストの左から右に適用されるからだ。このよう

なキー順序を保存するソートを安定ソートと呼ぶが、SQL の側に絶対に安定ソートを使わなければならない、という強制があるわけではない。安定ソートを使えば、たとえば、県でソートされたあとに市町村でソートすることになり、1 つの州の中で市町村が並ぶことになる。安定性は便利な性質だが、非安定的なソートのほうが一般にはかなり速い（安定ソートについては第28章も参照）。

　また、計算列を使うことで、特殊なソート順序を利用することができる。たとえば、文字列型の列と整数型の列でテーブルを作るとする。やりたいことは、結果をまず整数列の降順でソートし、次に文字列の列でグループ化することだ。サンプルを見たほうがわかりやすいだろう。

```
CREATE TABLE Fruit_Test
(fruit_name CHAR(10) NOT NULL,
 taste_score INTEGER NOT NULL,
     PRIMARY KEY (fruit_name, taste_score));
```

Fruit_Test：フルーツテーブル　　fruit_name：フルーツ名　　taste_score：美味しさ点数

```
INSERT INTO Fruit_Test
VALUES ('Apples', 2), ('Apples', 1), ('Oranges', 5), ('Apples', 5), ('Banana', 2);
```

　いま、次のような順序でソートしたいとする。これは単に（「美味しさ点数」の降順、「フルーツ名」の昇順）でソートしたわけではない。点数の高い行を持つフルーツのグループ（この例だと'Apples'）は、点数の低いものもまとめて上位に表示することがポイントだ。

```
('Apples', 5)
('Apples', 2)
('Apples', 1)
('Oranges', 5)
('Banana', 2)
```

まずは整数列の降順でソートしてみる。

```
SELECT F1.fruit_name, F1.taste_score
  FROM Fruit_Test AS F1
 ORDER BY F1.taste_score DESC;
```

得られる結果は次のうちのいずれかだ。

結果1

```
('Apples', 5)
('Oranges', 5)
('Apples', 2)
('Banana', 2)
('Apples', 1)
```

結果2

```
('Oranges', 5)
('Apples', 5)
('Apples', 2)
('Banana', 2)
('Apples', 1)
```

結果3

```
('Oranges', 5)
('Apples', 5)
('Banana', 2)
('Apples', 2)
('Apples', 1)
```

結果4

```
('Apples', 5)
('Oranges', 5)
('Banana', 2)
('Apples', 2)
('Apples', 1)
```

　これはフルーツの順序が固定されていないので、まだ完全な結果ではない。ではフルーツ名の順序も指定した次のクエリはどうなるだろう。

```
SELECT F1.fruit_name, F1.taste_score
  FROM Fruit_Test AS F1
 ORDER BY F1.taste_score DESC, F1.fruit_name ASC;
```

結果

```
('Apples', 5)
('Oranges', 5)
('Apples', 2)
('Banana', 2)
('Apples', 1)
```

　これでたしかに順序は固定されたが、我々の欲しかった結果ではない――フルーツのグルーピングが崩れてしまったからだ。ではフルーツで先にソートするのはどうか。

```
SELECT F1.fruit_name, F1.taste_score
  FROM Fruit_Test AS F1
 ORDER BY F1.fruit_name ASC, F1.taste_score DESC;
```

結果

```
('Apples', 5)
('Apples', 2)
('Apples', 1)
('Banana', 2)
('Oranges', 5)
```

これもまだうまくない――得点の順序が崩れているからだ（オレンジの5 点のほうが高いのに、2 点のバナナがオレンジより上に来ている。アルファベット順ではBはOより前だからだ）。この問題を解くには、順序を保存するためのダミー列が必要なのだ。

```
SELECT F1.fruit_name, F1.taste_score,
       (SELECT MAX(taste_score)
          FROM Fruit_Test AS F2
         WHERE F1.fruit_name = F2.fruit_name) AS taste_score_preserver
  FROM Fruit_Test AS F1
 ORDER BY taste_score_preserver DESC, F1.fruit_name ASC, F1.taste_score DESC;
```

これで常に欲しかった結果が得られる。

ORDER BY とCASE 式

SQL-99 では、ORDER BY 句で関数を使うことができる。これはいまでは適法な構文だが、それでもあまり推奨できるコーディングではない。なぜなら、ユーザーは帳票のデータがどのような順序でソートされているかを理解するため、ソートキーを把握できるべきであるのに、ORDER BY 句で関数を使うと、どんな順序でソートされているのか、ユーザーから隠すことになるからだ。

だが、どうしてもORDER BY 句で加工した列をキーに使いたいならば、最も汎用的な方法はCASE式を使うことだ。

```
CREATE TABLE Personnel
( first_name VARCHAR(8) NOT NULL,
  last_name  VARCHAR(8) NOT NULL,
  dept_name  VARCHAR(8) NOT NULL,
    PRIMARY KEY(first_name, last_name));
```

Personnel：社員テーブル　first_name：名　last_name：姓　dept_name：部署名

```
SELECT first_name, last_name, dept_name,
       CASE :flag
       WHEN 'f' THEN first_name
       WHEN 'l' THEN last_name
       WHEN 'd' THEN dept_name
       ELSE NULL END AS sort_col
  FROM Personnel
 ORDER BY sort_col;
```

当然のことだが、THEN句の式は、同じデータ型か、CASTで同じデータ型に変換しておく必要がある。ソートキーの順番を制御したいなら、少しトリッキーだが、2 列用意して、片方の列は常にNULLにしておくという方法がある。

```
SELECT last_name,
       CASE :flag
       WHEN 'la' THEN last_name ELSE NULL END AS sort_col1,
       CASE :flag
       WHEN 'ld' THEN last_name ELSE NULL END AS sort_col2
  FROM Personnel
 ORDER BY sort_col1, sort_col2 DESC;
```

こうすると、sort_col1とsort_col2のどちらかは必ずNULLになる。

このアイデアを一般化すると次のようなコードが得られる。

```
SELECT ...
       CASE :flag_1
       WHEN 'a' THEN CAST (a AS CHAR(n))
       WHEN 'b' THEN CAST (b AS CHAR(n))
       WHEN 'c' THEN CAST (c AS CHAR(n))
       ELSE NULL END AS sort_1,
       CASE :flag_2
       WHEN 'x' THEN CAST (x AS CHAR(n))
       WHEN 'y' THEN CAST (y AS CHAR(n))
       WHEN 'z' THEN CAST (z AS CHAR(n))
       ELSE NULL END AS sort_2,
       ...
       CASE :flag_n
       WHEN 'n1' THEN CAST (n1 AS CHAR(n))
       WHEN 'n2' THEN CAST (n2 AS CHAR(n))
       WHEN 'n3' THEN CAST (n3 AS CHAR(n))
       ELSE NULL END AS sort_n
  FROM MyTable
 WHERE ...
 ORDER BY sort_1, sort_2, ...;
```

もしソートキーが2つ以上あって、かつ限られた組み合わせしかない場合は、ソートキーを結合する方法もある。

```
CASE :flag_1
WHEN 'ab'
THEN CAST(a AS CHAR(n)) ||' ' || CAST(b AS CHAR(n))
WHEN 'ba'
THEN CAST(b AS CHAR(n)) ||' ' || CAST(a AS CHAR(n))
ELSE NULL END AS sort_1,
```

ASCとDESCオプションが必要ならば、次のように記述すればよい。

```
CASE and ORDER BY:
CASE :flag_1
WHEN :flag_1 = 'a' AND :flag_1_ad = 'ASC'
THEN CAST (a AS CHAR(n))
WHEN :flag_1 = 'b' AND :flag_1_ad = 'ASC'
THEN CAST (b AS CHAR(n))
WHEN :flag_1 = 'c' AND :flag_1_ad = 'ASC'
THEN CAST (c AS CHAR(n))
ELSE NULL END AS sort_1_a,
CASE :flag_1
WHEN :flag_1 = 'a' AND :flag_1_ad = 'DESC'
THEN CAST (a AS CHAR(n))
WHEN :flag_1 = 'b' AND :flag_1_ad = 'DESC'
THEN CAST (b AS CHAR(n))
WHEN :flag_1 = 'c' AND :flag_1_ad = 'DESC'
THEN CAST (c AS CHAR(n))
ELSE NULL END AS sort_1_d
...
ORDER BY sort_1_a ASC, sort_1_d DESC
```

このサンプルでは、一応、明示的にCAST (<exp> AS CHAR(n))関数を使っているが、THEN 句の返す値が必ず同じデータ型であると保証されていれば、型変換を行う必要はない。

ELSE NULL 句は、NULLの代わりに任意のデータ型を返すようにすることもできるが、読み手にとって有用なものである必要がある。もっとうまくするには、個々のソートオプションにつき1つの列を使うことだ。

```
SELECT ...,
       CASE WHEN :flag = 'a' THEN a ELSE NULL END AS sort1,
       CASE WHEN :flag = 'b' THEN b ELSE NULL END AS sort2,
       CASE WHEN :flag = 'c' THEN c ELSE NULL END AS sort3
  FROM MyTable
 WHERE ...
 ORDER BY sort1, sort2, sort3;
```

このコードは読みやすく、型変換について心配する必要もない。その代わり、カーソルに渡す結果セットのサイズは大きくなる。

7.3.3　OPEN 文

OPEN <cursor name>文は、カーソルの最初のレコードがアクセスされる前の、仮想的な読み書きヘッドの位置を決める。この仮想的な読み書きヘッドは、FETCH文によってレコードからレコードへ動く。ヘッドが最後のレコードを通過すると、磁気テープのファイルシステムにおけるEOFに相当する例外が発生する。

このモデルをよく見てほしい。ヘッドが最初のレコードからスタートし、最後のレコードを読んだところでEOFフラグがTRUEにセットされる。これはファイルシステムと同じモデルであることに気づくだろう。といっても、手続き型のコードから単純にアルゴリズムをコピーしただけでうまくいくわけでもないのだが。

7.3.4　FETCH 文

FETCH文はカーソルから一行を取り出し、SQLのデータ型をホスト言語のデータ型に変化し、結果を適切なホスト変数に入れる。もしSQLの値がNULLであれば、インジケータは–1がセットされる。インジケータが指定されていない場合は、例外が発生する。

```
<fetch statement> ::= FETCH [[<fetch orientation>] FROM] <cursor name>
  INTO <fetch target list>
<fetch orientation> ::= NEXT | PRIOR | FIRST | LAST | {ABSOLUTE | RELATIVE}
  <simple value specification>
```

ホストプログラムは必ずインジケータをチェックする必要がある。なぜなら、そうしないとパラメータの値が不正なゴミになってしまう可能性があるからだ。もしパラメータが問題なくホスト言語に受け渡されれば、インジケータにはゼロがセットされる。ホスト

プログラムに渡された値が非NULLの文字列で、インジケータを持っていれば、インジケータにはSQL文字列長がセットされる。この値は、文字列のオーバーフローのチェックに使われる。

<fetch orientation>は、ヘッドがどちらの方向に動くかを指定する。NEXTとPRIORは、現在の位置から前に動くか後ろに動くかを指定する。FIRSTとLASTは、ヘッドを最初または最後のレコードのどちらかに移動させる。ABSOLUTEを指定すると、フェッチ動作は指定されたレコード番号へピンポイントで移動する。RELATIVEフェッチはヘッドを現在の位置からn 行前または後に移動させる。

いかがだろう、カーソルの動作がファイルシステムをそっくり真似したものだ、ということがおわかりいただけたのではないだろうか。

7.3.5　CLOSE 文

CLOSE <cursor name>文は、カーソルのヘッドをカーソルの最初の行の前の位置へリセットする。カーソル自身はまだ存在しているが、再び使えるようにするためには、再オープンしなければならない。これはFORTRAN やCOBOLにおけるCLOSE FILE 文に似ているが、1 つ重要な違いがある。カーソルは、もう一度開くときにクエリが再実行される可能性があることだ。

7.3.6　DEALLOCATE 文

DEALLOCATE CURSOR文はホストプログラムにおけるワーク領域を解放する。ファイルシステムにたとえるなら、テープドライブからテープをディスマウントすることに相当する。

7.3.7　カーソルの使い方

データベースにおいて、カーソルのパフォーマンスを改善する最高の方法を教えよう。カーソルを使わないことだ。SQL エンジンは集合操作を行うように設計されており、個々の行を扱うよりも行の集合をひとまとまりに扱うほうが得意だ。どうしてもカーソルを使わねばならないときがあるとすれば、不適切な設計によって引き起こされた不具合に修正を施す場合か、製品の物理的な実装特性によってカーソルによるアクセスのほうが速い場合である。たとえば、キーを持たないテーブルから冗長な重複行を選択するような場合だ。

SQLは集合指向であり、クエリは答えの全体集合を返そうとする。もし読者がNP 完全問題（巡回セールスマン、巨大な組み合わせ問題など）を扱っているなら、「満足できる最初の答え」を見つけた時点で答えを返してくれるコードが便利だろう。これは手続き型言語のバックトラックなどのテクニックだが、SQLでは実現できない。

かつてSybase SQL Serverのトレーニングコースで、カーソルの使用例として取り上げられていたのが、第6 章で論じた例であった。読者が書店を経営しているとして、本の値段を変えようと考えている。25ドル以上の本はすべて10%引きにし、25ドル未満の本は15%増しにする。

```
BEGIN
UPDATE Books
   SET price = price * 1.15
 WHERE price < 25.00;

UPDATE Books
   SET price = price * 0.9
 WHERE price >= 25.00;
END;
```

だがこれは、すでに見たようにうまくいかない。現在の価格が24ドルの本は、最初のUPDATE 文で27.6ドルに上昇し、2 番目のUPDATE 文で24.84ドルに下落する。そこで登場するのがカーソルだ。1行ずつテーブルをスキャンし、それぞれの行を変更する。だが今日ではこんなことをせずとも次のように書けばよい。

```
UPDATE Books
SET price = CASE WHEN price < $25.00;
                   THEN price * 1.15
                 WHEN price >= $25.00
                   THEN price * 0.90
            ELSE price END;
```

SELECT文のカーソルにおける行と列の数は、妥当な結果セットに必要な最小限に絞るべきだ。それによって、不必要なデータをフェッチすることがなくなり、サーバーのリソース消費も抑えられ、カーソルのパフォーマンスは向上する。特に、不要な列まで結果セットに含まないよう注意してほしい。安易に'SELECT *'など使ってはならない。

また、もし可能ならUPDATEカーソルではなくFOR READ ONLYカーソルを使うのがよい。ただし、トランザクション分離レベルはチェックしておく必要がある。多くのDBMSでは、INSENSIT IVE カー

ソルをオープンすることで、カーソルの持つ行をワークテーブルにコピーできるか、テーブルレベルでロックすることができる。

結果セットに対する処理が終わったあとは、できるだけ早くカーソルをクローズするべきである。これによって、行に対するすべてのロックが解放される。作業を終えたら必ずカーソルを解放することを常に忘れないでほしい。

その他に、製品固有のオプションも重要だ。たとえば、SQL ServerにはFAST_FORWARDおよびFORWARD_ONLYというオプションがある。これは読み取り専用の行に対する移動を一方向にだけ動かせるよう制限するオプションだ。つまりFAST_FORWARDは、FORWARD_ONLYかつREAD_ONLYという意味になる。これによってパフォーマンスを向上できる。

トランザクション内に含まれるカーソル経由で、ループによって多くの行を変更する場合は、注意が必要だ。トランザクション分離レベルにもよるが、こうした行はトランザクション終了までロックが保持されることもある。これは時としてリソース競合の原因となる。

SQL文が実行されると、エラー状態コードが自動的に生成される。このコードは成功、失敗、警告、データが見つからなかったといった情報を表す。このエラー状態コードは、SQLSTATEというビルトイン変数に格納される。

SQLSTATE コードは、5 桁の文字列で数字または大文字のアルファベットしか含められない。SQLSTATEコードの最初の2 文字はクラスを、残りの3 文字はサブクラスを表す。たとえば、値08001は、上

2 桁の08 はクラスコードで、下3 桁の001はサブクラスコードである。08001という値はデータベース環境に接続できなかったというエラーを表す。下表はSQLSTATEクラスコードの値と状態の一覧だ。

表7.3　SQLSTATEのクラスコードの値と状態

クラス	状態
00	成功
01	警告
02	データが見つからない
07	動的SQL エラー
08	接続例外
0A	サポートされていない機能
21	カーディナリティ違反
22	データ例外
23	整合性制約違反
24	不正なカーソル状態
25	不正なトランザクション状態
26	不正なSQL 文名

27	トリガーによるデータ変更違反
28	不正な認証仕様
2A	SQL構文エラーまたはアクセスルール違反
2B	依存する権限定義がまだ存在している
2C	不正なキャラクタセット名
2D	不正なトランザクション終了
2E	不正な接続名
33	不正なSQL記述子名
34	不正なカーソル名
35	不正な条件数
37	動的SQL構文エラーまたはアクセスルール違反
3C	あいまいなカーソル名
3D	不正なカタログ名
3F	不正なスキーマ名
40	トランザクションロールバック
42	構文エラーまたはアクセスルール違反
44	with checkオプション違反
HZ	リモートデータベースアクセス。クラスHZはISO/IEC DIS 9579-2のリモートデータベースアクセスのために予約されて

	いる。

それ以外にも、特別なコードとして、21000はカーディナリティ違反を意味する。GET DIAGN OSTICS文を実行することで、さらなる情報を得られるが、それで得られる警告メッセージは次のようなものだ。

```
01I09：射影リストとINTOリストのカーディナリティが不一致
01I06：SQL-92であらかじめ定義された、ANSI互換構文のベンダー拡張SQLSTATEクラス
```

7.3.8　位置付けられたUPDATE 文またはDELETE 文

当然ながら、行を指定したUPDATEまたはDELETEを行うには、カーソルは明示的または暗黙的に<updatability clause>にFOR UPDATE を指定する必要がある。そして、位置を指定する文と同じモジュールに存在しなければならない。もしREAD ONLYのカーソルを変更しようとしたり、指定された位置でないレコードを更新しようとしたときは、例外が発生する。

CURRENT OF <cursor name>句は、仮想的なヘッドが現在指しているレコードを参照する。このカーソルレコードは、基底テーブルの特定の1行にマッピングされなければならない。

UPDATE 文の定義

```
<update statement: positioned>
  ::= UPDATE <table name>
        SET <set clause list>
      WHERE CURRENT OF <cursor name>
```

たとえ更新によって何らかの例外が発生したとしても、カーソルは当該のカレント行の上にとどまり続ける。

DELETE 文の定義

```
<delete statement: positioned>
  ::= DELETE FROM <table name>
       WHERE CURRENT OF <cursor name>
```

もしカーソルがオープンされている間に、別のDELETE 文またはUPDATE 文が同じカレントレコードを更新しようとした場合は、カーソル操作の衝突を知らせる警告が発生する。どのような動作になるかはトランザクション分離レベル次第である。また、もし<delete statement:positioned>が最後のカーソルレコードを削除したならば、カーソルは最終行の後ろに移動する。それ以外の場合は、カーソルの位置は次のカーソルレコードの前に移動する。

7.4

シーケンス

シーケンスは、呼び出されるたびに連続した値を生み出すジェネレータである。呼び出すときは関数のように使うことができ、数列における次の値を得られる。

宣言の中では、開始値、増分、上限と下限の値を指定する。CYCLE句は、上限（または下限）に達したとき、もう一度開始値から処理を再開するかどうかを指定する。サンプルとしては次のようなものだ。

```
CREATE SEQUENCE part_num
START WITH 1
INCREMENT BY 1
MAXVALUE 100000
MINVALUE 1
CYCLE;
```

後ろ3 つのオプションは、NO MAXVALUE、NO MINVALUE、NO CYCLE で置き換えることもできる。DB2には、データベースサーバーの複数ノード間においても要求された順番で数値を生成するかどうかを決めるORDER またはNO ORDER という独自オプションがある（デフォルトはNO ORDER）。ORDERにすることでノード間の数値の連続性を保証できるが、オーバーヘッドが発生する。

上記サンプルでは、100, 000 に達するとシーケンスは1 から再開される。つまり、これはサイクルサイズを超えたら重複値が発生するということだ。

シーケンスを使うには、一般的な関数呼び出しではなく、NEXT VALUE FOR <sequence name>またはPREVIOUS VALUE FOR

<sequence name>という構文でアクセスする【訳注4】。もしNEXT VALUEがPREVIOUS VALUEと同じ文の中で使われた場合は、文の中で記述されている両者の順序にかかわらず、PREVIOUS VALUEが1つ前の（つまりインクリメントされていない）値を返し、NEXT VALUEが次の値を返す。サンプルは次のようになる。

```
SELECT NEXT VALUE FOR part_num,
       PREVIOUS VALUE FOR part_num,
...
```

シーケンスの定義と基本的には同じ宣言を、テーブル定義における列の動作を記述するために使うこともできる。これも説明するよりサンプルを見たほうが早い。

```
CREATE TABLE Personnel
(emp_id INTEGER
     GENERATED [ALWAYS | BY DEFAULT] AS IDENTITY
          START WITH 100
          INCREMENT 1
          MINVALUE 100
          NO MAXVALUE
          NO CYCLE,
 salary_amt DECIMAL(7,2));
```

Personnel：社員テーブル　emp_id：社員ID　salary_amt:給与額

GENERATED [ALWAYS | BY DEFAULT] AS IDENTITYパラメータは、列に常にシーケンスの数値を割り当てるか、それとも明示的に値を指定して上書きできるかを制御する。上書きできる場合は、シーケンスの数値はいわば列のデフォルト値という扱いになる。

シーケンスはグローバルに利用できる。これが、次に見る生成列のスコープがテーブルに閉じるのと違うところだ。

7.5

生成列

生成列は計算列や導出列とも呼ばれる。基底テーブルの中で宣言され、列単位でビューのような働きをする。生成列はスカラ式によって定義される。そうした式における列参照は、生成列を含む基底テーブルに含まれる列を対象とする必要がある。つまり、他のテーブルの列を参照することはできない。生成列の値は、行が挿入されるとき自動的に計算され、割り当てられる。

これはサンプルを見たほうが早いだろう。

```
CREATE TABLE Personnel
(emp_id INTEGER NOT NULL,
 salary_amt DECIMAL(7,2) NOT NULL,
 bonus_amt DECIMAL(7,2) NOT NULL,
 total_comp_amt GENERATED ALWAYS AS (salary_amt + bonus_amt),
 hr_clerk_user_id GENERATED ALWAYS AS (CURRENT_USER));
```

Personnel：社員テーブル　emp_id：社員ID　salary_amt：給与額　bonus_amt：ボーナス額　total_comp_amt：収入総額　hr_clerk_user_id：事務員ID

なお、生成された式をデフォルト値として使うことはできないので注意してほしい。

7.6

テーブル関数

テーブル関数はSQL、C 言語、Javaといった特定の言語の文に基づいて、テーブル（1 行以上の行）を返す。テーブル関数は、テーブルを使うことのできる場所ならばどこでも使えるが、常に（0個でもよいので）引数を必要とする。もしテーブル関数が引数を持たない場合、テーブル全体が返される。この場合のテーブル関数は、ビューや基底テーブルと同じである。ただ、テーブルを返すためにSQLを使うテーブル関数の構文は、単純なビューとは異なる。というのも、テーブル関数の本当の目的は、外的データをSQLに取り込むことだからだ。たとえば、次の構文は、`Sales`テーブルから集計された売上げの値を返すテーブル関数を定義する。

```
CREATE FUNCTION SalesSummary
(fiscal_period INTEGER, store_id INTEGER)
RETURNS
TABLE (fiscal_period INTEGER,
       store_id INTEGER,
       sales_amt DECIMAL (7,2))
LANGUAGE SQL
READS SQL DATA
DETERMINISTIC
RETURN
(SELECT fiscal_period, store_id, SUM(sales)
   FROM Sales
  GROUP BY fiscal_period, store_id);
```

このテーブル関数は、2つの引数を持っている。`fiscal_period`と`store_id`である。関数は`IN`引数しか持たないため、`INOUT`や`OUT`オプションはない。

```
SELECT T1.fiscal_period, T1.store_id, T1.sales_amt
  FROM Calendar AS C,
       Stores AS S,
       TABLE (SalesSummary(C.fiscal_period, S.store_id)) AS T1
 WHERE C.fiscal_period = T1.fiscal_period
   AND S.store_id = T1.store_id;
```

CalendarとStoresは、それぞれテーブル関数の返す行と、fiscal_periodとstore_idをキーに結合されている。これはいわば相関サブクエリを、違う構文で表現しているのである。

これはテーブル関数の使い方としては良いサンプルではない。経験の浅いSQLプログラマは、導出テーブルやビューよりもテーブル関数を使うことを好むが、その理由は、構文が手続き型言語に似ているからである。だがテーブル関数をそのような理由で使えば、コードを読みにくくするだけであり、保守性も下がる。

テーブル関数の本当のメリットは、手続き型言語からテーブル関数を呼び出す場合にある。つまり、非SQL的なデータソースから、SQLにデータを受け渡すために使うのだ。もし読者が大量の外的データを持っているなら、テーブル関数よりもETLツールを使うことを考えるべきだが、少量のデータならばテーブル関数は便利だ。手続き型言語はまた、SQLでは作るのが難しいデータを作るのにも向いている。たとえば、大半のSQLエンジンが持っていないような浮動小数点補正を含む統計計算が必要なデータなどである。それ以外にも、SQLは文字列操作や地理データの取り扱いや、他の言語が特化しているアプリケーションの操作にも向いていないので、こうした処理が必要なデータを「外側」で作って、その結果を取り込むのに便利だ。

【訳注1】　多くの実装はREFERENCINGオプションをサポートしていないため、このコードは動かない。Oracle版のコードはダウンロードサンプルを参照。

【訳注2】　著者はトリガーに対して極めて否定的な評価をしているが、これは何も著者に限った話ではない。たとえば、Oracle社は公式ドキュメントで以下のように述べている。

> トリガーを設計するときは、次のガイドラインを使用してください。
> （中略）
> - Oracle Databaseにすでに組み込まれている機能は、トリガーに重複定義しないでください。たとえば、宣言整合性制約で同じチェックができる場合、不良データを拒否するためにトリガーを定義しないでください。
> - トリガーのサイズを制限してください。トリガーのロジックが60行をはるかに超えるPL/SQLコードを必要とする場合は、コードの大部分をストアド・プロシージャに組み込んで、トリガーからそのプロシージャをコールすることをお薦めします。
>
> （中略）
> - 再帰トリガーは作成しないでください。たとえば、Emp_tab表に対してUPDATE文を発行するAFTER UPDATE文トリガーをEmp_tab表に作成すると、このトリガーはメモリ不足になるまで再帰的に起動し続けます。

> - DATABASEに対するトリガーは、慎重に使用してください。
>
> 出典：Oracle Databaseアプリケーション開発者ガイド-基礎編, 10gリリース2（10.2）
> http://otndnld.oracle.co.jp/document/products/oracle10g/102/doc_cd/appdev.102/B19248-02/adfns_triggers.htm

【訳注3】 たとえば、バージョン9iまでのOracleでは、GROUP BYを使った場合、結果セットはその順序でソートされていた。そのため、GROUP BYを使ったSELECT文では、ORDER BY 句をあえて記述しないというコーディングも行われていた。だが10gからは、GROUP BYを使っても結果セットがソートされることは保証されなくなった。これは、GROUP BYの計算にハッシュアルゴリズムが使われるように変更されたためだ。当時、この仕様変更による改修を余儀なくされたコードも少なくなかった。これは、最初からソートを行う際は明示的にORDER BY 句を使っていれば避けられていた問題であり、DBMSの内部実装に依存するコーディングをするべきでない理由である。

【訳注4】 シーケンスへアクセスする構文は、実装によって異なる。標準SQLに準じているのは、DB2とSQL Server である。それ以外の実装では以下のようになる（なお、

MySQLはシーケンスオブジェクトをサポートしていない）。

- Oracle：<シーケンス名>.nextval／<シーケンス名>.currval
- PostgreSQL：nextval('<シーケンス名>')／currval('<シーケンス名>')

また、NEXT VALUE FORとPREVIOUS VALUE FORを同一の文の中で使ったときの動作も実装によって異なる。
読者も自分の環境で使うときは注意してもらいたい。

8

補助テーブル

Auxiliary Tables

補助テーブルは、SQLの限られた計算能力では実現できないとは言わないまでも、実現が困難な関数や参照テーブルを作るための方法の1つである。SQLが得意なのは、計算ではなくテーブルの操作である。補助テーブルは、実際にはデータモデルの一部ではないが、計算ではなく結合によってクエリを助ける役割を果たす。

補助テーブルは極めて静的で外部のデータソースから構築される。したがって、データ整合性のために業務テーブルと同じ水準の制約は必要としない。とはいえ、オプティマイザは制約の情報をパフォーマンス向上に利用するため、制約は付けておくほうがよい。一般的には、冗長性を排除するためではなく、補助テーブルと他のテーブルの結合を高速化するため、主キーを付けておけば十分だろう。

最も重要な補助テーブルはカレンダーだ。西暦紀元カレンダーは極めて不規則で計算が難しい。祝日は月と太陽のサイクルに影響されるし、会計用の暦だけで数百種類もある。本節では、時間を扱うクエリにおけるカレンダーテーブルの使い方を議論する。また、さまざまな種類の数値を含む補助テーブルを見ていく。

8.1

数列テーブル

数列テーブルは、1 からn までの数値を網羅した単純なテーブルだ。SQL ではこのテーブルを手続き型言語におけるループ相当の機能を実現するために使う。カウンタの値をインクリメントするのではなく、値の集合を一度に得るのがSQL流だ。

私はかつてこのテーブルを“Sequence”と呼んでいたが、この単語はいまではシーケンスオブジェクトを意味する標準SQLの予約語であるため、使うのを避ける。“Numbers”という語を使う人々もいるが、これはランダムな整数の集合をイメージさせるため好ましくない。数列テーブルの一般的な定義は以下のとおりだ。

```
CREATE TABLE Series
(seq INTEGER NOT NULL PRIMARY KEY
     CONSTRAINT non_negative_nbr
     CHECK (seq > 0)
  cardinal_name VARCHAR(25) NOT NULL,
  ordinal_name VARCHAR(25) NOT NULL,
 ...
     CONSTRAINT numbers_are_complete
     CHECK ((SELECT COUNT(*) FROM Series) =
            (SELECT MAX(seq) FROM Series));
```

Series：数列テーブル　seq：連番　cardinal_name：基数　ordinal_name：序数

このテーブルに格納するデータは表8.1のようなものだ。

表8.1　整数の一覧

seq	cardinal_name	ordinal_name
1	one	first

2	two	second
3	three	third
…	…	…
101	one hundred and one	one hundred and first
…	…	…

このテーブルは1からnまでのすべての整数を網羅する。基数（cardinal_name）と序数（ordinal_name）の列は、整数の意味を示すための英語での呼び方である。

この数列をゼロから始めるのは――プログラマにとっては自然な習慣ではあるが――あまりおすすめできない。というのも、このテーブルはしばしば、他のテーブルとクロス結合することで行のナンバリングを行うために使われるのだが、ゼロが含まれていると1対1対応が崩れてしまうことになるからだ。もしゼロの行を作るとしたら、cardinal_name列は「zero」でよいだろうが、ordinal_name列はどう表現するべきだろうか？

8.1.1　リストの列挙

たとえば、月々の売上げデータの列（将来の値が不明の場合を考慮してNULLを許可する）を持つ、データウェアハウス用途のテーブルを考えてみよう。

```
CREATE TABLE AnnualSales1
(salesman CHAR(15) NOT NULL PRIMARY KEY,
 jan DECIMAL(5,2),
 feb DECIMAL(5,2),
 mar DECIMAL(5,2),
 apr DECIMAL(5,2),
 may DECIMAL(5,2),
 jun DECIMAL(5,2),
 jul DECIMAL(5,2),
 aug DECIMAL(5,2),
 sep DECIMAL(5,2),
 oct DECIMAL(5,2),
 nov DECIMAL(5,2),
 "dec" DECIMAL(5,2)); -- DEC は予約語のためダブルクォートで囲う
```

いまから、このテーブルの列を行に変換してみよう。求めるテーブルの形は次のようなものだ。

```
CREATE TABLE AnnualSales2
(salesman CHAR(15) NOT NULL,
 sales_month CHAR(3) NOT NULL
     CONSTRAINT valid_month_code
     CHECK (sales_month
              IN ('Jan', 'Feb', 'Mar', 'Apr',
                  'May', 'Jun', 'Jul', 'Aug',
                  'Sep', 'Oct', 'Nov', 'Dec')),
 sales_amt DECIMAL(5,2) NOT NULL,
     PRIMARY KEY(salesman, sales_month));
```

トリックは、オリジナルのテーブルから、月に加えて整数の列を持つビューを作ることである。

```
CREATE VIEW NumberedSales
AS SELECT salesman,
          1 AS M01, jan,
          2 AS M02, feb,
          3 AS M03, mar,
          4 AS M04, apr,
          5 AS M05, may,
          6 AS M06, jun,
          7 AS M07, jul,
          8 AS M08, aug,
          9 AS M09, sep,
          10 AS M10, oct,
          11 AS M11, nov,
          12 AS M12, "dec" -- 予約語
FROM AnnualSales1;
```

これで数列テーブルを使うことができる。列を行に変換するビューは次のとおりだ。

```
CREATE VIEW AnnualSales2 (salesman, sales_month, sales_amt)
AS SELECT S1.salesman,
          (CASE WHEN A.nbr = M01 THEN 'Jan'
                WHEN A.nbr = M02 THEN 'Feb'
                WHEN A.nbr = M03 THEN 'Mar'
                WHEN A.nbr = M04 THEN 'Apr'
                WHEN A.nbr = M05 THEN 'May'
                WHEN A.nbr = M06 THEN 'Jun'
                WHEN A.nbr = M07 THEN 'Jul'
                WHEN A.nbr = M08 THEN 'Aug'
                WHEN A.nbr = M09 THEN 'Sep'
                WHEN A.nbr = M10 THEN 'Oct'
                WHEN A.nbr = M11 THEN 'Nov'
                WHEN A.nbr = M12 THEN 'Dec'
                ELSE NULL END),
                (CASE WHEN A.nbr = M01 THEN jan
                WHEN A.nbr = M02 THEN feb
                WHEN A.nbr = M03 THEN mar
                WHEN A.nbr = M04 THEN apr
                WHEN A.nbr = M05 THEN may
                WHEN A.nbr = M06 THEN jun
                WHEN A.nbr = M07 THEN jul
                WHEN A.nbr = M08 THEN aug
                WHEN A.nbr = M09 THEN sep
                WHEN A.nbr = M10 THEN oct
                WHEN A.nbr = M11 THEN nov
                WHEN A.nbr = M12 THEN "dec" -- 予約語
                ELSE NULL END)
     FROM NumberedSales AS S1
           CROSS JOIN
          (SELECT seq FROM Series WHERE seq <= 12) AS A(nbr);
```

もし読者の使っているDBMS が導出テーブルをサポートしていれば、1 つのビュー定義で書いてもいい。

8.1.2　数列を循環数列にマッピングする

時として、数列を循環数列にマッピングすると便利なことがある。よくやる方法は、剰余演算だ。

```
SELECT seq, MOD (((seq + (:n-1))/ :n), :n)
  FROM Series;
```

たとえば、こういう問題を考えてみよう。帳票のレイアウトにおける「とぐろを巻く蛇」問題だ。部署とその職務内容を表すデータを考える。部署と職務にはIDが振られており、それを4 列固定で画面に表示したい。ポイントは、もし部署が4つより多い職務内容を持っていた場合は、行を折り返して表示することだ。同じキーのデータが複数行にわたってずらずら表示される姿が、蛇がとぐろを巻いているように見える。

```
CREATE TABLE Companies
(dept_nbr INTEGER NOT NULL,
 job_nbr INTEGER NOT NULL, -- dept_nbr内の連番
 company_descr CHAR(6) NOT NULL,
     PRIMARY KEY (dept_nbr, job_nbr));
```

Companies：会社テーブル　dept_nbr：部署番号　company_descr：仕事の内容

```
INSERT INTO Companies VALUES
  (1, 1, 'desc01'), (1, 2, 'desc02'), (1, 3,'desc03'),
  (2, 1, 'desc04'), (2, 2, 'desc05'), (2, 3, 'desc06'),
  (2, 4, 'desc07'), (2, 5, 'desc08'), (2, 6, 'desc09'),
  (3, 1, 'desc10'), (3, 2, 'desc11'), (3, 3, 'desc12');
```

中間ステップを見やすくするために、導出テーブルではなくビューを使おう。

```
CREATE VIEW Foo2 (dept_nbr, row_grp, d1, d2, d3, d4) AS
SELECT dept_nbr,
    MOD((job_nbr + 3) / 4, 4),
    MAX(CASE WHEN MOD(job_nbr, 4) = 1 THEN company_descr ELSE ' ' END) AS d1,
    MAX(CASE WHEN MOD(job_nbr, 4) = 2 THEN company_descr ELSE ' ' END) AS d2,
    MAX(CASE WHEN MOD(job_nbr, 4) = 3 THEN company_descr ELSE ' ' END) AS d3,
    MAX(CASE WHEN MOD(job_nbr, 4) = 0 THEN company_descr ELSE ' ' END) AS d4
  FROM Companies AS F1
 GROUP BY dept_nbr, job_nbr;

SELECT dept_nbr,
    row_grp,
    MAX(d1) AS d1,
    MAX(d2) AS d2,
    MAX(d3) AS d3,
    MAX(d4) AS d4
  FROM Foo2
 GROUP BY dept_nbr, row_grp
 ORDER BY dept_nbr, row_grp;
```

結果

```
dept_nbr  row_grp  d1      d2      d3      d4
================================================
1         1        desc1   desc2   desc3
2         1        desc4   desc5   desc6   desc7
2         2        desc8   desc9
3         1        desc10  desc11  desc12
```

このクエリは、MOD関数で4で割った余りの値を求め、それによって表示する列位置を決定している。部署番号が2のデータが行を折り返して表示されているのがわかるだろう。

一言注意しておくと、これはほめられたコーディングではない。見た目の整形はフロントエンドが担うべき機能であって、データベースで行うことではないからだ。

8.1.3 ループを置き換える

決して推奨すべきテクニックではないし、細かいところでSQLの方言による違いがあるのだが、集合で考えるようにするための練習としては面白い問題があるので紹介しよう。複数の整数をカンマで区切って並べた文字列があるとする。ここから、個々の整数を切り出してテーブルに行として登録していく。手続き型のコードで解くならば、文字列を先頭文字から1文字ずつループして整数を切り出していくというわかりやすいやり方になるだろう。

```
CREATE TABLE Outputs(outputinteger INTEGER);

CREATE PROCEDURE ParseList (inputstring IN VARCHAR)
IS
    i INTEGER;
    tmpstring VARCHAR(1000);
    outputstring VARCHAR(1000);
BEGIN
    i := 1;       -- ループ制御変数

    -- 入力値の文字列に終端を示すカンマを付加する
    tmpstring := TRIM (BOTH ' ' FROM inputstring) || ',' ;
    WHILE (i < LENGTH(tmpstring)) LOOP
        WHILE (SUBSTR(tmpstring, i, 1) <> ',') LOOP
        i := i + 1;
        END LOOP;

        outputstring := SUBSTR(tmpstring, 1, i - 1);
        INSERT INTO Outputs       VALUES   (CAST (outputstring AS INTEGER)) ;
        tmpstring := SUBSTR(tmpstring, i + 1) ;
    END LOOP;
END;
```

この問題を解くもう1つの方法は、数列テーブルを使う方法だ。その場合のクエリは、一見すると次のような奇妙なものになる。

```
SELECT S1.seq,
       CAST (SUBSTRING ((',' || '1,22,333,4444' || ',') FROM (S1.seq + 1)
        FOR (S2.seq - S1.seq - 1)) AS INTEGER)
  FROM Series AS S1,
       Series AS S2
WHERE SUBSTRING((',' || '1,22,333,4444' || ',') FROM S1.seq FOR 1) = ','
  AND SUBSTRING((',' || '1,22,333,4444' || ',') FROM S2.seq FOR 1) = ','
  AND S2.seq = (SELECT MIN(S3.seq)
                 FROM Series AS S3
                WHERE S1.seq < S3.seq
                  AND SUBSTRING((',' || '1,22,333,4444' || ',') FROM S3.seq FOR 1)
                                = ',')
                  AND S1.seq < CHAR_LENGTH ('1,22,333,4444') + 1;
```

たとえば、`inputstring`が`'1, 22, 333, 4444'`という文字列の場合、結果は次のようになる。

結果

```
seq  substring
==============
1    1
3    22
6    333
10   4444
```

奇妙というのは、このコードにはループが登場しないからだ。トリックは、入力値の文字列の左端と右端の両方にカンマを付加することだ。これを実用に堪えるコードにするためには、さらに`TRIM`関数で空白をあらかじめ取り除いたり、`LOWER(:instring)=UPPER(:instring)`という入力チェックを設けてアルファベットが混じっていないことを確認したりなどする必要があるだろう。

切り出された整数の文字列は、i番目から(i+1)番目のカンマの間にある。実質的には、数列テーブルはループカウンタを代替する役割を果たしている。もちろん、数列テーブルには、文字列全体をカバ

ーできるだけの十分な整数が格納されている必要がある。しかし、よほど大きな文字列を操作しようと思わない限り、その制限が問題になることはないだろう。

8.2

参照補助テーブル

かつて、エンジニアが計算尺を使っていた頃、その他の人々は数学と金融の本を参照して、印刷された関数の入力と出力の対照表を見ていた——三角関数、複利、種々の統計関数。今日では、マシンパワーが安くなったので関数を計算するほうが安上がりだ。1960 年代に数百ドルした電卓は、いまでは事務用品店でチューインガムの隣の棚に並んでいる。

データ入力をキーパンチで行っていた頃は、入力帳票の隣にエンコーディングスキーマをメモしたルーズリーフが置いてあったものだ。いまでは、エンコーディングスキーマが知りたいならWIMP（Windows、Icons、Menus、Pointing device）インターフェイスを使えばいい。

物理的なメカニズムが変わっても、（非リレーショナルな意味での）テーブルを作ることは変わらない。補助テーブルは、静的または比較的静的なデータを保持する。基本的にユーザーはデータを変更しない。もし補助テーブルのデータを更新することがあれば、その仕事はデータベース管理者またはリポジトリ管理者のものである。

単純な参照テーブルにおいてさえ生じる問題の1つは、データの変更がしばしばエンコーディングの体系自身の変更を伴うことだ。これを実施するには管理者権限を必要とする。補助テーブルの主キーは決して識別子ではない。識別子というのは、スキーマ内で一意であり、エンティティが現れるときは常にそれを示す。識別子は値であってエンティティではない。値を持つ参照テーブルは、その定義

からエンティティではない。`value_id`とか`<something>_code_id`といった名前を持つ列の奇怪さは、実際に使ってみるとわかる。補助テーブルは、こういうポストフィックスを持つ列がキーとして使われることがしばしばある。以下にはそのごく一部の例を挙げるが、ポストフィックスの完全なリストは拙著『SQL Programming Style』（Morgan Kaufmann、2005）を参照してほしい。

- _nbr またはnum：数値であることを示す。何かを示す数値の並び。_no というポストフィックスを使うのは避けるべきだ。これはブール型のyes/noの2値であるような印象を与える。個人的にはnbrのほうがnumよりよいと思う。というのは、これは複数のヨーロッパ言語においてnumberの一般的な略称だからだ。
- _nameまたはnm：見てのとおり、名称を意味する。名義尺度とも呼ばれる【訳注5】。
- _codeまたは_cd：コードは信頼できる情報源によって管理される、通常は業務の外部から与えられる体系である。たとえば、郵便番号は米国郵便公社によって管理されている。コードはその体系においてきちんと理解できればよいので、人間が解釈できなくてもかまわない。
- _cat：カテゴリ、すなわち外部の情報源を持つ体系で、異なるエンティティのグループから構成される。カテゴリを作るためには、厳密に形式的な基準が必要とされる。たとえば、生物学における「界」がその一例である【訳注6】。

- _class：クラス、すなわちエンティティの下位クラスを反映する外的な情報源を持たない、内的な体系。クラスを作るためには、厳密に形式的な基準が必要とされる。生物学における植物の「綱」が一例である。
- _type：タイプ、すなわち内的にも外的にも共通の意味を持つ体系。タイプは階級に比べるとより非形式的だが、重なる部分もある。たとえば、運転免許のタイプは、二輪、四輪、中型四輪、大型四輪、といった具合に分類される。

タイプ、クラス、カテゴリの違いは、分類に必要なアルゴリズムの厳密さにある。カテゴリは決して間違えようのない基準が必要とされる。実際、対象を「動物、植物、鉱物」のどれかのカテゴリに分類するときに、迷う人はいないだろう【訳注7】。

クラスは、ある共通の性質を持つ対象の集合である。たとえば、動物を哺乳類か爬虫類に分類するには基準が必要になる。その基準を適用するのが難しいケースもある。オーストラリアに住むカモノハシは卵を産む哺乳類である。しかしこうした例外は、それだけを含む独自の分類項目を作って処理するのが一般的だ（カモノハシで言えば、「単孔目」という専用のサブクラスが設けられている）。

タイプは3つの中では最も弱く、しばしば機械的な判断ができない。米国のある州では、三輪自動車は二輪免許で運転できるが、別の州では四輪免許が必要だし、さらにまた別の州ではバックギアが付いている場合だけ四輪とみなされる、などなど。実務の現場では、この3つの用語はしばしば混同されている。そういうときは、上

に挙げた定義には違反していたとしても、各業界の基準に従うべきだろう。

- _status：ステータス、多くの要因によって決定される結果の状態を反映する内的な体系。たとえば、信用状態（credit_status）は、政府や企業の債券や金融商品に対して、さまざまな要因を鑑みて格付けされる。
- _addrまたは_loc：住所または位置。
- _dateまたは_dt：日付。入社日、誕生日、失効日などさまざまな意味を持つ日付。

8.2.1　単純な読み替えを行う補助テーブル

最も一般的な参照テーブルの使い方は、2列だけを持つテーブルである。片方の列の値から、もう片方の列の値へ読み替えるために使う。単純な例を挙げるなら、ISO 3166で決められている国コードのテーブルだ。

```
CREATE TABLE CountryCodes
(country_code CHAR(2) NOT NULL PRIMARY KEY, -- iso-3166
 country_name VARCHAR(20) NOT NULL);
```

やろうと思えば、説明が記述されている列`country_name`にも一意制約を付けることが可能だが、一般的にはそこまで神経質になる必要はない。というのも、データはほとんど変わらないので不整合が起きる可能性は低いし、変更があるときも、信頼できる情報源からデータを得られるからだ。こういうゆるい考え方は、OLTPに慣れ

ているプログラマを不安にさせるかもしれないが、主にデータウェアハウスを扱っているプログラマなら理解してくれるだろう。

8.2.2　複数回の読み替えを行う補助テーブル

エンコーディングの体系はできれば不変であってくれるのが望ましいのは言うまでもないが、複数回の読み替えを行わなければならないこともよくある。略称と正式名称を使い分けなければならなかったり、参照する主体によって読み替え方が変わったりするときだ。例として、さまざまな言語でエラーメッセージを表示する例を考えよう。

```
CREATE TABLE ErrorMessages
(err_msg_code CHAR(5) NOT NULL PRIMARY KEY,
 english_err_msg CHAR(25) NOT NULL ...
 french_err_msg NCHAR(25) NOT NULL ...
 ...
 esperanto NCHAR (25) NOT NULL);
```

お気づきのように、このテーブル構成では、新しい言語を追加するたびに新規の列を追加しなければならない。これは欠点と言えば欠点だが、データが静的なので、読み替え後のメッセージを一元管理しておくことには十分なメリットがある。

もちろん、この方法を採用するとすべての言語に対応したエラーメッセージを持たなければならないので、（一般的にあまり良いテーブル設計とは言えない）列持ちテーブルになる。

このテーブルをアプリケーションで利用しようとすると、素直に考えればSQL 文はこうなるだろう。

```
SELECT CASE :my_language
       WHEN 'English' THEN english_err
       WHEN 'French'  THEN french_err
       ...
       END AS err_msg
  FROM ErrorMessages
 WHERE err_msg_code = '42';
```

だが実際にこういうSQL文を使うことはないだろう。普通は、ログインしているユーザーがわかるCURRENT_USER変数があるので、それから適切な言語を選択するテーブルを利用すれば、このエラーメッセージのテーブルに対するSELECT文はシンプルなものになる。

ともあれ、このテーブルの利点は、新しい言語はそうそう発明されない、という事実に基づいている。私は、クリンゴン語でエラーメッセージを表示できるソフトウェア製品を知っているが【訳注8】、真面目な話、それを見たのは国際化機能をアピールするためのトレードショーでのデモだった。もっとも、ユーザーフレンドリーというよりユーザーを怒らせるためにあるような機能だったが……。

8.2.3　複数のパラメータを持つ補助テーブル

このタイプの補助テーブルは、1つの値を探すために2つ以上のパラメータを使う。古典的な例は、大学の新入生を対象にした統計学のコースから得られる。サンプル数が少ない場合に使うスチューデントのt分布（表8.2）である。r（自由度）の値は標本数から1を引いたもの、パーセンテージは信頼区間である。rと信頼区間の2つのパラメータを入力として値を検索する。

スチューデントのt分布をおおまかに説明すると、一定レベルの信頼性を持った標準偏差を知ることができない場合に、分散を推計す

る最良の方法である。

表8.2　スチューデントのt分布表

r	90%	95%	97.5%	99.5%
1	3.07766	6.31371	12.7062	63.65600
2	1.88562	2.91999	4.30265	9.92482
3	1.63774	2.35336	3.18243	5.84089
4	1.53321	2.13185	2.77644	4.60393
5	1.47588	2.01505	2.57058	4.03212
10	1.37218	1.81246	2.22814	3.16922
30	1.31042	1.69726	2.04227	2.74999
100	1.29007	1.66023	1.98397	2.62589
∞	1.28156	1.64487	1.95999	2.57584

　この分布は1908年にウィリアム・ゴセットが発見した。彼が勤めていたビール会社Guinness Breweriesは従業員に論文発表を禁じていたため、「スチューデント（学生）」という匿名で論文を公表したのが、この分布の名前の由来である。

8.2.4　範囲補助テーブル

　範囲補助テーブルでは、パラメータは1つである。しかしそれはある範囲の中に収まらなければならない。よくある例は、帳票におけ

る期間や範囲だろう。この範囲は重なりあうこともある。たとえば、デパートの売り場における水着とバーベキューグリルのセール期間は大部分が重なっている。もっとも、一般的にはそれぞれの範囲は分けて管理したほうが何かとややこしくない。

```
CREATE TABLE ReportPeriods
(period_name CHAR(15) NOT NULL,
 period_start_date DATE NOT NULL,
 period_end_date DATE NOT NULL,
     CHECK(period_start_date < period_end_date),
     PRIMARY KEY (period_start_date, period_end_date));
```

ReportPeriods：レポート期間テーブル　period_name：期間名
period_start_date：期間開始日　period_end_date：期間終了日

データが静的なのでCHECK 制約がなくても間違いは少ないが、制約があったほうがオプティマイザが追加情報を得られるので、パフォーマンス上のメリットがある。検索はBETWEEN 述語を使う。終了日が決まっていない場合はNULLで代用するのが便利だろう。

同じようなケースで、学校の成績評価の等級の例を考えよう。

```
CREATE TABLE LetterGrades
(letter_grade CHAR(1) NOT NULL PRIMARY KEY,
 low_score DECIMAL(6,3) NOT NULL,
 high_score DECIMAL(6,3));
```

LetterGrades：等級テーブル　letter_grade：等級　low_score：下限点
high_score：上限点

```
INSERT INTO LetterGrades
VALUES ('F',0.000, 60.000),
       ('D', 60.999, 70.000),
       ('C', 70.999, 80.000),
       ('B', 80.999, 90.000),
       ('A', 90.999, NULL);
```

もし最後の範囲を('A', 90.999, 100.000)のようにしてしまうと、100点を超えるでき栄えを見せた生徒に等級を与えられない。NULL を使う以外の方法に('A', 90.999, 999.999)のようなダミー値を使うか、次のようにNULLを使った上でWHERE句でCOALESCE関数を使って値に変換する、という方法もある。

```
SELECT ...
  FROM ...
 WHERE Exams.score BETWEEN LetterGrades.low_score
                       AND COALESCE(LetterGrades.high_score, Exams.score);
```

どちらがよいかは、データの性質に依存する。

8.2.5　階層補助テーブル

階層補助テーブルでは、使うパラメータは1つだが、そのパラメータは1つ以上の範囲に収まらなければならず、かつ範囲は入れ子になっていなければならない。ある値が含まれるカテゴリのすべての経路を結果として得たい場合に使い、代表例は図書分類のデューイ十進分類法だ。

```
CREATE TABLE DeweyDecimalClassification
(category_name CHAR(35) NOT NULL,
 low_ddc INTEGER NOT NULL
     CHECK (low_ddc BETWEEN 000 AND 999),
 high_ddc INTEGER NOT NULL
     CHECK (high_ddc BETWEEN 000 AND 999),
     CHECK (low_ddc <= high_ddc),
     PRIMARY KEY (low_ddc, high_ddc));
```

DeweyDecimalClassification：デューイ十進分類法テーブル　category_name：カテゴリ名　low_ddc：下限分類コード　high_ddc：上限分類コード

```
INSERT INTO DeweyDecimalClassification
VALUES ('Natural Sciences & Mathematics', 500, 599),
       ('Mathematics', 510, 519),
       ('General Topics', 511, 511),
       ('Algebra & Number Theory', 512, 512),
       ...,
       ('Probabilities & Applied Mathematics', 519, 519);
```

511を範囲条件で検索すると、以下の3つの分類がヒットする。

- Natural Sciences & Mathematics
- Mathematics
- General Topics

階層のリーフノードは常に`(low_ddc = high_ddc)`であり、相対的なネストレベルは`(high_ddc - low_ddc)`か条件に指定した値そのものによって求められる。

このテーブルの場合も、範囲が重なることを防止するCHECK 制約を付けることが可能だが、通常はデータをテーブルにロードする時点でチェックされるため、テーブル自身にはそのような制約は付けない。ただし、`(low_ddc <= high_ddc)`というCHECK制約はオプ

ティマイザに有用な情報を提供するほか、主キー(low_ddc, high_ddc)は検索に有利なインデックスを作成する。

8.2.6　単一参照テーブル

私が知る限り、ポール・キースターがOTLT（One True Look-up Table：単一参照テーブル）という言葉の発明者である。これは新米SQLプログラマの間では非常にポピュラーなテクニックだ。後にD.C.ピーターソンはこれをMUCK（Massively Unified Code-Key）テーブルと呼んだ。その後もこのテクニックは繰り返し「発明」されてきた。アイデアを簡単に説明すると、すべてのコード参照を1つのテーブルで解決してしまおうというものだ。単一参照テーブルのよくある定義は次のようになる。

```
CREATE TABLE MagicalUniversalLookups
(code_type CHAR(10) NOT NULL,
 code_value VARCHAR(255) NOT NULL,
 code_description VARCHAR(255) NOT NULL,
     PRIMARY KEY (code_type, code_value));
```

MagicalUniversalLookups：魔法のようなごった煮参照テーブル　code_type：コードタイプ　code_value：コード値　code_description：コード内容

このテーブルを使えば、デューイ十進分類（図書コード）も、ICD（国際疾病分類）も、2文字だけからなるISO 3166の国コードも同じスキーマの1つの巨大テーブルに保持することになる。

このDDLが持つ問題と、これを使うと書かなくてはならない恐ろしいクエリを見ていこう（クエリをビューの中に隠すこともできるが、あまり慰めにはならない）。まず、オリジナルのDDL の

code_type 列にCHECK 制約を付けよう（この制約を付けないと、タイプミスによって新しいコード体系を「発明」するはめになる）。

デューイ10 進分類とICDは数字、ISO 3166 はアルファベットである。ということは、code_typeごとにそれぞれ別の制約を付けてコードの文字列が正しいフォーマットに従っていることを保証しなければならない。したがって、厳密に作ろうとするとテーブル定義は次のようになる——ここまでやる人はほとんどいないが。

```
CREATE TABLE MagicalUniversalLookups
(code_type CHAR(10) NOT NULL
     CHECK(code_type IN ('DDC', 'ICD', 'ISO3166')),
 code_value VARCHAR(255) NOT NULL,
     CHECK(CASE WHEN code_type = 'DDC'
                 AND code_value SIMILAR TO '[0-9][0-9][0-9].[0-9][0-9][0-9]'
                  THEN 1
                WHEN code_type = 'ICD'
                 AND code_value SIMILAR TO '[0-9][0-9][0-9].[0-9][0-9][0-9]'
                  THEN 1
                WHEN code_type = 'ISO3166'
                 AND code_value SIMILAR TO '[A-Z][A-Z]'
                  THEN 1 ELSE 0 END = 1),
 code_description VARCHAR(255) NOT NULL,
     PRIMARY KEY (code_value, code_type));
```

普通のアプリケーションで使われるデータベースには何十というコードが保存されるので、必要に応じて上記の制約パターンを繰り返すことになる。考えただけでも楽しい話ではないだろう？ そういうわけで、OTLTを使うプログラマがこのような制約を使うことはない。かくしてデータ整合性は損なわれる。

このテーブルにはまだ問題がある。気がついたと思うが、このテーブルの列はかなり長いVARCHAR(n)か、もっと悪い場合は

NVARCHAR(n)で定義する必要がある。

列長はたいていの場合、そのDBMSで可能な最大長で定義することになる。どのようなコードがテーブルに投入されるか、テーブルを作った時点では予測できないため、安全を見て十分に大きさなサイズで宣言するしかないのだ。文字列のサイズについての制約は、2番目のCHECK 制約で、code_typeとcode_value列の間に設定する必要がある。

このように大きな列長を許可していると、今度はそれが不正なデータを許すことにつながりかねない。VARCHAR(n)を使うということは、たくさんのスペースと、文字列の最後にくっついてくる可能性のある変なゴミ文字を許可するということだ。NVARCHAR(255)の列を使えば、誰かがいたずらで、ユニコードで中国語仏典の一節をテーブルに登録したとしても気づかない。著者がこう断言するのは、実際にあるデータベースのテストを頼まれたときに般若心経のデータをロードしたことがあるのだが、誰も気づかなかったからだ。

もし同じ値を持つコードの中で、code_typeやcode_descriptionに間違いがあれば、それを見つけるのは非常に困難だろう。500.000というコードは、デューイ十進分類では「自然科学／数学」を意味するが、ICDでは「炭鉱労働者の塵肺症」を意味する。似通ったコード体系を持つ2つのコードを混同して使ってしまったときに間違いを見つけるのは至難である。というのは、こういう場合、エラーは何も発生せず、ただ黙って結果不正になるからだ。

DMLの中でOTLTを使ったときの問題を具体的に見てみよう。コードを引くためには、コード値だけでなく常にcode_typeもWHERE句の条件に必要になる。

```
SELECT P1.ssn, P1.last_name, ..., L1.code_description
  FROM MagicalUniversalLookups AS L1, Personnel AS P1
 WHERE L1.code_type = 'ICD'
   AND L1.code_value = P1.icd
   AND ...;
```

このサンプルクエリでは、Personnelテーブルの傷病を表す列のcode_typeと、コードを含むすべての列を知っていなければならない。もしcode_typeを間違えても、エラーにはならず、間違った結果が返される。

また、型変換のため多少のオーバーヘッドも出ることになるだろう。デューイ十進分類ではVARCHAR(n)の代わりにDECIMAL(6, 3)を使うのが最も自然だ。だが別のコードではCHAR(7)が適している場合もあるだろう。もしコード体系ごとにテーブルを分けていれば、このような問題に頭を悩ませる必要はないし、外部キーを設定することで、コードテーブルを参照するテーブルのコードのデータ型も揃えることができる。OTLTには、このようなデータ型の選択に関する統一的な指針はない。

さらにOTLTの欠点を挙げよう。たとえ1種類のコード体系しか利用しない場合であっても、常にテーブルに含まれる複数のコード体系すべてが検索対象になる。もし物理ストレージにおいて、あるコードが先頭に格納されていて、別のコードが末尾に格納されていたら、I/O回数は非常に多くなる。OTLTテーブルを更新するときは、終

わるまですべての人をロックアウトして待たせることになる（行ロックならばまだ被害は軽減されるが）。たとえて言うなら、OTLTとは、実際に必要なのは1つの雑誌記事なのに、常に百科事典を持ち歩いているようなものである。

これは私の推測にすぎないのだが、このOTLTは、オブジェクト指向言語のプログラマが思いついたものではないだろうか。テーブルを一種のポリモルフィズムと考えているのだ。テーブルをクラスと考えれば、ポリモルフィズム的な動作をさせようとしてもおかしくはない。だがもちろん、テーブルはクラスではない。

データモデルには、よく設計されたテーブルは同じ種類の物事の集合であって、無関係な物事の寄せ集めではない、という原則がある。それには十分な理由があることなのだ。

8.3

補助関数テーブル

SQL はFORTRAN のような計算向きの言語ではないし、計算に特化した算術のパッケージも持っていない。たいていは、浮動小数点の丸め誤差を補正したり、オプティマイザが使う代数的な算術ルーチンも持っていない。だがその代わり、SQLは結合に優れている。補助テーブルの多くは、単純なデコードを目的としているが、もっと複雑な用途にも使える。

例として、なかなか簡単ではない金融分野の計算問題を考えてみよう。正味現在価値（Net Present Value：NPV)と、それに関連する内部収益率（Internal Rate of Return：IRR）の計算である。さて、ここでちょっと読者に聞いてみたい。SQLでどうやってNPVとIRRを計算すればよいだろうか？ こういう質問をニュースグループに投げると、必ずと言っていいほど返ってくる答えが、DBMS 製品が持っている4GL 言語でプロシージャを組むというものだ。だがそれでは練習にならない。

まずNPVから見ていこう。ちょうど運良く、読者が宝くじに当たったとしよう。読者は20 年間にわたって毎月一時払い金を受け取ることができる。このとき、最も有利になる条件はなんだろうか？ 1回の支払いで受け取るよりも、分割で受け取ったほうがトータルの金額は大きくなる。だが、もし特定の利率で1回の支払金を投資することができれば、そのほうがより多くの金額を受け取ることができるかもしれない。これは真剣に考えるべき資産運用の問題だ。

こういうとき、NPVは、一連の支払いを選択したとき、1回の支払額と同じになる利率はいくらかを教えてくれる。具体的なイメージ

を持つために、2つの投資オプションに対するテーブルとデータをお見せしよう。

```
CREATE TABLE CashFlows
(project_id CHAR(15) NOT NULL,
 time_period INTEGER NOT NULL
     CHECK (time_period >= 0),
 payment_amt DECIMAL(12,4) NOT NULL,
     PRIMARY KEY (project_id, time_period));
```

CashFlows：キャッシュフローテーブル　project_id：プロジェクトID
time_period：投資期間　payment_amt：支払額

```
INSERT INTO CashFlows
VALUES ('Acme', 0, -1000.0000),('Acme', 1, 500.0000),
       ('Acme', 2,400.0000),
       ('Acme', 3, 200.0000), ('Acme', 4, 200.0000),
       ('Beta', 0, -1000.0000), ('Beta', 1, 100.0000),
       ('Beta', 2, 200.0000),
       ('Beta', 3, 200.0000), ('Beta', 4, 700.0000);
```

投資対象のプロジェクトとしてAcme（'Acme'）とBeta（'Beta'）がある。最初に、1,000ドルを2 つのプロジェクトに投資する。このとき投資期間（time_period）は0 であり、合計の支払額は必ずマイナスになる。その後4 年間、異なる金額の一時払い金を毎年受け取ることになる。プロジェクトAcmeでは、合計1,300ドル（500＋400＋200＋200）を得る。最初の投資額1,000ドルを引いて、利益は300ドルである。同様にプロジェクトBeta では、最終的に1,400ドル（100＋200＋200＋900）を手にする。単純に比較すると、Betaのほうが良い投資ということになる。

だが、NPVで見ると、事態は異なる。いま、投資に対する割引率は10%で、キャッシュフローをまた投資に使うと仮定しよう。NPVを

求める公式を擬似的なコードで表すとこうなる。

```
FOR t FROM 0 TO n
  DO SUM(a[t]/ POWER(1.00 + r), t)
END FOR;
```

ここで`a[t]`は投資期間（`t`）に対するキャッシュフローで、投資期間(`t=0`）が初期投資を意味する（このときはNPV は常にマイナスになる）。r は割引率である。この数式を解くと、プロジェクトAcme のNPV は65、Beta は19となり、この指標ではAcme のほうが良い投資先である。つまり、BetaよりもAcmeのほうが、より多くのキャッシュフローを得られるのである。

8.3.1　補助テーブルを使った逆関数

IRRはNPVから求める。これは、ある投資で得られる金額と投資額がちょうど同じになるような割引率を知るための指標だ。もしこの割引率が高ければ、それだけ良い投資先ということになる。IRRを計算するためにもう1つテーブルを作ろう。

```
CREATE TABLE Rates
(rate DECIMAL(6,4) NOT NULL PRIMARY KEY);
```

次に、このテーブルにデータを入れる。割引率を入れるときのトリックは、クロス結合を使って値を現実的な範囲内に収めることだ。

```
CREATE TABLE Digits(digit DECIMAL (6,4) PRIMARY KEY);

INSERT INTO Digits VALUES (0.0000), (0.0001), (0.0002), (0.0003), (0.0004), (0.0005),
(0.0006), (0.0007), (0.0008), (0.0009);

INSERT INTO Rates (rate)
SELECT DISTINCT (D1.digit *1000) + (D2.digit *100) + (D3.digit *10) + D4.digit
  FROM Digits AS D1, Digits AS D2, Digits AS D3, Digits AS D4;
DROP TABLE Digits;
```

ここからは、2 つの選択肢がある。1 つは、キャッシュフローのテーブルを使うビューまたは共通表式（CTE）を作ること。この場合は次のようになる。

```
CREATE VIEW NPV_by_Rate(project_id, rate, npv)
AS
SELECT CF.project_id, R1.rate, SUM(payment_amt / POWER((1.00 + R1.rate), time_period))
  FROM CashFlows AS CF, Rates AS R1
 GROUP BY R1.rate, CF.project_id;
```

もう1つは、割引率1に対する合計額を数式の中に埋め込み、別のテーブルに`(rate, time_period)`ペアに対する乗数を持つ方法だ。

```
INSERT INTO NPV_Mulipliers (time_period, rate, npv_multiplier)
SELECT S.seq, R1.rate, SUM(1.00/(POWER((1.00 + R1.rate), seq)))
  FROM Series AS S, Rates AS R1
 WHERE S.seq <= {上限} --擬似コード
 GROUP BY S.seq, R1.rate;
```

数列テーブルには、1からnまでの整数が含まれている。これを使うのは、繰り返しを避けるための常套手段だ。ビューを使う方法を選択した場合、IRRは次のような1つのクエリで求められる。

```
SELECT 'Acme', rate AS irr, npv
  FROM NPV_by_Rate
 WHERE ABS(npv) = (SELECT MIN(ABS(npv))
                     FROM NPV_by_Rate)
   AND project_id = 'Acme';
```

私の用意したサンプルデータでは、プロジェクトAcmeに対してIRR＝13.99%が得られる。だが、投資先はそれこそ何百とあるので、そうしたケースもすべて1つのクエリで処理したいと思うだろう。

内挿法を使うと、プロジェクトAcmeに対する内挿は次のようなクエリで書ける。

```
SELECT R1.rate + (R1.rate * (R1.npv/(R1.npv - R2.npv))) AS irr
  FROM NPV_by_Rate AS R1, NPV_by_Rate AS R2
 WHERE R1.project_id = 'Acme'
   AND R2.project_id = 'Acme'
   AND R1.rate = 0.1000
   AND R2.rate = 0.2100
   AND R1.npv > 0
   AND R2.npv < 0;
```

結果は14.01%であり、先ほどの結果とかなり近い。

重要なことは、R1とR2のNPVはともに正負の符号は同じでなければならないということだ。でないと、線形内挿は使えない。これは非常に高速なクエリだが、その代わり正確性を犠牲にする。IRR関数は微妙な曲線を描き、線形ではないからだ。グラフを描いてみるとわかるが、曲線は下に凸となる。したがって、正確な近似を得るためには、うまく`(R1.rate, R2.rate)`のペアを選ぶことが重要だ。だが、近似的な値を求められればよいならば、きっと想定よりも広い範囲を使うことができるはずだ。実際、もともと参照テーブルを

使って求めた0.1399という数値に対して、表8.3の内挿で求めた近似値は14%であり、決して悪くない。

表8.3　内挿法によるIRRの推定値

R1	R2	IRR
0.1000	0.2100	0.140135
0.1000	0.2000	0.143537
0.0999	0.2000	0.143457
0.0999	0.1999	0.143492
0.0800	0.1700	0.135658

補助関数テーブルを使うことのメリットは以下のとおりだ。

メリット1　どんなホスト言語でも同じ計算を実装できる。

メリット2　スプレッドシートなどを使うときと同様、非常にたくさんのプロジェクトに対して一度に数式を適用できる。

ロバート・J・ハミルトンは、実装依存のT-SQLでNPVとIRRを求める関数を提供してくれた。NPV関数は、そのままずばりだが、IRRを求めるにあたり、彼はいくつかの問題を指摘した。

定義上、IRRはキャッシュフローのNPVがゼロになる割引率のことだ。もしIRRが正当な値を取れば、NPV 曲線はグラフ上で一度しかx軸と交わらない。だがIRR が異常な値を取ると、NPV曲線は複数回x

軸と交わる。ここでIRR が異常とは、IRR が複数の値を持ったり、定義されていない状態のことだ。

この点については、我々はIRRの適切な定義域の範囲を考えなければならない。NPVについて言えば、正負あらゆる数を取りうる――NPV があまり漸近線に近づきすぎると（たとえば–100%なんていう値を取ると)、累乗関数がオーバーフローエラーを返してしまうという問題はある。だがNPVを計算するとき、負の割引率とはいったい何を意味するのだろう？ これはIRRをどのように解釈するかによって変わってくる。

数学的な観点から見れば、負のIRRは単に方程式の解の1つにすぎない。だが経済学的な観点から見れば、それはプロジェクトに投資するとお金を失う（キャッシュの減少）という意味だ。デフレーションが起きている状況ならば、キャッシュフローがマイナスになるのは有益かもしれない。しかしそれはごく例外的な状況であり、通常は、IRRが負になるプロジェクトは儲けが出ないとして却下してよい。

これが意味することは、参照テーブルを使うアプローチでは、IRRは極めて細かい粒度と、一般的なケースをカバーするのに十分な範囲を持つ必要があるということだ。要するに、参照テーブルを使うのはIRRを計算するのに実用的な方法ではないということでもある。小数点第5 位や6 位ぐらいまで割引率を求めるのは、住宅ローンなどでは当たり前のことだ。この精度を計算するには、集合ベースのアプローチはあまりうまくスケールしない。

8.3.2　補助関数テーブルを使った内挿

SQLは関数型言語ではないため、標準SQLの限られた力では書けない関数もある。そうした場合にはベンダーの独自拡張のライブラリに頼る必要がある。だがSQLはテーブルを扱うのに長けており、補助テーブルを一般的な形式で用意しておけばうまく利用できる。つまり、関数の引数と戻り値のペアを持ったテーブルを作るのだ。

```
CREATE TABLE SomeFunction
(parameter <data type> NOT NULL PRIMARY KEY,
 result <data type> NOT NULL);
```

もし関数の定義域が比較的狭ければ、擬似コードはこうなる。

```
SELECT SomeFunction(T1.x), ...
  FROM TableOne AS T1
 WHERE etc
```

これはテーブル参照を使えば、次のように置換できる。

```
SELECT F1.result,
  FROM TableOne AS T1, SomeFunction AS F1
 WHERE T1.x = F1.parameter
   AND etc
```

だが関数の定義域が広い場合は、`SomeFunction`テーブルが大きくなり、パフォーマンス上実用的ではない。これに対処するテクニックの1 つが内挿法である。この方法は正確な値を計算できない場合に近似を求める方法の1 つで、高性能のコンピュータが安く手に入るようになって、計算コストが下がってからは廃れつつある。考え方としては、2 つの既知の関数の入力aとb、および出力f(a)とf(b)を用意して、未知の入力x（ただしaとbの中間にある）に対する結果を求めるのだ。

最も単純な方法は線形内挿である。もしテーブルが十分な精度のデータを持っていれば、たいていのアプリケーションに対しては非常にうまく働く。線形内挿は、2 つの出力f(a)とf(b)の間には直線を引くことができる、という前提に立っている。

線形内挿の公式は次のとおりである。

f(x) = f(a) + (x – a) * ((f(b) – f(a))/(b – a))

ここで(a≦ x≦ b)であり、x はテーブルには存在しない。この数式をSQLに翻訳するとこうなる（xは:myparameter、F1は変数a、F2は変数4に相当する）。

```
SELECT :myparameter AS my_input,
       (F1.answer + (:myparameter - F1.param)
         * ((F2.answer - F1.answer)
         / (CASE WHEN F1.param = F2.param
                 THEN 1.00
                 ELSE F2.param - F1.param END))) AS answer
  FROM SomeFunction AS F1, SomeFunction AS F2
 WHERE F1.param -- aとf(a)を作る
           = (SELECT MAX(param)
                FROM SomeFunction
               WHERE param <=:myparameter)
   AND F2.param -- bとf(b)を作る
           = (SELECT MIN(param)
                FROM SomeFunction
               WHERE param >= :myparameter);
```

除数のCASE式はゼロ除算を防止するために使っている。内挿のアルゴリズムは常に四則演算で表現できる。ということは、標準SQLでも十分に表現できるということだ。余談だが、かつて関数テーブルには入力と出力のペア以外に、「デルタ2乗（delta squared）」という第3の列が存在していた。これは有限差分をもとに計算した値だ。

デルタ2 乗は、2 次導関数に似ていて、数式の近似出力の正確性を改善するのに使うことができた。本書は数値解析の本ではないので、これ以上は踏み込まないが、興味のある方は図書館へ行くか年輩のエンジニアにたずねてみてほしい。

8.4

グローバル定数テーブル

システムを作るとき、スキーマに定数をセットし、それを維持するための方法が欲しくなるだろう。1つの方法は、1行だけのテーブルを作り、最初にデフォルト値をセットしておき、管理者権限を持つ誰かが更新するタイミングでのみ更新していくというものだ。

```
CREATE TABLE Constants
(lock CHAR(1) DEFAULT 'X'
     NOT NULL PRIMARY KEY
     CHECK (lock = 'X'),
 pi  FLOAT DEFAULT 3.141592653 NOT NULL,
 e   FLOAT DEFAULT 2.71828182 NOT NULL,
 phi FLOAT DEFAULT 1.6180339887 NOT NULL,
...);
```

Constants：定数テーブル　lock：ロック　pi：パイ　e：イー　phi：ファイ

テーブルにデータをロードするには、次の文を実行する。

```
INSERT INTO Constants VALUES (DEFAULT, DEFAULT, DEFAULT, DEFAULT);
```

lock列の主キーとCHECK制約は、このテーブルに1行しか存在し得ないことを保証している。このテーブルでは保持する列の例として数学定数を使ったが、実際の業務システムでは、カレントユーザーやカレントタイムスタンプを保持するのが便利だ。基本的に更新を許さないテーブルというアイデアの変奏として、テーブル構築子を使ってビューを定義する方法がある。

```
CREATE VIEW Constants (pi, e, phi)
AS VALUES ((3.141592653), (2.71828182), (1.6180339887));
```

次のステップは、このDDLを異なるプラットフォームに移しても、そこでの数学ライブラリやハードウェアの精度においても計算できるよう、定数を定式化することだ。

【訳注5】 名義尺度とは、統計学の用語で、数字を単なる名前として対象に割り当てたもの。例としては、性別の男性に「1」、女性に「2」を割り当てたり、血液型のA型に「1」、B型に「2」を割り当てたりすること。この場合、数値の大小関係に意味はなく、ただ対象を区別をするためだけに使われるのが特徴。

【訳注6】 生物学における「界」は、古典的には動物界と植物界の2つのカテゴリからなるが、これに原生生物界も加えた三界説や、六界説も提唱されており、必ずしも分類基準は確定的ではない。

【訳注7】 実際にはウイルスのように生物か非生物かで論争のあった対象も存在するが（現在ではウイルスは非生物に分類されている）。

【訳注8】 クリンゴン語は、米国のSFテレビドラマ『スタートレック』に登場する架空の異星人が操る言語。米国のプログラマやエンジニアには『スタートレック』のファンが多く、これを元ネタにしたギャグはいわば彼らの内輪ネタの十八番であるが、日本人には理解できないこともしばしば。

9

正規化

Normalization

関係モデルと正規化を最初に定義したのはエドガー・F・コッドだった（Codd, 1970）。その後、何人かの論者がこれを拡張した。コッドは「正規形」という言葉を当時の政治的ジャーゴンから借りてきた【訳注1】。数学においては、関係という言葉は述語計算によって定義された集合間の対応を扱うものだ。代数の方程式と同様、同じ関係式にも多くの形式があるが、関係の「正規形」は記述可能な構造を形式的に定義したものだ。正規形の目的は、非正規化された状態のテーブルで起きるデータ不整合を防ぐことである。

データ不整合が何かを説明するには、実例を見てもらうのが早い。だが最初は、我慢して理論的な定義に付き合っていただきたい。ある述語が A(x)という文の形式で表現されるとき、この文はxが属性Aを持つ、ということを意味する。たとえば、「ジョンはインディアナ出身だ」という文を、述語を使って表現すると「ジョン」が対象で「インディアナ出身」が述語になる。x＝ジョン、A(x)＝インディアナ出身、ということだ。形式的に表現するなら、インディアナ出身（ジョン）である【訳注2】。

関係（リレーション）は、2つ以上の対象を持つ述語のことである。たとえば「ジョンとボブは兄弟だ」という文は、関係文の例である。形式的に表現するなら、兄弟（ジョン，ボブ）となる。兄弟

という関係が、ジョンとボブという2つの対象に適用されているわけだ。この関係文の集合を視覚的に表現する一般的な方法が、テーブルである。テーブルの列は属性を、行は特定の関係文を意味する。

コッドが関係モデルを定義したとき、彼は、関係をテーブルとして視覚化するための0番から12番までのルールを設けた。

ルール0

システムはRDBMSとしての条件を満たしている必要がある。つまり、システムは関係に基づく機能（のみ）を使ってデータベースを管理しなければならない。SQLは、しばしば手続き型のやり方でのデータアクセスを許すため、必ずしもこのルールに必ずしも忠実ではないが、SQLはいまでも進化しており、時が経つにつれていまよりもこの目標の達成に近づいていくだろう。

ルール1

情報のルール：これは単純で、データベース内のすべての情報はただ1つの方法によって表現されなければならない。つまり、テーブルの行における特定の列の値はスカラ値でなければならない。SQLはここでも十分にルールを守っているとは言いがたい【訳注3】。しかし、列は名前によってアクセスすることが可能であり、位置によってアクセスするわけではない。SQLでは列名をリストする代わりに「*」を使うことができるが、これはただの省略形であり、デフォルトの列名の順序は、テーブル定義における列の順番である。

ルール2

アクセス保証のルール：このルールは、本質的には、キーの必要性を言い直しただけである。つまり、データベース内のすべてのスカラ値は、それを含むテーブルと列の名前、およびキーの値を指定することで、論理的に特定可能でなければならない、ということである。SQLは、キーを持っているテーブルに対しては、このルールに従う。しかし、SQLは、すべてのテーブルがキーを持つことまでは要求していない【訳注4】。

ルール3

NULLの体系的な取り扱い：DBMSは失われた値および適用不能な値を表現する手段をサポートする必要がある。そうした値は、通常の値からは区別され、あらゆるデータ型から独立している。また、そのような表現はDBMSによって体系的に行われる必要がある。コッドは失われた値と適用不能な情報を分けて管理することを望んでいた（4値論理）。だがSQLは、2つを区別せず、`NULL`という1つの値で表現している。

ルール4

関係モデルに基づいた動的オンラインカタログ：システムは通常の問い合わせ言語でユーザーがアクセス可能なオンラインのカタログをテーブルとして持たなければならない。SQLはこの条件を満たしている。

ルール5

包括的なデータサブ言語のルール：システムは、①線形の構文を持った、②インタラクティブにもアプリケーション内でも利用できる、③データ定義操作（ビュー定義を含む）／データ操

作（参照および更新）／トランザクション制御（`begin`、`commit`、`rollback`）が可能な関係言語を少なくとも1つサポートしなければならない。SQLはこの点については高得点を取れる。コッドが定義したすべての操作はSQLで実施可能である。

ルール6

更新可能なビューのルール：理論的に更新可能なすべてのビューは、システムによって更新可能でなければならない【訳注5】。SQLはこの点は弱い。標準SQLでは、極めて安全な不整合の起きないケースでのみ、ビューの更新を認めている。更新可能なビューの問題は、いまではNP 完全問題の1 つに属することが知られており、したがって、一般的にすべての更新可能なケースを網羅することは現実的でない。いくつかのケースでは、SQL の`INSTEAD OF`トリガーが解になることもある。

ルール7

高レベルのINSERT、UPDATE、DELETE：システムは、集合レベルでの`INSERT`、`UPDATE`、`DELETE`操作をサポートしなければならない。SQLはこれをサポートしている。

ルール8

物理的データ独立：これは読んで字のごとくである。データベースを使うとき、ユーザーは物理的実装を意識することはなく、論理モデルのみを操作する。もっとも、どんな製品であれ、何らかの物理依存な特徴は持っているものだが、それでもたいていのプログラミング言語と比べれば、SQLはこの点では優れている。とりわけ、インデックスやアクセス方法の検討とい

った物理レベルの仕事がDBAに任されていて、ユーザーが論理レベルのみを考慮すればよいことはSQLの大きな利点である。

ルール9

論理的データ独立：これもまた読んで字のごとく。論理的データ独立性とは、アプリケーションプログラムの変更をすることなく、概念スキーマの変更ができることである。SQL は、きちんとした汎用的な書き方をしていれば、この点でも非常に優秀だ。

ルール10

整合性の独立：整合性制約はアプリケーションプログラムとは独立に制御されなければならず、カタログに保存されるべきである。既存のアプリケーションプログラムの変更なしに制約の変更が可能でなければならない。標準SQLではこれが可能である。

ルール11

分散独立性：DBMSが分散配置された場合でも、アプリケーションは、それを意識することなく利用を継続できなければならない。

ルール12

破壊防止のルール：もしシステムが低レベルの（つまりレコードレベルあるいはビットレベル）のインターフェイスを提供しているならば、そのインターフェイスが（たとえば、関係モデルのセキュリティや整合性制約を迂回することで）システムを

破壊するようなことを許してはならない。SQLはこの点では問題ない。

コッドはほかにも、リレーショナルデータベースが備えるべき9 つの構造的な特徴、3 つの整合性の特徴、18 の操作面での特徴を定義しており、当然、真のリレーショナルデータベースはそれらすべてを満たさねばならないとした。ちなみに、後に彼は、上記の12 のルールを333まで増やしている。詳しく知りたい人は『The Relational Model for Database Management:Version 2』（Codd, 1990）を参照してもらいたい。

さて正規形とは、データベースにおいて正しいデータを破壊しないこと、または間違ったデータを作らないことを保証するための試みである。そうしたエラーを回避するための方法の1つは、データベースにおいて1つの事実を1つの場所にだけ格納することだ。というのも、もし1つの事実が2 つの場所に現れるとしたら、更新の際に正しく同期を取らないと不整合が起きることになる――2 つの腕時計をしている人間は、どちらが正しい時刻を指しているのかという不安に常につきまとわれる。だから我々は、多くの時計を1 つのマスタとなる時計に同期させ、1 つのデータソースから派生するビューとして扱うのだ。

こうしたテーブル設計のプロセスのことを正規化と呼ぶ。正規化は決して不思議なものではないが、複雑だと思うことはあるだろう。正規化を自動化してくれるCASEツール（Computer Assisted Software Engineering）もあるが、まずはツールを使う前に理論について理解しておこう。

9.1

関数従属性と多値従属性

正規形はテーブルを関数従属性（functional dependency：FD）という基準に従って分類する作業である。関数従属性は、もし1つの属性の値がわかれば、常に別の属性の値もわかる、という性質である。関係理論での表記法では、2 つの属性の間に矢印（→）を引く。たとえば、A → Bは、「AはBを決定する」とか「BはAに従属する」と読む。具体的な例としては、社員番号がわかれば社員名がわかるとか、部品番号がわかれば、その部品の重さや色がわかる、という具合だ。

多値従属性（multivalued dependency）は、もし1 つの属性の値がわかれば、常に別の属性の集合の値がわかる、という性質である。関係理論での表記法は、矢印の矢尻を2 つ重ねて表記する。「A ↠ B」は「A はB の集合を決定する」と読む。具体的な例としては、教師の名前がわかれば生徒の一覧がわかるとか、部品番号がわかれば、その構成部品の番号もわかるという具合だ。

9.2

第1 正規形（1NF）

学校の時間割のデータを管理する業務を考えよう。まず管理すべき情報をざっと挙げると以下のようなものがあるだろう。

- 講座名（course_name）
- クラス分類（class_section）
- 分野（dept_name）
- 時間（time_period）
- 教室番号（room_nbr）
- 教室サイズ（room_size）
- 教授（professor）
- 学生（student）
- 専攻（student_major）
- 学年（student_grade）

これを手続き型言語、たとえばPascalの構造体（レコード型）で定義するなら次のようになる。

```
Classes = RECORD
    course_name: ARRAY [1:7] OF CHAR;
    class_section: CHAR;
    time_period: INTEGER;
    room_nbr: INTEGER;
    room_size: INTEGER;
    professor: ARRAY [1:25] OF CHAR;
    dept_name: ARRAY [1:10] OF CHAR;
    students: ARRAY [1:class_size] OF RECORD
        student_name: ARRAY [1:25] OF CHAR;
        student_major: ARRAY [1:10] OF CHAR;
        student_grade: CHAR;
    END;
END;
```

Pascalを選んだのは、コードが読みやすいのと、データ型の名前がSQLと少し似ているからだ。ただしこのテーブルは、リレーショナルデータベースならば、最も基本的な正規形も満たしていない。第1 正規形（First Normal Form：1NF）は、テーブルがすべての列がスカラ値のみから構成される、すなわち配列のような繰り返しグループが存在しないという条件を満たす必要がある。`students:`列は、内部に`student_name:`、`student_major:`、`student_grade:`という列を含んでおり、スカラ値ではないので第1正規形違反である。

SQLでは、ベンダーが配列型や複合型の拡張を実装していない限り、逆に1NF以外のテーブルを作ることは不可能である【訳注6】。PascalのRECORDをSQLで表現する際には、入れ子構造を次のように「フラット化」する必要がある。

```
CREATE TABLE Classes
(course_name CHAR(7) NOT NULL,
 class_section CHAR(1) NOT NULL,
 time_period INTEGER NOT NULL,
 room_nbr INTEGER NOT NULL,
 room_size INTEGER NOT NULL,
 professor_name CHAR(25) NOT NULL,
 dept_name CHAR(10) NOT NULL,
 student_name CHAR (25) NOT NULL,
 student_major CHAR(10) NOT NULL,
 student_grade CHAR(1) NOT NULL);
```

このテーブルならば、SQLで作ることができる。実際には、course_name、class_section、student_nameの3列で行を特定することができるので、この列に主キー制約を付与しておくべきだ。ここで行った変更はstudents:のレコード配列をフラット化しただけだが、それによって本質的に失われた情報があるわけではない。ただ表現方法が変わっただけだ。

さて、ここからが問題だ。数学科の「Jones」教授が亡くなって、Classesテーブルから彼の存在するすべてのレコードを削除しようとすると一緒に彼の数学の講義を取っていた生徒たちの情報も削除されることになる。だが別に生徒たちまで大学を去ったわけではない。つまり、1 つの事実を削除しようとしただけなのに、それ以上の事実まで削除してしまうことになるのだ。これが削除異常である。

あるいは、生徒の「Wilson」が、「Jones」教授の数学から英語の講義へ受講を変えようとした場合、そのままでは英語の講義も「Jones」教授が教えることになってしまう。単純に1 つの事実だけを変えたのでは整合性が取れないのだ。誤った情報の結び付きが生まれてしまう。これが更新異常である。また、学校が新しい学科を新設しようとした場合、まだ学生は1 人もいないので、新たに雇った

教授が受け持ちの講義と生徒を持つようになるまでClassesテーブルに行を挿入できない。単純な1 つの事実を挿入することができないわけだ。これが挿入異常である。

このテーブルにはこれ以外にもまだまだ問題があるが、要点はわかってもらえたと思う。テーブルを変更することなく、これらの問題を回避する方法はいくつかある。1 つは、テーブルにNULLを許可することだ。また、誤ったデータをチェックするためにトリガーを書くという方法もある。しかしこれらは、データと関連をより複雑にするだけの悪手である。本当の解決策は、1つの関連や1つの事実の簡単な集合を表すようなテーブルに分割することだ。

9.2.1　繰り返しグループについての注意

第1正規形の定義は、テーブルが繰り返しグループを持っていないこと、すなわちすべての列がスカラ値であるということだ。これは、他のプログラミング言語で見られるような、配列、リスト、テーブル内テーブル、レコード構造を持たないということだ。昔の標準SQL は、これらをサポートしていなかったので、こうした複合型のデータを回避することは簡単だった。しかし、いくつかの極めて非リレーショナルな「機能」を導入したSQL:1999 以後は、話はそう単純ではなくなった。ベンダーの中には、配列、ネストしたテーブル、variantデータ型といった機能を独自にサポートするものも現れた。リレーショナル理論の純粋性というアカデミックな観点を別にしても、こうした新機能は使うべきでない。ベンダー独自機能は広く実装されていないため、拡張性がなくプラットフォーム間の移植が難しい。さらに、オプティマイザがこうした機能を利用するこ

とも簡単ではないので、新機能の利用はパフォーマンスを悪くさせる。

古い習慣を変更するのは難しいものだ。それゆえSQLの経験が浅いプログラマは、自らの古い世界観をSQLに当てはめようとする。次に、それらの例を見ていくことにしよう。

繰り返し列

SQLで配列を「擬似的」に表現する方法の1 つが、意味的に同じ複数の列から構成される列グループを使うことだ。「意味的に同じ」とは、テーブルにおいて同じ属性を表現するということである。社員とその子どものテーブルを例に考えよう。

```
CREATE TABLE Employees
(emp_nbr INTEGER NOT NULL,
 emp_name VARCHAR(30) NOT NULL,
 ...
 child1 CHAR(30), birthday1 DATE, sex1 CHAR(1),
 child2 CHAR(30), birthday2 DATE, sex2 CHAR(1),
 child3 CHAR(30), birthday3 DATE, sex3 CHAR(1),
 child4 CHAR(30), birthday4 DATE, sex4 CHAR(1));
```

Employees：社員テーブル　emp_nbr：社員番号　emp_name：社員名　child1～child4：子ども1～子ども4　birthday1～birthday4：誕生日1～誕生日4　sex1～sex4：性別1～性別4

これは、COBOLや他の3GL言語における多くの古いファイルシステムのレコードのレイアウトのように見えるだろう。子どもの誕生日と性別の情報が繰り返しグループを形成している。これはいわば、SQL で4 つの要素を持つ配列を擬似的に表現しているのだ。添え字は、ここでは単なる列名の一部でしかない！これはもちろん、第1正規形に違反している【訳注7】。

別の例として、特定の年の各月で、製品（product_name）の販売数量を記録するテーブルを考えよう。まず最初に、次のようなテーブルを作ったとする。

```
CREATE TABLE Abnormal
(product_name CHAR(10) NOT NULL PRIMARY KEY,
 month_01 INTEGER, -- NULLは当月のデータが存在しないことを意味する
 month_02 INTEGER,
 ...
 month_12 INTEGER);
```

Abnormal：正規形違反テーブル　product_name：製品名　month_01～month_12：月01～月12

より正規化された形にテーブルをフラット化するとこうなる。

```
CREATE TABLE Normal
(product_name CHAR(10) NOT NULL,
 month_nbr INTEGER NOT NULL,
 product_qty INTEGER NOT NULL,
     PRIMARY KEY (product_name, month_nbr));
```

Normal：正規形テーブル　product_name：製品名　month_nbr：月　product_qty：販売数量

このテーブルに先の正規形違反テーブルからデータを入れるには次の文を使う。

```
INSERT INTO Normal (product_name, month_nbr, product_qty)
SELECT product_name, 1, month_01
  FROM Abnormal
 WHERE month_01 IS NOT NULL
UNION ALL
SELECT product_name, 2, month_02
  FROM Abnormal
 WHERE month_02 IS NOT NULL
...
UNION ALL
SELECT product_name, 12, month_12
  FROM Abnormal
 WHERE month_12 IS NOT NULL;
```

UNION ALLの処理はとても遅いが、正規化されたテーブルにデータをロードするときにだけ使うものなので、我慢できるだろう。

文字列のリストを受け取って、こういう正規化テーブルにロードするにはどうすればいいだろうか。文字列のリストをパースする一般的な方法は、カンマ区切りで文字列を埋めることだ。その場しのぎの方法に頼らないとすれば、いくつかのステップを踏まなければならない。次の例で考えてみよう。

```
CREATE TABLE InputStrings
(key_col CHAR(10) NOT NULL PRIMARY KEY,
 input_string VARCHAR(255) NOT NULL);
```

InputStrings：入力文字列テーブル　key_col：キー　input_string：入力文字列

```
INSERT INTO InputStrings VALUES ('first', '12, 34, 567, 896');
INSERT INTO InputStrings VALUES ('second','312, 534, 997, 896');
...
```

このテーブルは、1行につきカンマ区切りの文字列全体を格納する。ここから、以下のテーブルに、1行につき1つのパラメータとキー列を持つような形式でデータを移したい。

```
CREATE TABLE Parmlist
(key_col CHAR(10) NOT NULL,
 place INTEGER NOT NULL,
 parm INTEGER NOT NULL,
     PRIMARY KEY (key_col, place));
```

Parmlist：パラメータリストテーブル　key_col：キー　place：位置　parm：パラメータ

入力文字列の開始と終了がカンマで囲まれていれば、処理はもっと簡単になる。1からnまでの数列テーブルを使おう。

```
INSERT INTO Parmlist
SELECT key_col,
       COUNT(S2.seq) AS place,
       CAST (SUBSTRING (',' || I1.input_string || ',',
               MAX(S1.seq + 1), (S2.seq - MAX(S1.seq + 1))) AS INTEGER)
  FROM InputStrings AS I1, Series AS S1, Series AS S2
 WHERE SUBSTRING (',' || I1.input_string || ',', S1.seq, 1) = ','
   AND SUBSTRING (',' || I1.input_string || ',', S2.seq, 1) = ','
   AND S1.seq < S2.seq
   AND S2.seq <= CHAR_LENGTH(I1.input_string) + 2
 GROUP BY I1.key_col, I1.input_string, S2.seq;
```

Parmlist

key_col	place	parm
first	1	12
first	2	34
first	3	567
first	4	896
second	1	312
second	2	534

second	3	997
second	4	896

S1とS2はSeriesテーブルのコピーであり、文字列の両端をカンマで囲むために使っている。これによって、カンマで囲まれた部分文字列の集合全体が抜き出され、1 つの非手続き的なステップで整数に型変換される。このコードのトリックは、ペアとなるカンマのうち、左側のカンマが2 番目のカンマに最も近いという事実を利用している点だ。place 列は、入力文字列内におけるパラメータの相対的な位置（つまりカンマで区切って何個目のパラメータか）を意味する。

こうした方法によって、非正規形テーブルから正規形テーブルへの変換はSQLでも可能なのだが、それを承知の上で言うと、一番正しい方法は、最初から正規形テーブルにデータをロードすることである。これを実現するには、SQL パラメータに配列の要素を渡せる言語を使う必要がある。サンプルの構文は次のようなものだ。

```
INSERT INTO Parmlist
VALUES (:a[1]), (:a[2]), (:a[3]), ..., (:a[n]);
```

さらにNULLと重複行を排除するには次のようになる。

```
INSERT INTO Parmlist
SELECT DISTINCT x
  FROM VALUES (:a[1]), (:a[2]), (:a[3]), ..., (:a[n]) AS List(x)
 WHERE x IS NOT NULL;
```

9.3

第2 正規形（2NF）

テーブルが第1 正規形を満たし、かつ部分的関数従属を持たないとき、そのテーブルは第2 正規形（Second Normal Form：2NF）を満たす。部分的関数従属とは、列XとYが存在し、X がキーであり、かつ任意のZ がX の部分集合である場合、Z → Yとなることである。もっと直観的に言えば、第1 正規形を満たしたテーブルで、すべての非キー属性を決定するキーが存在するならば、第2正規形ということである。

もう一度、学校の時間割の例に戻ろう。いま、学生名（student_name）と講座名（course_name）がわかれば、クラス分類（class_section）と学年（student_grade）を決定できるとする（なぜなら、学生は同じ講座名について2 つ以上のクラス分類を登録できないからだ）。これはつまり、

(student_name, course_name) → (class_section, student_grade)

と同じ意味だ。

もう少しデータの分析を行うと、

(student_name → student_major)

という関係も存在することがわかる。というのも、学生は1つしか専攻を持てないからだ。student_nameは(student_name, course_name)キーの部分集合であるため、ここには部分的関数従

属が存在する！ したがって、このテーブルは第2 正規形を満たしていない。これを第2 正規形にするには、次のようにテーブルを分解する。

```
CREATE TABLE Classes
(course_name CHAR(7) NOT NULL,
 class_section CHAR(1) NOT NULL,
 time_period INTEGER NOT NULL,
 room_nbr INTEGER NOT NULL,
 room_size INTEGER NOT NULL,
 professor_name CHAR(25) NOT NULL,
     PRIMARY KEY (course_name, class_section));
```

Classes：クラステーブル　　列名は新しいものがないので省略

```
CREATE TABLE Enrollment
(student_name CHAR (25) NOT NULL,
 course_name CHAR(7) NOT NULL,
 class_section CHAR(1) NOT NULL,
 student_grade CHAR(1) NOT NULL,
     PRIMARY KEY (student_name, course_name));
```

Enrollment：受講テーブル　　列名は新しいものがないので省略

```
CREATE TABLE Students
(student_name CHAR (25) NOT NULL PRIMARY KEY,
 student_major CHAR(10) NOT NULL);
```

Students：学生テーブル　　列名は新しいものがないので省略

　これで第2正規形が完成した。すべての属性はテーブルの包括的なキーにのみ従属している。いま、もしある学生が専攻を変えたとする。その場合、変更は`Students`テーブルの`student_major` を1 行だけ変更すればよい。つまり、1 つの変更は1 か所だけで起きている。さらに、`Enrollment`テーブルに主キーを作成したことで、あ

る学生が同一の講座について、複数の登録を行うことはできないという制約も実現できた。しかし、これで問題が完全になくなったわけではない。

　気づいたかもしれないが、room_sizeはもちろんClasses テーブルの主キーに従属するが、この列はまた、room_nbrにも従属している。もしある講座の行われる部屋番号が変更されたら、room_sizeも変更しなければならない。そしてもしroom_nbrそのものが（ビルの改築などによって）変更されたら、Classesのその部屋のroom_sizeも変更しなければならない。このように、非キー属性であるroom_sizeにroom_nbrが従属するようなケースを、推移的関数従属と呼ぶ。

9.4

第3 正規形（3NF）

この問題を解決するために必要になるのが、第3 正規形である。第3 正規形（Third Normal Form：3NF）とは、あるテーブルが第2 正規形を満たし、X がキーであるか、またはY が候補キーの一部である場合に、常にX → Yが成立することである（候補キーとは、テーブルの各行を一意に特定できる列の集合である。逆に言うと、候補キーから1つでも列を取り除くと、一意性が破壊される）。つまり、上記のテーブル群が第3正規形ではなく第2正規形にとどまるのは、部分的関数従属が推移的関数従属の一形態にすぎず、推移的関数従属を完全に除去できていないからだ。非形式な言い方をすれば、本来非キー属性は、キー（全体）によって決定されるべきであり、キー以外の属性によって決定されてはならないのである。

第3正規形を満たす基準としては、通常「推移的関数従属がないこと」と説明される。だがこれは少し正確さを欠いた説明だ。推移的関数従属は、列(A, B, C)から成り、(A → B)かつ(B → C)が成り立つテーブルにおいて、(A → C)が導かれる場合を指す言葉である。だが、いま我々が見ている例の従属性は、(`course_name`, `class_section`) → (`room_nbr`)かつ(`room_nbr`) → (`room_size`)である。このケースでは、`room_nbr`はキーの一部である`course_name`にだけ従属しているため、単純な推移的関数従属ではない。だが、やはり原則的には同じだ。このサンプルを第3正規形にして問題を解決するには、次のような分解を行う。

```
CREATE TABLE Rooms
(room_nbr INTEGER NOT NULL PRIMARY KEY,
 room_size INTEGER NOT NULL);
```

Rooms：部屋　　列名は新しいものがないので省略

```
CREATE TABLE Classes
(course_name CHAR(7) NOT NULL,
 class_section CHAR(1) NOT NULL,
     PRIMARY KEY (course_name, class_section),
 time_period INTEGER NOT NULL,
 room_nbr INTEGER NOT NULL);

CREATE TABLE Enrollment
(student_name CHAR (25) NOT NULL,
 course_name CHAR(7) NOT NULL,
     PRIMARY KEY (student_name, course_name),
 class_section CHAR(1) NOT NULL,
 student_grade CHAR(1) NOT NULL);

CREATE TABLE Students
(student_name CHAR (25) NOT NULL PRIMARY KEY,
 student_major CHAR(10) NOT NULL);
```

関係理論についてのありがちな誤解は、第3 正規形のテーブルには推移的関数従属は存在しない、というものだ。しかし上でも説明したように、もしX → Yが成り立ち、かつYが候補キーの一部である場合、Xはキーである必要はない。実際、このテーブル群は第3正規形だが、まだ(room_nbr, time_period) → (course_name, class_section)という推移的関数従属が存在する。しかし従属する側の(course_name, class_section)は主キーである（当然、候補キーでもある）。したがって、これは理論的には第3正規形の要件を満たしている。

このテーブル構造において、まだ問題が顕在化するのは、1 つの room_nbr に対して複数のcourse_nameを割り当てることが可能な

点である。

推移的関数従属のもう1つのパターンは、計算列を使うケースだ。

```
CREATE TABLE Stuff
(width INTEGER NOT NULL,
 length INTEGER NOT NULL,
 height INTEGER NOT NULL,
 volume INTEGER NOT NULL
     CHECK (width * length * height = volume),
     PRIMARY KEY (width, length, height));
```

Stuff：物体テーブル　width：幅　length：長さ　height：高さ　volume：体積

体積（volume）列は残りの3 つの列から導かれる。したがって、3 つのどれかの値が変更されると体積も変更しなければならない。これは余談だが、計算列を使っておけばわざわざ計算し直す必要はない。

```
CREATE TABLE Stuff
(width INTEGER NOT NULL,
 length INTEGER NOT NULL,
 height INTEGER NOT NULL,
 volume AS (width * length * height),
     PRIMARY KEY (width, length, height));
```

CHECK 制約はエラーを返すだけだが、計算列を使えば、ユーザーは変更を意識しないまま整合性を維持できる。

9.5

基本キー正規形（EKNF）

基本キー正規形（Elementary Key Normal Form：EKNF）は、第3正規形を少し発展させたものだ。定義上、基本キー正規形のテーブルは第3 正規形である。基本キー正規形が現れるのは、2つ以上の一意な複合キーが存在し、かつそれらが重なりあっている場合である。こういうとき、重なっている列において冗長な情報が生じる。たとえば、次のテーブルにおいて、講座コード（course_code）は、教科（course_name）の一意な識別子でもあるとしよう。

```
CREATE TABLE Enrollment
(student_id INTEGER NOT NULL,
 course_code CHAR(6) NOT NULL,
 course_name VARCHAR(15) NOT NULL,
     PRIMARY KEY (student_id, course_name)
     -- , UNIQUE (student_id, course_code) 代理キー
);
```

Enrollment

student_id	course_code	course_name
1	CS-100	ER Diagrams
1	CS-114	Database Design
2	CS-114	Database Design

このテーブルは第3正規形だが、基本キー正規形ではない。このテーブルの主キーは(stude nt_id, course_name)の組み合わせだ。しかしまた、(student_id, course_code)という代理キーでも主

キーになりうる。結果として、このテーブルは更新や削除において異常が生じる【訳注8】。なぜなら、ある教科に対して、course_nameとcourse_codeの同じ組み合わせが複数行存在しうるからだ。

　基本キー正規形を満たすためには、次のようにテーブルを分解する。

```
CREATE TABLE Subjects
(course_code CHAR(6) NOT NULL PRIMARY KEY,
 course_name VARCHAR(15) NOT NULL);
```

Subjects：教科テーブル　　course_code：講座コード　　course_name：講座名

```
CREATE TABLE Enrollment
(student_id INTEGER NOT NULL,
 course_code CHAR(6) NOT NULL,
     PRIMARY KEY (student_id, course_code));
```

Enrollment：受講テーブル　　student_id：学生ID　　course_code：講座コード

　普通は正規化の過程においてテーブルが基本キー正規形を満たすかどうかを考慮することはあまりない。というのは、たいていの設計においては、第3正規形を満たしたあとは次に見るボイス–コッド正規形に移行してしまうからだ。したがって、ここで基本キー正規形について触れたのは、あくまで歴史的な正確さと完全さのためにすぎない。

9.6

ボイス－コッド正規形（BCNF）

ボイス–コッド正規形（Boyce-Codd Normal Form：BCNF）は、すべての関数従属（X → A）において、Xがテーブル全体のスーパーキーであるということだ。スーパーキーとは、テーブルの各行を一意に特定できる列集合である。ただし、スーパーキーから一部の列を取り除いても、なおテーブルの各行を一意に特定できるキーが残るような列集合である。要するに、「余計な」キーを含んでいるような一意キーである。ボイス–コッド正規形において、初めてすべての推移的関数従属が存在しないことが保証される。

関係理論では、ボイス–コッド正規形よりも「高次」の正規形も存在し、それらも有用かつ興味深いものではあるが、ここでの議論では省略する。我々の例においては、ボイス–コッド正規形にすることですべての重要な不整合を除去することができた。

第3 正規形はキーと非キーの関係に着目していた。だが、列はキーと非キーの両方でありうる。たとえば、セールスマンの給与を例に考えよう。このセールスマンの賞与額は、基本給（年俸）とセールスポイント（営業実績）に依存するだろう。一方、賞与支給のときにもらえる記念品は、基本給の範囲とセールスポイントに依存する。具体的に言うと、まだ新米セールスマンの基本給ランクは一番下（15, 000～20, 000ドル）で、セールスポイントは100である。彼がもらえる記念品は万年筆だ。しかし、これが達人セールスマンになると、基本給ランクは30, 000ドル～60, 000ドル、セールスポイントは200である。彼は記念品として車をゲットした。

この例における、関数従属性を次のように定式化しよう。基本給はsalary_amtである。

(salary_amt, points) → (gift)
(gift) → (points)

まずは、すべてのデータを放り込んだテーブルから始めて、これを正規化していこう。

Gifts

salary_amt	points	gift
15000	100	Pencil
17000	100	Pen
30000	200	Car
31000	200	Car
32000	200	Car

Gifts：記念品テーブル　　salary_amt：基本給　　points：セールスポイント
gift：記念品

このテーブルはすでに第3 正規形を満たしているが、まだ問題がある。対応する基本給が存在しないと、記念品やセールスポイントを登録できないのだ。また、任意のセールスポイントを削除すると、記念品と給与の情報も失われる（たとえば、セールスポイントが200の行を削除すると、基本給が30, 000ドルの人々が車を得られるとい

う情報が消えてしまう）。そして最後に、特定のセールスポイントに対応する記念品を変えようとすると、同じ基本給ランクの行すべてが影響を受けてしまう。これが、ボイス–コッド正規形を満たしていないテーブルにおける問題である。このテーブルは、次の2つのテーブルに分解する必要がある。

PayGifts

salary_amt	gift
15000	Pencil
17000	Pen
30000	Car
31000	Car
32000	Car

GiftsPoints

gift	points
Pencil	100
Pen	100
Car	200

PayGifts：給与記念品テーブル　　GiftsPoints：記念品ポイントテーブル
salary_amt：基本給　　gift：記念品　　points：セールスポイント

PayGifts テーブルの主キーは、(salary_amt, gift)で、GiftsPoints テーブルの主キーは、(gift)である。

9.7

第4 正規形（4NF）

第4正規形（Fourth Normal Form：4NF）は多値従属性がある場合に有用である。たとえば、部署（dept_name）、仕事（jobs）、およびその仕事で使う部品（parts）を管理するテーブルを考えよう。このテーブルに存在する多値従属性は次のようなものだ。

```
dept_name ↠ jobs
dept_name ↠ parts
```

いま、dept_name「d1」は仕事「j1」と「j2」および部品「p1」と「p2」を決定し、dept_name「d2」は仕事「j3」と「j4」と「j5」および部品「p2」と「p4」を決定するとする。dept_name「d3」は仕事「j2」と部品「p5」と「p6」を決定する。テーブルは次のようになる。

dept_name	jobs	parts
d1	j1	p1
d1	j1	p2
d1	j2	p1
d1	j2	p2
d2	j3	p2
d2	j3	p4
d2	j4	p2

d2	j4	p4
d2	j5	p2
d2	j5	p4
d3	j2	p5
d3	j2	p6

いま、ある`dept_name`に部品を追加しようとすると、複数行を登録せざるを得ない。同様に、仕事や部品を削除しようとすると、他の情報を破壊してしまう。部品や仕事の名前を更新する場合でも、複数行が影響を受ける。こうした問題を解決するには、テーブルを次のような2つに分解することである。

(`dept_name`, `jobs`)
(`dept_name`, `parts`)

第4 正規形の定義は、テーブル内に多値従属性を持たないことである。また、もしテーブルが第4正規形を満たしていれば、ボイス–コッド正規形も満たしている。

9.8

第5 正規形（5NF）

第5正規形（Fifth Normal Form：5NF）は、別名を結合射影正規形または射影結合正規形と呼ぶ。この名称は、無損失結合や結合射影異常の概念に由来する。この問題は、n方向（n >2）の関連が存在する場合に生じる。

第5 正規形と、それが解決する問題について、たとえば住宅ローンの情報を管理するテーブルを考えよう。テーブルには、買い手（`buyer_name`）、売り手（`seller_name`）、貸し手（`lender_name`）が含まれる。候補キーは`(buyer_name, seller_name, lender_name)`である。

Mortgages

buyer_name	seller_name	lender_name
Smith	Jones	National Bank
Smith	Wilson	Home Bank
Nelson	Jones	Home Bank

Mortgages：住宅ローンテーブル　buyer_name：買い手　seller_name：売り手　lender_name：貸し手

このテーブルには3方向の関連が存在する。これは第5正規形のテーブルだが、これを3つの2方向の関連に分解すると問題が発生する。テーブルは次のようになる。

BuyersLenders

buyer_name	lender_name
Smith	National Bank
Smith	Home Bank
Nelson	Home Bank

BuyersLenders：買い手－貸し手テーブル　buyer_name：買い手　lender_name：貸し手

SellersLenders

seller_name	lender_name
Jones	National Bank
Wilson	Home Bank
Jones	Home Bank

SellersLenders：売り手－貸し手テーブル　seller_name：売り手　lender_name：貸し手

BuyersSellers

buyer_name	seller_name
Smith	Jones
Smith	Wilson

Nelson	Jones

BuyersSellers：買い手－売り手テーブル　buyer_name：買い手　seller_name：売り手

これによって、貸し手が未定でも買い手と売り手を登録することができるし、残りの関連についても同様のことができるようになる。困るのは、この3 つのテーブルを結合して、オリジナルの情報を復元しようとする場合だ。この結合SQLは次のようになる。

```
SELECT BS.buyer_name, SL.seller_name, BL.lender_name
  FROM BuyersLenders AS BL, SellersLenders AS SL, BuyersSellers AS BS
 WHERE BL.buyer_name = BS.buyer_name
   AND BL.lender_name = SL.lender_name
   AND SL.seller_name = BS.seller_name;
```

結果

```
buyer_name  seller_name  lender_name
=====================================
Nelson      Jones        Home Bank
Smith       Jones        National Bank
Smith       Jones        Home Bank     ←偽物の行
Smith       Wilson       Home Bank
```

たしかに、これでオリジナルの行、たとえば`('Smith', 'Jones', 'National Bank')`を復元できる。だが困ったことに、もともとのテーブルには存在しなかった偽物の行`('Smith', 'Jones', 'Home Bank')`まで現れてしまうのだ。これが結合射影異常と呼ばれる現象だ。

結合射影正規形には、強い結合射影正規形などいくつかのバリエーションが存在する。そこでのキーとなる概念が結合従属性であ

る。残念なことに、あるテーブルが結合射影正規形や第4正規形であるかどうかを判別する機械的な方法はない。というのは、この問題はNP完全であることが知られており、この問題に含まれる要素の数が増えれば増えるほど、問題を解くのに必要なリソースが急速に増え、一般解を見つけることが難しいからだ。

これは余談だが、第4正規形をCASEツールで扱うのはよく行われている。たいていのCASEツールは、すべてのテーブルが第3 正規形であるようなスキーマを出力してくれる。ツールは、関数従属性をER図や、実データの統計的解析から抽出し、その情報をテーブルに付加することで正規形かどうかをチェックしている。ただ残念なことに、1つの関数従属性の集合から得られる第3正規形スキーマは複数存在しうるが、多くのER図を作成するCASEツールは、そのうちの1つを見つけたら終了し、すべての解を出力しない。だが、もしORM（Object Role Model）ツールを適切に利用すれば、スキーマは第5正規形になる。このテクニックについては、テリー・ハルピンの著作【訳注9】を読むことを強く推奨する。

9.9

ドメイン－キー正規形（DKNF）

1981 年、ロナルド・フェイギンはドメイン–キー正規形（Domain-Key Normal Form：DKNF）を定義した。これは、すべてのドメイン制約と関数従属が有効になっているスキーマのことだ。与えられた制約の集合に対して、ドメイン–キー正規形を自動的に生成する一般的アルゴリズムは存在しない。だが、個別の事例についてドメイン–キー正規形かどうかをチェックすることはできる。それには、現実世界でDDLを書いてみるのがわかりやすい。

ここで少し話を戻して、正規化の土台になっている数学的モデルを見てみよう。関数従属性には、正規化の問題を解くのに使うことのできる公理がある。これらの公理は6個あり、アームストロングの公理系と呼ばれている。

1. **反射律**： $X \rightarrow X$
2. **増加律**：もし $X \rightarrow Y$ ならば $XZ \rightarrow Y$
3. **合弁律**：もし($X \rightarrow Y$ かつ $X \rightarrow Z$)ならば $X \rightarrow YZ$
4. **分解律**：もし $X \rightarrow Y$ かつ $Z \subseteq Y$ ならば $X \rightarrow Z$
5. **推移律**：もし($X \rightarrow Y$ かつ $Y \rightarrow Z$)ならば $X \rightarrow Z$
6. **擬推移律**：もし($X \rightarrow Y$ かつ $YZ \rightarrow W$)ならば $XZ \rightarrow W$

集合論の公理系とよく似ているので、見ただけでもその意味は十分に理解できるだろう。現実の世界では、関数従属性はビジネスルールである。我々は、それを数学モデルに変換しなければならない。第3 正規形を導く正規化アルゴリズム（P. A.バーンスタインが 1976 年に発展させた）においては、冗長な関数従属を除去するため

に上の公理を使うことができる。たとえば、次のような関数従属が与えられているとする。

A → B
A → C
B → C
DB → E
DAF → E

ここで、A → Cは冗長である。なぜなら、A → BとB → Cおよび推移律から導けるからだ。同様に、DAF → Eも冗長だ。これはDB → EとA → Bおよび推移律（これでDA → Eがわかる）と増加律（これでDAF → Eがわかる）で導ける。我々が見つけたいのは、与えられたすべてのルールをカバーできる関数従属性の極小集合である。この集合は極小被覆（nonredundant cover）と呼ばれる。極小被覆は複数存在しうるが、前出の関数従属に対しては、たとえば次のようになる。

A → B
B → C
DB → E

こうしてみると、バーンスタインのアルゴリズムは、A、BおよびDBをそれぞれキーとするテーブルを作ることで、与えられた関数従属を満たすスキーマを作れる、ということを示したものだ。ここまでは、簡単な例を使って説明したが、難しいのはここからだ。

複数の第3 正規形テーブルのスキーマを扱う例として、DBStar Corporation（現在はEvoke Software）がデモンストレーションで使ったサンプルを借用しよう。同社は、この問題を自社のCASEツールの機能アピールに使っていた。

いま、仮想上の、かつ単純化した航空会社がフライトとパイロットのスケジューリングを行うデータベースを持っているとしよう。関連の大半は自明のものだ。飛行機の便は出発時刻と目的地しか持っていない。便ごとに別々のパイロットが割り当てられることもあるし、日によって異なるゲートから出発する。こうした関数従属性を定式化するとこうなる。

1） `flight → destination`

2） `flight → hour`

3） `(day, flight) → gate`

4） `(day, flight) → pilot`

5） `(day, hour, pilot) → gate`

6） `(day, hour, pilot) → flight`

7） `(day, hour, pilot) → destination`

8） `(day, hour, gate) → pilot`

9） `(day, hour, gate) → flight`

10） `(day, hour, gate) → destination`

理論肌の人は、この大量の関数従属性の集まりを見て、そのリストに含まれる冗長性にうんざりするだろう。しかし現実世界で人々と接して生きる限り、顧客は決して最小の要件に絞って教えてくれ

るということなどありえない。人々は同じことを繰り返すし、データを自分の立場からしか見ていない。彼らの目からはあまりに明らかで、話す価値などないと判断した情報を省いてしまうこともよくある。

　問題は、これらの関数従属性を満たすような第3 正規形以上のスキーマを見つけることだ。注意点は、解の中には、第3正規形は満たしていても、なおいくつかの関数従属性を満たしていないパターンもあることだ。たとえば、次のようなテーブルはうまくいかない。

```
CREATE TABLE PlannedSchedule
(flight, destination, hour, PRIMARY KEY (flight));
```

PlannedSchedule：予定スケジュール　flight：便　destination：目的地　hour：時間

```
CREATE TABLE ActualSchedule
(day, flight, gate, pilot, PRIMARY KEY (day, flight));
```

ActualSchedule：実スケジュール　day：日　flight：便　gate：ゲート　pilot：パイロット

PlannedSchedule

flight	hour	destination
118	17:00	Dallas
123	13:00	Omaha
155	17:00	Los Angeles

171	13:00	New York
666	13:00	Atlanta

ActualSchedule

day	flight	pilot	gate
Wed	118	Tom	12A
Wed	155	Tom	13B
Wed	171	Tom	12A
Thu	123	John	12A
Thu	155	John	12A
Thu	171	John	13B

いま、関数従属性のいくつかに合弁律を適用すると、次のような従属性が得られる。

```
(day, hour, gate) → (destination, flight, pilot)
(day, hour, pilot) → (destination, flight, gate)
```

これは、もし特定の曜日、時間、ゲートが与えられれば、一意な便を決定できるということである。そしてまた、特定の曜日、時間、パイロットが与えられれば、やはり一意な便を決定できる。

`PlannedSchedule`および`ActualSchedule`を使って考えると、上の2つの制約を満たすようなビューを作ることは、端的に言ってで

きない。もし「パイロットXがY日のZ時に搭乗する便は何か？」というクエリが2つ以上の答えを持つ場合、関数従属性を破ることになるし、何より常識に反する。このサンプルスキーマは、我々の望む制約を与えるには望ましくない。

この2つの制約が意味することは、我々が、以下のクエリに対する答えが1つに決まり、かつ、データを挿入したり削除しても情報が失われない、ということである。

（1）木曜の13時に12Aゲートから出発する便は？

この問いに答えられることは、システムとして当然の要件だと思うだろう。だが、ActualSch edule テーブルには、PlannedSchedule に存在しない便を追加することもできる。こうした便をActualSchedule テーブルから特定することはできない。

（2）木曜の13時に12Aゲートから出発する便に搭乗するパイロットは？

このクエリは（1）と同じ問題を持っている。つまり、ActualSchedule テーブルには、必ずしもパイロットの情報は登録する必要がないので、パイロットの情報がNULL の便については、この問いに答えられない。

（3）クエリ1および2の目的地は？

これも同様である。

（4）木曜の13時にジョンが出発するゲートは？

（5）水曜の17時にトムが向かう目的地は？

（6）水曜の17時にトムが乗る便は？

この問題を考えるには、関数従属性の1 つがいかにして関数従属の公理から導かれるかを――ちょうど幾何学の証明を見るように――見てみると、ヒントになるかもしれない。

結果

1） (day, hour, gate) → (pilot)

2） (day, hour, pilot) → (flight)

以下を証明する

(day, hour, gate) → (flight)

証明

3） (day, hour) → (day, hour)；**反射律**

4） (day, hour, gate) → (day, hour)； **3および増加律**

5） (day, hour, gate) → (day, hour, pilot)； **1、4および合弁律**

6） (day, hour, gate) → (flight)； **2、5および推移律**

Q.E.D

この証明は、個々の関数従属性を、別の関数従属性から導いている。それぞれの証明は短いが、どのような関数従属性を導くかに応じて、異なる関数従属性を「前提」とする必要がある。

以下に、この問題において10 個の異なる関数従属を導く証明を示す。どの証明においても、すべての妥当な証明ステップと関数従属の計算を含んでいる。唯一、公理には含まれていなかった操作で、これらの証明で使われているのは、左項削減（left reduction）である。左項削減は、もしXX → YならばX → Yということで、要するに冗長性排除である。非常に単純なので、公理に含まれていてもよさそうだが、これが公理に含まれていないのは、他の公理から導出できるからだ（興味のある人は、自分で証明してみてもらいたい）。

証明

(day, hour, pilot) → (gate)

a） (day) → (day); 反射律

b） (day, hour, pilot) → (day); 増加律**(a)**

c） (day, hour, pilot) → (day, flight); 合弁律**(6, b)**

d） (day, hour, pilot) → (gate); 推移律**(c, 3)**

Q.E.D.

証明

(day, hour, gate) → (pilot)

a） (day → day); 反射律

b） (day, hour, gate) → (day); 増加律**(a)**

c） (day, hour, gate) → (day, flight); 合弁律**(9, b)**

d） (day, hour, gate) → (pilot); 推移律**(c, 4)**

Q.E.D.

証明

(day, flight) → (gate)

a） (day, flight, pilot) → (gate); 擬推移律**(2, 5)**

b） (day, flight, day, flight) → (gate); 擬推移律**(a, 4)**

c） (day, flight) → (gate); 左項削減**(b)**

Q.E.D.

証明

(day, flight) → (pilot)

a） (day, flight, gate) → (pilot); 擬推移律**(2, 8)**

b） (day, flight, day, flight) → (pilot); 擬推移律**(a, 3)**

c） (day, flight) → (pilot); 左項削減**(b)**

Q.E.D.

証明

(day, hour, gate) → (flight)

a） (day, hour) → (day, hour); 反射律

b） (day, hour, gate) → (day, hour); 増加律**(a)**

c） (day, hour, gate) → (day, hour, pilot); 合弁律 **(b, 8)**

d） (day, hour, gate) → (flight); 推移律**(c, 6)**

Q.E.D.

証明

(day, hour, pilot) → (flight)

a） (day, hour) → (day, hour); 反射律

b） (day, hour, pilot) → (day, hour); 増加律**(a)**

c） (day, hour, pilot) → (day, hour, gate); 合弁律 **(b, 5)**

d） (day, hour, pilot) → (flight); 推移律**(c, 9)**

Q.E.D.

証明

(day, hour, gate) → (destination)

a） (day, hour, gate) → (destination); 推移律**(9, 1)**

Q.E.D.

証明

(day, hour, pilot) → (destination)

a） (day, hour, pilot) → (destination); 推移律**(6,**

1)

Q.E.D.

さて、これで10 個の関数従属のうち、8 個をどうやって他の関数従属から導くかを例示してみせた。要は、次の3 つの基準を満たすように、複数の関数従属を混ぜ合わせたり足し合わせたりするのである。

基準1　各属性は、少なくとも1つの関数従属の左項か右項に現れていなければならない。

基準2　もし与えられた関数従属が集合の中に含まれていれば、それを導くために必要なすべての関数従属は、集合に含まれていてはならない。

基準3　もし与えられた関数従属が集合に含まれていなければ、それを導くために使われた関数従属は集合に含まれている必要がある。

この手続きによって、「極小被覆」の集合を得ることができる。この集合は、試行錯誤や常識によっても発見できるが、機械的手続きで求めることもできるのだ。たとえば、もし`(day, hour, gate) → flight`という関数従属を除外すると、`(day, hour, gate) → pilot`を含めなければならない。逆も同様である。なぜなら、両者はそれぞれ互いの導出において使われているからだ。もちろん自分で導出を最後まで行うことは可能だが、次からはCASEツールを使うなどの方法で機械的にやりたいと思うだろう。

極小被覆を得るためのアルゴリズムは、基本的にはすべての関数従属の集合の組み合わせを生成するものだ。ただし、`(flight → destination)`および`(flight → hour)`は生成されない（これらを他の関数従属性から導くことはできないからだ）。このように考えると、組み合わせの数は(2^8)、つまり256 通りの関数従属の組み合わせが得られる。そのあとに、各組み合わせについて、基準を満たしているかチェックしていく。

ありがたいことに、この退屈な作業は簡単なスプレッドシート1枚で自動化できる。この問題では、基準1は15個の集合を除外するにすぎない。だが基準2は152個の集合をふるい落とす。さらに基準3は67個。最後まで残るのは22の被覆である。そのうちの5個が我々の探していたものだ。

5 個の極小被覆

集合I

```
(flight) → (destination)
(flight) → (hour)
(day, hour, gate) → (flight)
(day, hour, gate) → (pilot)
(day, hour, pilot) → (gate)
```

集合II

```
(flight) → (destination)
(flight) → (hour)
(day, hour, gate) → (pilot)
```

(day, hour, pilot) → (flight)

(day, hour, pilot) → (gate)

集合III

(flight) → (destination)

(flight) → (hour)

(day, flight) → (gate)

(day, flight) → (pilot)

(day, hour, gate) → (flight)

集合IV

(flight) → (destination)

(flight) → (hour)

(day, flight) → (gate)

(day, hour, gate) → (pilot)

(day, hour, pilot) → (flight)

集合V

(flight) → (destination)

(flight) → (hour)

(day, flight) → (pilot)

(day, hour, gate) → (flight)

(day, hour, pilot) → (gate)

(day, hour, pilot) → (flight)

ここでは、同じ左項を持つ関数従属に対して合弁律を適用し、左項をキーとしてそれぞれをグループ化したテーブルを作っている。

我々はまた、対称的な関数従属を除外する（対称的とは、X → Yかつ Y → Xというパターンだ。これをX ⇔Yのように表記する）。

　こうして作られるスキーマは、少なくとも第3正規形を満たす。上記の5つの集合を、データ型を省略してSQLのDDLで表現するとこうなる。

解 1

```
CREATE TABLE R1 (flight, destination, hour,
     PRIMARY KEY (flight));

CREATE TABLE R2 (day, hour, gate, flight, pilot,
     PRIMARY KEY (day, hour, gate),
     UNIQUE (day, hour, pilot),
     UNIQUE (day, flight),
     UNIQUE (flight, hour));
```

解 2

```
CREATE TABLE R1 (flight, destination, hour,
     PRIMARY KEY(flight));

CREATE TABLE R2 (day, flight, gate, pilot,
     PRIMARY KEY (day, flight));

CREATE TABLE R3 (day, hour, gate, flight,
     PRIMARY KEY (day, hour, gate),
     UNIQUE (day, flight),
     UNIQUE (flights, hour));

CREATE TABLE R4 (day, hour, pilot, flight,
     PRIMARY KEY (day, hour, pilot));
```

解 3

```
CREATE TABLE R1 (flight, destination, hour, flight
     PRIMARY KEY (flight));

CREATE TABLE R2 (day, flight, gate,
     PRIMARY KEY (day,flight));

CREATE TABLE R3 (day, hour, gate, pilot,
     PRIMARY KEY (day, hour, gate),
     UNIQUE (day, hour, pilot),
     UNIQUE (day, hour, gate));

CREATE TABLE R4 (day, hour, pilot, flight
     PRIMARY KEY (day, hour, pilot),
     UNIQUE(day, flight),
     UNIQUE (flight, hour));
```

解 4

```
CREATE TABLE R1 (flight, destination, hour,
     PRIMARY KEY(flight));

CREATE TABLE R2 (day, flight, pilot,
     PRIMARY KEY (day,flight));

CREATE TABLE R3 (day, hour, gate, flight,
     PRIMARY KEY (day, hour, gate),
     UNIQUE (flight, hour));

CREATE TABLE R4 (day, hour, pilot, gate,
     PRIMARY KEY (day, hour, pilot));
```

これらの解を一見しただけでも、「なんだこの雑然としたテーブル群は」という感想を持つだろう。だがこれらは、れっきとした第3正規形なのだ！ これらよりもっと良い解があるだろうか？ 1つはボイス–コッド正規形で、2つのテーブルしか含まないものが考えられる。これはクリス・デイトが提案した（Date, 1995:224）。

```
CREATE TABLE DailySchedules (flight, destination, hour
PRIMARY KEY (flight));
CREATE TABLE PilotSchedules (day, flight, gate, pilot,
PRIMARY KEY (day, flight));
```

これは実用に堪えるスキーマだ。このスキーマに、さらに制約を追加することもできる。スケジュールはあまり変更されないと想定すれば、制約は、パフォーマンス改善と、より詳細なエラーメッセージを表示することに役立つだろう。

```
CREATE TABLE DailySchedules
(flight, hour, destination,
     UNIQUE (flight, hour, destination),
     UNIQUE (flight, hour),
     UNIQUE (flight));

CREATE TABLE PilotSchedules
(day, flight, day, hour, gate, pilot,
     UNIQUE (day, flight, gate),
     UNIQUE (day, flight, pilot),
     UNIQUE (day, flight),
     FOREIGN KEY (flight, hour) REFERENCES R1(flight, hour));
```

9.10

正規化を行うにあたっての実践的なヒント

CASEツールは正規化を形式的に行う方法を実装している。特にER図を描く機能は非常に有用だ。とはいえ、形式的なやり方は、時間がかかることも多い。そこで、時間を節約するための、ちょっと非形式的なやり方についてお教えしよう。

一般に、テーブルはエンティティか、エンティティ間の関連を表現している。そうでなければ補助テーブルだ。テーブルの分類にこの2 種類しかないのが、ER 図が設計ツールとしてうまく動作する理由だ。なお、補助テーブルはER図には描かれない。なぜならこれらは関数やマッピングといった、宣言的計算モデルを補助するためだけに存在しており、現実世界のエンティティに対応していないからだ。

エンティティを表すテーブルには、シンプルで直接的な名前を与えるべきだ。見ればその内容がわかるような、わかりやすい名前だ。`Students`という名前のテーブルには、学生のデータが入っているべきで、かつそれ以外のデータは入っていてはいけない。つまり、学生のボウリングのスコアまで入れてはいけないのだ。

テーブルの名前に複数形や集合名詞を使うのも良い考えだ。これによって、テーブルが常にエンティティの集合、つまり行はエンティティの1つのインスタンスであることを意識できる。

1 対1または1 対多の関連を表すテーブルは、その内容を表す名前を付けるべきだ。そして、その名前はできる限り短くすべきだ。たとえば、`Students`テーブルと`Classes`テーブルは、第3の（関連）テーブルによって関連付けられる。このテーブルは関連のみを表現

している。学位を得る唯一の手段は、授業に出席することである。したがって、関連テーブルは両テーブルを参照する複合キーを持つことになる。この関連テーブルは、たとえばReportCards とかGradesとか、そんな名前を付けるべきだ。多対多の関連に基づいて、2 つのテーブルの名前を結合したエンティティ名を採用するのは、避けるべきだ。たとえば、Students_Coursesは、出席を管理するエンティティとしては安直だが、良くない名前だ。

常に可能な限りNULLを避けること。もしテーブルがあまりに多くのNULLを許可した列を持っていると、適切に正規化されていない可能性が高い。NULLを使ってもよいのは、いま現在、値が失われているが、いずれ値が判明する場合においてのみである。もっとよいやり方は、失われた値をその列のエンコーディングスキーマに含めておくことだ（たとえば、コード「999」を割り当てるなど）。この点については、拙著『SQL Programming Style』（Morgan Kaufmann、2005）で論じている。

一般に、正規化されたデータベースでは、1つのテーブルあたりの列数は少なくなるが、テーブル数は多くなる。だが、これは正常なことなので、たとえテーブル数が増えてもあわてないでほしい。最初に（特に磁気テープを使っていた頃に）ファイルシステムで仕事をした人は、1 つのアプリケーションには、1 つの巨大なファイルを対応させるような設計をしがちだ。昔はこのやり方にも合理性があった。実際、もし当時、ファイルを細かく分けて管理していたら、多くの小さなファイルを結合するたびに磁気テープをマウント／アンマウントしなければならず、とても効率的とは言えるものではな

かった。この頃の設計慣習は、ディスクシステムの時代にも引き継がれた。なぜなら、手続き型言語は、データベースに対してもシーケンシャルなファイルシステムに対するのと同じやり方でアクセスしていたからだ。

同じ非キー属性が2 つ以上のテーブルに現れるのも、正規化に問題がある兆候だ。ある属性を決定するキーは、1つのテーブルにだけ存在せねばならない。したがって、同じ属性が複数のテーブルに現れるということは、キーも関連する諸テーブルから参照されることになる。

もっとも、これは確実にそう言えるわけではなく、ただのガイドラインだと思ってほしい。実際の話、同じ属性が違う名前で複数のテーブルに存在することはよくある。その場合は、名前を変更して統一しなければならない。`date_of_birth`、`birthdate`、`birthday`……。これらは皆社員の同じ属性を表している。ISO 11179では、ネーミングルールのガイドラインが決められている。この点についても『SQL Programming Style』で論じているので、参考にしてほしい。

9.11

キーのデータ型

論理的にせよ物理的にせよ、テーブルに対するキーはその機能と作られ方によって表9.1のように分類できる。それぞれを詳細に見ていこう。

表9.1　キーの分類

	自然キー	人工的なキー	露出した物理的ロケータ	サロゲートキー
外部の属性から作られている	Y	N	N	Y
外部で検証可能	Y	N	N	N
それ自身で検証可能	Y	Y	N	N
ユーザーに見える	Y	Y	Y	N

9.11.1　自然キー

自然キーは、テーブルに存在する属性の部分集合（これらは実際にユーザーが目にすることができる）であり、一意識別子として機能する。また、キー値の妥当性についてテーブルの外部に存在するソースを参照することで、妥当性を検証できる。例としては、商品に付いているバーコードがある。

経験の浅いエンジニアがよく抱く心配事に、複合的な自然キーは非常に長くなるのではないかというものがある。その問いに対する私の答えはこうだ。長いことが何か問題なのかい？ いまや21世紀である。キーサイズが物理的問題を引き起こしていた1950年代ではないのだから、コンピュータの処理性能をもっと信じてほしいものだ。処理速度を向上させるために、たかだか2、3列の整数型の複合キーを人間の読めないGUIDで代用しようとすることは、純粋にバグのリスクを増やすだけである。私は、緯度と経度の組み合わせが正しく位置を示しているかを検証する方法は知っているが、それに割り当てられたGUID が正しいかどうかをどうやって判断すればいいかは知らない。

また、長いキーは必ずしもパフォーマンスにとって悪いわけではない。たとえば、`(city, state)`をキーにしたとしよう。多くのシステムでは、単純に考えて`city`列だけにインデックスが付与されているだろう。このキーには、さらに列を追加してスーパーキーにすることができる。もしそうしたスーパーキーが有効なインデックスを提供してくれるならば、悪い話ではない（たとえば、クエリでアクセスされる列すべてを含むインデックスを作ることで、インデックスオンリースキャンが可能になり、テーブルへのアクセスを省略できる）。

9.11.2　人工的なキー

人工的なキーは、本来テーブルには存在しない「余計な」列を追加してキーにする手段だ。このキーはユーザーからは見えない。テーブルの外部には存在せず、その構文や値の妥当性を検証する手段

もない。一例は、UPC/EAN スキーマにおけるオープンコードである。これは、ユーザーが自分の管理する物品に自由に割り当てられる。チェックディジットで検証することはできるが、あくまでその業務内での妥当性にとどまる。

経験豊かなデータベース設計者は、UPC/EAN、VIN、GTIN、ISBNといった業界標準コードをキーとして使うよう注意する。彼らはデータを外部の現実と突き合わせて検証する手段が必要であることを知っている。そのため、信頼できる外部の情報源が必要なのだ。なぜこのVINがこの車に割り当てられたかは検証すればわかるが、なぜ42という自動採番された番号がこの車に割り当てられたかはわからない。モデリング対象となっている現実に当たって、妥当性を検証することは非常に大事だ。それは、車をその駐車場の番号で位置を特定するのと同じぐらい意味のあることだ。

9.11.3　露出した物理的ロケータ

露出した物理的ロケータは、データモデルにおける属性には基づいていないが、ユーザーに見えるようになっている。そのため、次にどのような値が割り当てられるか事前に知る方法はないし、値の妥当性を検証することもできない。システムはこのロケータの値を、何らかの物理的なプロセスを通して割り当てるが、それは論理的なデータモデルとは一切何の関係もない。ユーザーがこれを変更しようとするには、データ要素同士の関係を壊さざるを得ない。こうしたロケータの例は、物理的な行の位置を数値、文字列、あるいは適当な実装依存のデータ型にエンコードしたものだ。もしそのエンコーディングのハッシュテーブルがSQL からアクセス可能なら

ば、ロケータの値を検証することも可能だが、普通はユーザーから隠されている。

　多くのプログラマは、オートナンバリング機能もこうしたロケータと同一カテゴリに属するものと考える。数値を物理的な位置に変換するには、ハッシュテーブルや読み取り書き込みヘッダーをディスクドライブ上に配置するよりは、むしろ検索を必要とするが、考え方は同じだ。ハードウェアが教えてくれるのは、物理的位置にたどりつく方法だが、その位置は論理的なデータモデルとは何の関係もないし、物理的に変更したり、外部の情報源によって検証することはできない。長い間、露出した物理的ロケータは、シーケンシャルファイルのレコード番号を擬似的に表現する方法として使われてきた歴史がある。それで、1960年代にISAMファイルをSQL で使っていた頃には、物理的ストレージの位置を参照することができたのだ。このように物理層との結び付きが強いと、抽象的なデータモデルやSQLの集合指向プログラミングの利点は根こそぎ失われる。余計なデータを保持せねばならず、コードの移植性は台無しだ。

　初期のSQL製品は、当時すでに存在していたファイルシステムの上に作られた。データは物理的に連続したディスクのページ内に保持され、物理的に連続した行内の物理的に連続した列から構成されていた――手短に言えば、パンチカードや磁気テープと大差なかった。多くのプログラマが、いまでもその時代の心理的モデルをひきずっている。筆者がファイルVSテーブル、行VSレコード、列VSフィールドといった対立構図を繰り返し指摘するのも、心にこびりついた古いモデルを払い落としたいからだ。

物理的に連続したストレージは、リレーショナルデータベースを作る方法の1つにすぎないし、最良のアイデアというわけでもない。リレーショナルデータベースの基本理念は、データがどのように、どこに格納されているかをユーザーが知る必要はないし、特定のハードウェア、特定の製品、特定のリリースバージョンといった物理的制約に依存しないコーディングが可能であるべきだというものだ。この点は、第4章で`IDENTITY`列について見た際も論じたことだ。

最後に、お前の言うことはどうも信用できないという人のために、コッドの言葉を引用して権威に訴えておこう（Codd, 1979）。

> データベースユーザーは、システムの利用を通じてサロゲートキーの生成や削除を行うかもしれない。だが、データベースユーザーはその値についてはコントロールしないし、そもそも値がユーザーの目に触れることはない。

これはつまり、サロゲートキーはインデックスのように振る舞うべきだ、ということである。作るのはユーザーだが、管理するのはシステムである。そして絶対にユーザーには見えない。コードやDRI、その他ユーザーの書くすべてのものの中にインデックスが現れることはないように。

コッドはまた次のようにも述べている。

ユーザーが管理するキーを永続的なサロゲートキーとして採用することには3つの難点がある。

1. ユーザーが管理するキーの値は、ユーザーによって決められるわけで、それはユーザーによって変更可能でなければならない（たとえば、もし2つの企業が合併する場合、2つの社員データベースは統合されるだろう。その結果、社員の一部または全員の社員IDは変更を余儀なくされる）。
2. 2 つの関係が、それぞれ異なるドメインで定義されたユーザー管理キーを持つことがありうる（たとえば、あるユーザーには社会保障番号が使われ、別のユーザーには社員IDが使われる、というような）。だが、どちらのエンティティも同じ社員を指し示している。
3. ユーザー管理キーに値が付与される前あるいは値を失ったあとであっても、エンティティについての情報を保持することが必要になるかもしれない（たとえば、仕事への応募者や退職者。前者はまだ社員IDを付与されていないし、後者の社員IDは削除されるだろう）。

こうした難点からわかる重要な結論は、共通のキー値に対する等値結合が、必ずしも共通のエンティティに対する結合と同じ結果になるわけではない、ということだ。この問題に対する解決は、システムによって割り当てられるサロゲートキーのエンティティドメインを導入することだ。データベースユーザーは、システムの利用を通じてサロゲ

ートキーの生成や削除を行うかもしれない。だが、彼らはその値についてはコントロールしないし、そもそも値がユーザーの目に触れることはない。

9.12

非正規化に対する実践的なヒント

非正規化について議論することは、宗教戦争を経験したい人にとっては最高の方法だ。一方の極には、リレーショナル原理主義者がいる。彼らはデータベース設計は最低でも第5 正規形を満たすべきだと頑なに信じており、それ以下の正規形に甘んじることは「自然に反する罪」だと考えている【訳注10】。もう一方の極には、ALTER文でデータベース全体にわたって列を追加削除し、スキーマの安定性など考えない人々がいる。

非正規化を実施する理由はパフォーマンスである。完全に正規化されたデータベースは、ユーザーが望む形でのデータを得るために非常に多くの結合を必要とする。結合はかつて、レスポンスタイムとマシンリソースの両方の点で、非常にコストが高かった。そのため、非正規化して、結合を「事前に済ませておく」ことで、時間とリソースを大きく節約できたのだ。だが今日、ハードウェアとソフトウェアは非常に進歩した。

もしビューが頻繁にアクセスされるものであれば、実体化してインデックスを付与しておく方法もある。今日では、非正規化してよいのはデータウェアハウスだけである。決してOLTPシステムで非正規化してはならない。非正規化したスキーマでは、データ整合性を保つために余分なプロシージャが必要になる。これはOLTPシステムでは割に合わない。

この点について、実際の問題を考えてみよう。数年前、コンピュサーブのOracleフォーラムに投稿された質問だ。ある製薬会社が薬の

在庫テーブルと薬価の履歴テーブルを持っており、薬は米国食品医薬品局が決めたNDC（National Drug Code）で管理する。

```
CREATE TABLE Drugs
(ndc CHAR(11) NOT NULL PRIMARY KEY,
 drug_name VARCHAR(30) NOT NULL,
 drug_qty INTEGER NOT NULL
     CONSTRAINT positive_quantity
     CHECK(drug_qty >= 0),
 ...);
```

Drugs：薬品テーブル　ndc：薬品コード　drug_name：薬品名　drug_qty：分量

```
CREATE TABLE Prices
(ndc CHAR(11) NOT NULL,
 start_date DATE NOT NULL,
 end_date DATE NOT NULL
     CONSTRAINT started_before_ended
     CHECK(start_date <= end_date),
 drug_price DECIMAL(8,2) NOT NULL,
     PRIMARY KEY (ndc, start_date));
```

Prices：価格履歴テーブル　ndc：薬品コード　start_date：開始日　end_date：終了日　drug_price：薬価

注文された時点の薬価を知るには、注文日をキーに検索する。現在の薬価の行は、終了日（end_date）に「無限大」を意味する値（'9999-12-31'のようなダミー値やNULL）を持つ。ある薬価について、(end_date + INTERVAL '1' DAY)という日付は、次の薬価の開始日付と同じになる。

これは正規化されたテーブルだが、パフォーマンスはよくない。あらゆる帳票や明細を出力するたびに、DrugsとPricesをテーブル

結合する必要がある。結合を回避するには、Drugsテーブルに列を追加すればよい。

```
CREATE TABLE Drugs
(ndc CHAR(11) PRIMARY KEY,
 drug_name VARCHAR(30) NOT NULL,
 drug_qty INTEGER NOT NULL
     CONSTRAINT positive_quantity
     CHECK(drug_qty >= 0),
 current_start_date DATE NOT NULL,
 current_end_date DATE NOT NULL,
     CONSTRAINT current_start_before_ended
     CHECK(current_start_date <= current_end_date),
 current_drug_price DECIMAL(8,2) NOT NULL,
 prior_start_date DATE NOT NULL,
 prior_end_date DATE NOT NULL,
     CONSTRAINT prior_start_before_ended
     CHECK(prior_start_date <= prior_end_date
       AND current_start_date = prior_end_date + INTERVAL '1' DAY),
 prior_drug_price DECIMAL(8,2) NOT NULL,
 ...);
```

このテーブルは、実務において95%の注文をカバーできるだろう。なぜなら、在庫から出荷される前に価格変更が3回以上行われる薬品は極めて少ないと想定されるからだ。少数の例外については、プロシージャのルーチンでトラップすればよい。このテーブルのもう1つのトリックは、正規化を壊したことで破壊されたルールを施行するためのCHECK 制約を付与したことだ。制約はテーブルに対する挿入、更新、削除を行う際にオーバーヘッドを発生させる。実際には、非正規化したテーブルの多くは、テーブルのすべての列の値が完全に確定するまではテーブルを変更することができない。さらに、一部のクエリのパフォーマンスは改善されるかもしれないが、他のクエリが悪影響を受けることもある。

繰り返すが、今日では、非正規化が許されるのはデータウェアハウスのみである。以前に比べれば結合のコストは非常に低くなっているのに対し、非正規化して例外をプロシージャのコードで処理するオーバーヘッドは、他のいかなるオーバーヘッドよりも大きい。

【訳注1】 当時米国は冷戦下にあり、中ソなど共産主義国との関係は冷え込んでいたが、米政治家ヘンリー・キッシンジャーらの努力により、米中関係は好転しつつあった。このような国交の回復は、当時の政治用語で「正常化された関係（normalized relations）」と呼ばれていたようだ。

【訳注2】 述語（predicate）とは、述語論理における用語で、真理値を戻り値とする関数のこと。たとえば、「人間」という述語を考えれば、人間(タケシ)＝真、人間(ポチ)＝偽となる。

【訳注3】 SQL-99において、配列型などの複合データ型が導入されて以来、SQLが扱うデータ型はスカラ型だけではなくなった。

【訳注4】 DB2のように、主キーのないテーブルを作ろうとするとDDLがエラーになる制限を設けているDBMSもあるが、これはあくまでDB2という実装独自の制限にとどまっている。

【訳注5】 理論的に更新可能でないビューとしては、たとえば基底表から`GROUP BY`を使って集約した結果を保持するビューがある。集約結果に対する更新を、どのように分解

して明細へ反映すればよいかは、一意には決定しない。ビューが更新不可能となる条件の一覧は第26章を参照。

【訳注6】 実際には現在では多くのDBMS が複合型をサポートしているので、作ろうと思えば1NF 違反のテーブルをSQLで作ることも可能だが、もちろんすすめられない。

【訳注7】 厳密に言えば、繰り返しグループは第1 正規形違反とは言い切れないところがある。テーブルのすべての列は、スカラ値から構成されているからだ。ただ、著者も言うように、配列を擬似的に構成しているという点で、いわば第1正規形の「理念」には反していると言えるだろう。

【訳注8】 たとえば、生徒「1」が「ER Diagrams」の受講を取りやめた場合、この講座そのものの情報が消えてしまう。また、「Database Design」の`course_code`が「CS-114」から「CS-115」に変わった場合、複数行を更新しなければならない。

【訳注9】 Terry Halpin, *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*, Morgan Kaufmann, 2001.

【訳注10】 「自然に反する罪」は法律用語で、歴史的には主に、同性愛、近親相姦などの性行為を指して使われてきた。ここでは、リレーショナル原理主義者にとって、正規化をないがしろにすることは「極めて反倫

理的な変態行為」に見えるということを、いささか皮肉めかした言い回しで表現している。

10

数値型

Numeric Data Types

SQLは計算向きの言語ではない。算術能力に関しては、他のあらゆる高級プログラミング言語に劣る。だが、SQLで数値を扱い、それらをホスト言語に受け渡す際に知っておくべきトリックがいくつかある。SQLの算術演算子や関数の多くは実装依存のため、読者の扱う製品で実際に試してみてデフォルトの設定、精度、ライブラリなどを調べる必要がある。

本章で扱うのは、同一行内のデータに対して水平方向に行う算術である。列方向に垂直に行う演算（集計など）は第30章以降で取り上げる。

10.1

数値型

標準SQLは非常に多くの数値型を持っている。その豊富さは、どのようなホスト言語の数値型であっても、SQLの側に対応する型を見つけられるほどだ。数値型にはまた、ベンダーが独自拡張しているものもある。一番有名なのは、Sybase/SQL Serverファミリーの通貨型（MONEY）である。実のところこれは、DECIMAL 型やNUMERIC 型と同じなのだが、入力や出力の際に$や¥などの通貨記号を付与することができる。

これはデータのフォーマッティングは行わないというSQL の原則を破る拡張だが、SQL Serverはそれだけでなく、計算においても問題を抱えている。たとえば、前ゼロやカンマといったフォーマッティング情報までデータベース内に保存できる。言うまでもないが、読者はこうした「機能」を使ってはならない。

SQLにおける数は、厳密な型と近似的な型に分類される。厳密な数値は、精度（P）とスケール（S）を持つ。精度とは、特定の基数においていくつの有効桁数を持つかを表現する正の整数である。標準SQLでは、基数は2または10に決められている【訳注1】。スケールは0以上の整数で、小数部の桁数を意味する。

今日では、10 を基数とするプラットフォームは多くない。ほとんどは2 進数のマシンだろう。だが、2 進数の中でも数値を表現する手段は多く存在する。2 の補数を使うもの、1 の補数を使うもの、ハイエンド向けとローエンド向けでも異なるし、扱える文字数もさまざまだ。SQLの数値に対する適切な心理モデルを一言で言うなら、「ビットとバイト」を気にするな、というものだ。これらは物理的表現

のレベルの概念であり、抽象度が低い。SQLはもっと抽象度の高い言語だ。

NUMERIC、DECIMAL、INTEGER、BIGINT、SMALLINTといったデータ型は、いずれも厳密な数値型で、整数のみを保持する。定義上、整数のスケールは0 なので、宣言の構文は単純にINTEGERとだけ記述し、スケールは記述しない。省略形のINTを使うこともできるが、本物のSQLプログラマであれば決して省略形は使ってはならない。こうした数値型はいずれも、負数も正数も保持することができる。

SMALLINTもまたスケールはゼロであるが、その保持できる値の範囲はINTEGER以下である。同様に、BIGINTもスケールはゼロだが、値の範囲はINTEGER以上である。それぞれの型が具体的にどの程度の範囲の値を保持できるかは実装に依存する。なおBIGINT はSQL-99 で追加されたデータ型なので、かつては実装している製品は少なかった。

もしかすると、読者の使っている製品はTINYINTという型を持っているかもしれない【訳注2】。これも厳密な数値型だが、その保持できる範囲はわずか0～255 である。かつ、これは標準SQL の型ではない。この型を持っていない製品では、次のように適当な整数型を使ってドメインを定義すれば、同じ範囲の型を作ることができる。

```
CREATE DOMAIN TinyInt
AS
SMALLINT CHECK (VALUE BETWEEN 0 AND 255);
```

DECIMAL(p,s)はDEC(p,s)と書くこともできるが、もちろん省略形は使ってはならない。DECIMAL(8,2)と宣言した場合、保持できる精度とスケールは、たとえば123456.78のようになる。有効桁数は8で、小数部は2である。

NUMERICとDECIMALの違いは微妙である。NUMERICは厳密な精度とスケールを指定する。一方、DECIMALはスケールは厳密に指定するが、精度は実装依存であり、指定された値と等しいか、それより大きいこともある。つまり、DECIMALには丸めの余地があるが、NUMERICにはないということである。メインフレームのCOBOL プログラマならば、NUMERICはPICTURE で定義する数値型、DECIMALはBCD（2進化10進数）だと思ってもいい。パーソナルコンピュータが普及してからのプログラマは、こういうデータ型には馴染みがないだろう。

次に、近似的な数値型について説明しよう。近似的な数値は仮数と指数から構成される。仮数は符号付きの数値であり、指数は符号付きの整数である。近似的な数値型は、精度を持つ。精度は正の整数で仮数部の有効桁を2 進数で表す。近似的な数値は、仮数部を10の指数乗倍することで表される。FLOAT(P)、REAL、DOUBLE PRECISIONといった型が近似的な数値型である。

FLOAT(P)とREALの間には微妙な違いがある。前者は与えられた値以上の2進精度を持つが、後者の精度は実装依存である。現実の世界では、REALとDOUBLE PRECISION は浮動小数点数を表すIEEE 標準754 である。FLOAT(P)はほとんど使われることはなく、いずれ標準において非推奨とされていくだろう。IEEEの算術関数はプロセッ

サのチップセットに組み込まれており、ソフトウェアで実装するよりもパフォーマンスが良い。IEEE 標準754 は、単精度に32 ビット、倍精度に64ビットを使っている。PCやほとんどのUNIX/LINUXプラットフォームもこれを採用している【訳注3】。

単精度の数値の範囲は、約±10^–44.85 ～10^38.53 であり、倍精度は約±10^–323.3 ～10^308.3 である。ただし、IEEE 標準にはSQLで定めていない特別な数がいくつかある。たとえば、0 はこの形式では直接的に表現することができない。そのため、0 は仮数が0 の特殊な数として定義されており、符号が別に定義されていることから、–0と＋0 の2 つが存在する。比較演算においては両者は等しい。

もし指数部がすべて0だが、仮数が1より小さかった場合、その値は「非正規化数」と呼ばれる（偶然にもテーブルに対して使うのと同じ用語だが、意味はまったく異なる）【訳注4】。この数は、(–s *0.f *2–126)で表現される。ここでsは符号、fは小数である。非正規化数は最小の正規化数よりも小さいが、ゼロよりは大きな値を意味する【訳注5】。倍精度では、非正規化数は(–s *0.f *2–1022)という形式を取る。ゼロは非正規化数の特殊タイプと見ることができる。

「＋無限大」と「–無限大」の2 つの数は、指数部がすべて1、小数部がすべてゼロである。符号ビットがあるため、正の無限大と負の無限大がある。無限大に特定の値を割り当てられることにはメリットがある。オーバーフローの場合でも演算を継続することができるのだ。無限大に対する演算はきちんとIEEEで定義されている。これはSQLにはない機能である。

NaN（Not a Number）という記号は、数値を表現しないビット設定を表現するために使われる。NaNは指数部がすべて1、小数部がすべて非ゼロというパターンで表現される。NaNには2つのカテゴリがある。QNaN（Quiet NaN）とSNaN（Signaling NaN）である。QNaNは算術の演算過程で伝播していく以外には、何ら例外を発生させない。これが発生するのは、ゼロ除算のように、演算結果が数学的に定義されていない場合である。

他方、SNaN は、発生したときに例外を発生させる。初期化されていない変数が早すぎるタイミングで使われたことなどを検知するのに便利だ。意味的には、QNaN は解を定められない演算を意味するのに対し、SNaNは不正な演算を意味するという違いがある。

SQLは算術に関してIEEEモデルを採用していないが、これにはいくつかの理由がある。標準SQLでは多くの場合、丸め、切り捨て、精度については実装に依存している。これはSQLが特定のハードウェアプラットフォームに制限されることを防ぐためである。もしIEEEの算術ルールをSQLにも適用しようとすると、無限に対する型変換と、変換後の有限の厳密な数値を表す方法が必要になる。「NULLを使えば…」という考えが頭をよぎるかもしれないが、すでに我々はNULLに大いに苦しめられている。これ以上NULLにまつわる問題を複雑にするのは得策ではない。

10.1.1　BIT、BYTE、BOOLEANデータ型

ANSI 標準はBOOLEAN、BINARY、BINARY VARYING のデータ型とそれに対する演算を定義している。だがすでに述べたように、ビットとかバイトというマシンレベルの概念は、SQLにおいては場違い

だし、事実、ほとんど使われない。SQLは3値論理を採用しており、当然のことながら、これはブール代数との相性が悪い。TRUE は FALSEより大きいが、NULL やUNKNOWN を含むあらゆる演算はUNKNOWNを返す。

SQLは高級言語である。抽象度が高く、物理的実装と無関係に定義されている。この基本的なデータモデルの原則は「データ抽象」と呼ばれている。ビットとバイトは物理的ハードウェアに依存する極めて低レベルの単位である。読者が使っているのは、ハイエンドマシンだろうか、ローエンドマシンだろうか。32 ビットだろうか、64 ビットだろうか。あるいは128 ビットということもあろう。2の補数を使うか、それとも1の補数を使うか。こうしたすべては、SQLにおいては心底どうでもよい。そもそも標準SQLは10 進マシンを許容しており、そこではビットが存在する必要はまったくない！

また、ビットにおけるNULLの扱いはどう考えればよいだろうか？ SQLのデータ型はすべて、値が存在しない状態、すなわちNULLを許している。では、ビットにおけるNULLは何を意味するか？ 定義上、ビットは2つの状態のうちの1つだ。オンかオフのどちらかだ。NULL の入り込む余地はない。もしDBMSベンダーがビットにNULLを追加したら、ビット演算はどう定義されるのだろう？ 論理と物理のモデルを混同すると、かくも混乱した状況を考えねばならないということが、わかってもらえるだろう【訳注6】。さらに、ホスト言語との連携でも問題が起きる。読者は、＋1、＋0、–0、–1といった数がすべてBOOLEAN型として使えるが、しかし整合的ではないことを知っていただろうか。C# では、ブール値はFALSE/TRUE を0/1 で表す。他

方、Visual Basic （.NET）はFALSE/TRUEを0/–1 で表す。周知のように、この2 つの言語は同じベンダーが提供しているが、それでもなお仕様が不統一なのだ。他の言語については推して知るべしであろう。

標準的なプログラミング言語であるCとCOBOLでは、ブール値は整数にマッピングされる。他方、Ada、Fortran、Pascal、PL/Iでは、ブール値そのものがサポートされる。SQLのすべてのデータ型は、NULLを許可しなければならない。そのため、標準SQLはNULLに相当するブール値にUNKNOWNを割り当てている。これがSQLのBOOLEAN型の動作をわけのわからないものにしている元凶だ【訳注7】。NULLのやっかいな特性の1つに、「伝播（propagation）」がある。これは演算の過程にNULL が含まれると結果もNULLになってしまう、という現象だ。たとえば次のような例を見てみよう。

(1 = 1) OR NULLの結果はNULL（UNKNOWN）
(1 = 1) OR UNKNOWNの結果はTRUE
(1 = 1) AND UNKNOWNの結果はUNKNOWN
(1 = 1) AND NULLの結果はNULL（UNKNOWN）

アセンブリ言語スタイルのビットフラグの扱い方にも問題はある。現実には、ビットを利用する状況はだいたい2通りに分類できる。それは、ビットが独立した属性の場合と、単独の属性を表現するベクトルとして使われる場合だ。前者の場合、エンコーディングは2 値に限られるし、ホスト言語や他のSQLに値が受け渡されること

はない。エンドユーザーにも簡単には理解できないし、拡張性もない。

後者のケースは、新米プログラマがよくやるのだが、yes/noのステータスコードを表すビットフラグを何個も用意して属性の状態を表す方法だ。これはパンチカード時代――百歩譲っても第2 世代から第3 世代――のプログラミングスタイルであり、はっきり言って時代遅れである。多くのフラグを使うと、結局は属性の状態を把握するのが難しくなる。子どもの頃、「20の質問」というゲームをやったことがある人もいるだろう【訳注8】。

病院において、患者に6 つの条件が該当するかどうかを考えてみよう。条件に該当するか否かを表すために、ビットを割り当てる。

条件1　患者は男性である
条件2　患者は妊娠している
条件3　患者は痩せている
条件4　患者は太っている
条件5　患者は熱を持っている
条件6　患者は高血圧である

考えうるビットの組み合わせは64 通りである。しかしそのうちで有意味な組み合わせは限られる（たとえば、1と2 についてともにyesのビットが立つ患者はいないし、3と4 が同時に成り立つ患者もいない）。データ整合性の観点から、このフラグ群に対しては次の2つの道が考えられる。

1. 問題を無視する。多くの新米プログラマが実際にとる道である。データ不整合のリスクは非常に大きい。
2. 膨大なCHECK制約をDDLに付与する。そのCHECK制約にはユーザー定義関数や実装依存のビットレベルのライブラリを使う必要があるだろう。その結果、パフォーマンスは悪いし、移植性もないコードができあがる。

さらに、患者に7番目の条件が追加されたらどうなるだろう。変更後にも有効なようにCHECK 制約を変更するのはかなり骨が折れるだろうし、まったく制約を付与しない場合には、データ不整合のリスクはさらに大きくなる。

ここはひとつ、落ち着いて考えてみなければならない。抽象度が高く、拡張性に富んで、移植可能な設計とエンコーディングの仕組みを作るにはどうすれいいのか。たとえば、患者の条件は階層的だろうか、それとも連結的だろうか。はたまたベクトルだろうか。未知の値、失われた値、適用不能の値はどのように扱うべきか。これらはそんなに簡単な設計ではない。ビットフラグを使って制御すればいいなどという安易な考えは捨てなければならない。

なお、BINARYおよびBINARY VARYINGは、SQLのデータ型では扱うことのできないさまざまなフォーマットのデータ、たとえば画像や動画、音声といったデータを保持するための標準的手段を提供するためのものである。本書ではこれらの詳細は割愛する。

10.2

数値型の型変換

数値型から別のデータ型へ変換するときに、少し驚くことがある。というのも、標準SQLはこの点についてあまり決めておらず、多くを実装任せにしているからだ。したがって、プログラマは実際に使っている環境で、基本的にどういう動作をするか、試してみる必要がある。

10.2.1　丸めと切り捨て

厳密な数値型の列に、厳密な数値または近似的な数値が割り当てられると、適合しない場合がある。こういうとき、データベースエンジンは、上位の有効数字を優先的に保存するよう、丸めや切り捨てを行い近似値を求める。切り捨てるか丸めるかの選択は実装依存である。このため、SQLの実装間、またはホスト言語とSQLとの間で値を受け渡すときに、予期せぬ動作をして驚くことがある。こうした「サプライズ」を避けるためには、要件上必要とされるよりも大きなサイズの小数部を列に持たせておくとよい【訳注9】。

さて、これまで断りなく「切り捨て」という言葉を使ってきたが、SQL での定義は0 へ近づくような切り捨てである。たとえば、1.5を切り捨てると1になり、21.5を切り捨てると21になる。これは他のプログラミング言語の仕様とは異なることがある。正数についてゼロに近づくよう切り捨てられる点は同じだが、他のプログラミング言語では、負数については0から遠ざかるような切り捨てが行われる可能性もある（たとえば、–21.5は–22になる）。

SQL は丸めについても標準では何も決めていない。プログラミングにおける丸めの方法は、代表的なものが2 つある。科学的方法と商

業的方法である。科学的方法は、いわゆる我々が学校で習う四捨五入である。これは一般に小さな数に向いている。商業的方法は、一風変わっている。端数が1～4 ならば切り捨て、6～9 ならば切り上げる。ここまでは科学的方法と変わらないが、違うのは端数が5の場合である。商業的方法では、切り捨てと切り上げのうち結果が偶数となるほうへ丸める【訳注10】。この方法にはほかにもいくつかのバージョンがあるが、共通しているのは、丸めによる誤差を可能な限り小さくしようとしている点である。この方法は銀行でよく使われるため「銀行家の丸め」とも呼ばれるが、その理由は、多くの数を合計したときの丸め誤差を少なくできるからである（データがランダムと仮定すれば、多くの数を足し合わせても、丸め誤差が特定の側に偏って累積することがない。逆にデータに偏りがある場合はこの限りでない）。

　通常、丸めの関数には科学的方法が採用されているが、商業的方法もMODを使えば実装できる。試しに、小数点2桁の適当な小数を持つテーブルを用意して確認してみよう。

```
CREATE TABLE Accounts (amount DECIMAL(5,2) NOT NULL);
```

Accounts：小数テーブル　　amount：小数

```
INSERT INTO Accounts
VALUES (12.15), (12.25), (12.35),
       (12.13), (12.27), (12.30),
       (12.45), (12.55), (12.65),
       (12.75), (12.85), (12.95);

SELECT amount, ROUND(amount, 1) AS scientific,
       CASE WHEN MOD((100 * amount), 10) <> 5
       THEN ROUND(amount, 1)
  ELSE CASE WHEN MOD (FLOOR(MOD((10 * amount), 10)), 2) = 0
                       THEN TRUNC(amount, 1) -- 偶数なら切り捨て
                       ELSE ROUND(amount, 1) -- 奇数なら切り上げ
                  END
       END AS commercial
  FROM Accounts;
```

結果

```
 amount   scientific   commercial
==================================
  12.15         12.2         12.2  ⬅偶数
  12.25         12.3         12.2  ⬅偶数
  12.35         12.4         12.4  ⬅偶数
  12.13         12.1         12.1
  12.27         12.3         12.3
  12.30         12.3         12.3
  12.45         12.5         12.4  ⬅偶数
  12.55         12.6         12.6  ⬅偶数
  12.65         12.7         12.6  ⬅偶数
  12.75         12.8         12.8  ⬅偶数
  12.85         12.9         12.8  ⬅偶数
  12.95         13.0         13.0  ⬅偶数
```

なお、商取引では、金額を小数第4 以上の精度で保持することが求められているが、表示するときは第2 位で丸める。これは、米国および欧州で適用されるGAAP（ギャープ：Generally Accepted Accounting Practice）という会計基準で定められている。

以下に、丸めについてのSQLの練習問題を掲載する。5000個のランダムな数（正数も負数も含む）を保持するテーブルを作る。この列は小数点第4 位までを保持する。そのデータを小数第2位までで丸

め、科学的方法と商業的方法の両方の結果で合計額を比較する。オリジナルの合計額とどの程度の違いが生じるか、確認してほしい。

```
CREATE TABLE RoundTest
(original DECIMAL(10,4) NOT NULL,
 rounded DECIMAL(10,2) NOT NULL);

-- テストデータの登録
INSERT INTO RoundTest (original,rounded) VALUES (2134.5678. 0.00),
etc.

UPDATE RoundTest
   SET rounded = original;

-- 科学的方法と商業的方法の両方で丸めを行うプログラムを書いて
-- 以下のクエリで、丸めた結果の合計を、オリジナルの数値の合計と比較しなさい

SELECT SUM(original), SUM(rounded)
  FROM RoundTest;
```

この両者の結果と、読者が使っているSQLの結果とを比較することによって、自分の製品がどのようなタイプの丸めを行っているかがわかるだろう。もしどちらの結果とも一致しなかったならば、そこには別の原因が働いている可能性がある。それはSUM関数の仕様に関わるものだ。実はSUM関数は、浮動小数点数の合計を求める際、足し合わせる行の順番によっては誤差が発生する可能性がある。この議論については、第29章で詳しく見ることにしよう。

科学技術計算用のソフトウェアには、浮動小数点数の集計時に生じる誤差を修正する特殊なライブラリを持っているのだが、大半のRDBMSは持っていない。浮動小数点数の算術は、なかなか商業的アプリケーションで行うことは珍しいので、プロセッサの側も対応していないことが多い。

10.2.2　CAST 関数

標準SQL はすべてのデータ型に適用できる一般的な型変換の関数として、CAST(<cast operand> AS <data type>)を定義している。<cast operand>は、列名、式、NULLのいずれでも代入できる。数値から数値への変換においては制限はほとんどなく、どんな型でも自由に変換できるが、やはり丸め誤差だけは注意しなければならない。また、比較述語は入力値の型が異なると暗黙の型変換を行ってしまうため、やはり注意が必要だ。実装によっては、独自のオプションを備えた型変換の関数を持っていることがある。こうした関数は、極めて実装依存であり、フォーマッティングの機能まで持つものもある。これはフォーマッティングはデータベース層では行わないという多層アーキテクチャの原則に反するものであるため、使ってはならない。

10.3

四則演算

かつてSQLの計算力は計算機にも劣っていた。今日では、標準SQLにも基本的な数学関数の多くが取り入れられるようになった。+、−、*、/といった2 項演算子は問題なく利用できる。乗算と除算が加算と減算に優先的に実行されるという順序も、普通の算術ルールと変わらない。

代数およびいくつかのプログラミング言語では、算術演算子の優先順位はより厳密に決められている。いわゆる「親愛なるサリーおばさん（My Dear Aunt Sally）」ルールである【訳注11】。このルールでは、強い順に、乗算＞除算＞加算＞減算となる。このルールにおいては、微妙な誤差を招くケースがある。たとえば、(大きな数＋大きな数−大きな数)の場合を考えよう。ここで、「大きな数」とは、特定の数値型で表現できる最大値だとする。もしこの演算を左から右に優先的にグループ化して計算すると、次のようになる。

```
((largenum + largenum) - largenum) = オーバーフローエラー!
```

これは加算を減算より優先させるという点で、「親愛なるサリーおばさん」ルールにも則っている。だが、ここでもし右から左へ優先的にグループ化すれば、次のようになる。

```
(largenum + (largenum - largenum)) = largenum
```

こちらは正しく計算ができ、結果も合っている。

このように、演算の順序にも複数のルールが存在し、それゆえ、ホスト言語のあるルールで得られた計算結果が、SQLでは異なる結果

になるということが発生する。SQLでも、計算順序を最適化するように式の計算順序を変えることはできなくはないが、実際には、多くの実装は単純に式を左から右へパースしている。計算結果を常に保全しようと思うなら、一番良い方法は、ホスト言語だろうとSQLだろうとおかまないなしに、しつこいぐらい括弧を使って明示的に順序を指定することだ。

数値の符号に関しては、＋および–の単項符号を使うことができ、数値や変数の前にいくらでも付けることができる。一点、SQLにおいて符号に関して注意が必要なことは、SQLがAda を真似した1行コメント「--」を使えることだ。このためパーサは「--」を、コメントの開始なのか、2つ重ねたマイナスなのか判別しなければならない。標準SQLではまた、C言語スタイルの/* */で囲う形式のコメントを許可している。これは何行にもわたってコメントを記述することができるし、国際的なデータ移行標準においても使われている（移行標準では逆に「--」のコメントは使えない）。

2 項演算子の入力が両方とも厳密な数値型であれば、結果も厳密な数値型になる。この動きに疑問はないだろう。他方、もし計算の中に1つでも近似的なデータ型が含まれていた場合は、ほかに厳密な数値型が含まれていたとしても、結果は近似的な数値型に変換される【訳注12】。問題は、その場合の精度とスケールがどのように決まるかである。

たとえば、S1とS2を最初の入力値のスケールとしよう。加算と減算の結果の精度は実装依存で決まるが、スケールはS1とS2の大きいほうが採用される。乗算の場合は、精度はやはり実装依存だが、ス

ケールは(S1＋S2)になる。除算の場合は、結果の精度とスケールはともに実装依存である。

ANSI のデータベース標準化委員会X3H2 INCITS（旧ANSI X3H2）は、かつて標準SQL における精度とスケールをどのような基準にするか議論したことがあったが、最終的には放棄した経緯がある。この結果、特に除算の場合は上の桁に向かって桁を減らしていき、最終的には結果が1桁の整数になってしまっても、計算上は合法とみなすことになった。

もちろん、現実にこんな馬鹿げたことを行う人はいない。実際のベンダーの対応はさまざまだが、代表的なものは、小数点をシステムパラメータとして調節できるようにする、特定の小数の桁数をデフォルトにする、実装上可能な限りの小数桁を表示させ、ユーザーにどの桁で丸めさせるかを決めさせる、のいずれかである。どのような方法を採用しているか、自分の使う実装の仕様を調べてほしい。

現在のANSI/ISO 標準が要求する基本的な算術関数を以下に示す。実際には多くのDBMS がこれ以外の算術関数を持っているが、その多くは集約などテーブルレベルで動作するものだ。

1. `MOD(<numeric dividend expression>, <numeric divisor expression>)`：剰余
2. `ABS(<numeric expression>)`：絶対値
3. `LN(<numeric expression>)`：自然対数
4. `EXP(<numeric expression>)`：指数

5. `POWER(<numeric expression>, <numeric power expression>)`：累乗
6. `SQRT(<numeric expression>)`：平方根
7. `FLOOR(<numeric expression>)`：入力以下の最大の整数
8. `{CEIL | CEILING}(<numeric expression>)`：入力以上の最小の整数

これらの関数の精度とスケールは実装依存だが、演算の仕様は通常の算術のそれを模倣している。その理由は、明白なことだが、データベースエンジン内部の算術パッケージは、皆共通のライブラリを使用しているからである。

10.4

算術とNULL

NULLは初心者にとって最も理解しがたいデータベースの概念であろう。本書はこの概念について極めて詳しく扱っているが、それはあとにとっておこう（第13 章参照）。本節では、算術の中でNULLがどのように振る舞うか、という点だけに限って解説する。

NULLはSQLで失われた値を表現する唯一の手段である。NULLの一般的な使用法は、現時点で不明の値を表現することである。「現時点で」というのはつまり、将来的には値が判明し、きちんとした値で置き換えられるということを含意している。

一口に失われた値といっても、その内容は多岐にわたる。ANSI X3（SPARC Study Group 1975）に対する中間報告（75-02-08）では、演算の結果あるいは属性の値として現れうる不完全なデータとして14 種類も挙げている。その中には、単純に失われたり不明な値というだけでなく、計算のオーバーフローやアンダーフロー、ゼロ除算、文字列の切り捨て、0 の0 乗といった計算上のエラーも含まれている。NULLというのは、こうした細かい区別をまったく気にせず、すべてを一括りに扱うために使われる。NULLはいかなるデータ型にも属さないが、すべてのデータ型の値を置換できる。これには、算術の定義を簡単にできるというメリットもある。

NOT NULL制約が付与されている列については、NULLは一切使うことができない。だが、標準SQLではCAST関数を使うことでNULLに対して特定のデータ型を宣言することができるようになっている。たとえば、CAST(NULL AS INTEGER)というようにだ。この変換が可能な理由の1つは、CASTの関数としての完全性を担保するためで

ある（つまり、あらゆる入力に対して同じ構文を適用できる、ということだ）。もう1つは、データベースエンジンに、その列がどのように作られたかを知らせるためである。

NULLに対する算術の基本的なルールは、「伝播」である。これは、NULLが演算の中に含まれると、問答無用で結果がNULLになるという原則だ。これは直観的にも理解できるだろう。NULL が失われた値を意味するという点に立ち返れば、それを計算した結果もわかるはずがない。しかし、(NULL / 0)の結果までNULL になるのは奇妙に思う人もいるだろう。どんな数であれ、0 で割ればエラーを返すべきだ。NULL が失われた値を意味するとすれば、あとできちんとした数値が判明するだろう。そのときにこの数式を実行すれば、エラーが返るのは当然だ――これはもっともな意見だが、SQLでは通らない。NULLには、あくまで伝播の原則が優先されるのである。

10.5

値からNULL、NULL から値への変換

ホスト言語はNULLをサポートしていないので、プログラマは、ホスト言語に渡すときは他の値で置き換えるか、INDICATOR変数を使って、ホスト言語にNULLに対する特別な動作をさせるかを選ばなければならない。

INDICATOR は整数のホストパラメータであり、他のホストパラメータの直後に続いて指定される。最初のホストパラメータがNULL を得ると、INDICATOR には負数がセットされる。また、SQLからホスト言語への受け渡し時に文字列の切り捨てが起こった場合は、INDICATORに正数がセットされる。INDICATORが0なら、変換に何の問題もなかったことを意味する。

10.5.1　NULLIF 関数

標準SQLは2つの関数NULLIFとCOALESCEを定義している。これはNULLを他の値に置き換えたり、その逆をやるときに使う。これはCASE式を使って書くことができる。NULLIF(V1, V2)は2つのパラメータを取るが、これは次のCASE式と同じだ。

```
NULLIF(V1, V2) := CASE
                  WHEN (V1 = V2)
                  THEN NULL
                  ELSE V1 END;
```

つまり、最初のパラメータが2 番目のパラメータと等しければNULLを返す。そうでない場合は、最初のパラメータを返す。変な関

数だと思うかもしれないが、実はこの関数には広い応用方法がある。NULLIF関数の重要な特性は次のようなものがある。

特性1

NULLIF(x, x)はx の値が何であれNULLを返す。これはx が NULL の場合もそうだ。というのも、(NULL = NULL)は UNKNOWN でありTRUE ではないため、この場合はV1（つまり NULL）が返されるからだ。

特性2

NULLIF(0, (x-x))はすべての非NULLのxをNULLに変換する。xがNULLの場合は（V1 である）0 が返される。なぜなら、(NULL-NULL)はNULL であり、(V1 = V2)の比較はUNKNOWN となるからである。

特性3

NULLIF(1, (x-x + 1))はすべての非NULLのxをNULLに変換する。xがNULLの場合は、（V1である）1 が返される。なぜなら、(NULL-NULL)はNULL であり、(V1 = V2)の比較は UNKNOWNとなるからである。

これらの特性は、すべての数値型と数値について当てはまる。

10.5.2 COALESCE 関数

COALESCE(<value expression>, ..., <value expression>)関数は、<value expressi on>のリストを左から右へスキャンして、最初のNULL でない値を返す。もしリストのすべての<value expression>がNULLだったら、結果はNULLになる。

これは特にSELECT 句で最も頻繁に使われる、NULLから値への変換関数である。たとえば、社員への給与の合計額を帳票出力する場合、次のようなクエリを書くだろう。

```
SELECT emp_nbr, emp_name, (salary_amt + commission_amt) AS pay_tot
  FROM Personnel;
```

だが、社員の中には、commission_amt（歩合給）しか受け取らない者と、salary_amt（基本給）とcommission_amt の両方を受け取る者がいる。また、スタッフ部門はsalary_amtしか受け取らない。とすると、salary_amt列やcommission_amt列がNULLの社員がいてもおかしくない。つまり、列の加算を行うとNULLの伝播が発生し、結果がNULLになる可能性がある。

このようなNULLに起因する問題を防ぐ改良版はこうだ。NULLを0.00に変換することでNULL の伝播を防いでいる。

```
SELECT emp_nbr, emp_name
       (COALESCE(salary_amt, 0.00) + COALESCE(commission_amt, 0.00)) AS paycheck_amt
  FROM Personnel;
```

COALESCE関数の別の応用例を見てみよう。ある年の月別の購入額を列として持つテーブルを作る。

```
CREATE TABLE Payments
(cust_nbr INTEGER NOT NULL,
 jan DECIMAL(8,2),
 feb DECIMAL(8,2),
 mar DECIMAL(8,2),
 apr DECIMAL(8,2),
 may DECIMAL(8,2),
 jun DECIMAL(8,2),
 jul DECIMAL(8,2),
 aug DECIMAL(8,2),
 sep DECIMAL(8,2),
 oct DECIMAL(8,2),
 nov DECIMAL(8,2),
 "dec" DECIMAL(8,2), -- DEC は予約語のためダブルクォートで囲む
     PRIMARY KEY (cust_nbr));
```

Payments：支払いテーブル　　cust_nbr：顧客番号　　jan～dec：1月～12月

問題は、顧客が最後に購入した月の支払額を得るクエリである。支払いが発生していない月の列はNULL が入っているとすれば、クエリのイメージはCOALESCE関数を使って次のようになる。

```
SELECT cust_nbr,
       COALESCE ("dec", nov, oct, sep, aug, jul, jun, may, apr, mar, feb, jan)
  FROM Payments;
```

もちろん、このイメージはまだ不完全だ。これだと結局最後の支払いが発生したのが何月なのかがわからない。これを知るには、少し読みにくいコードになるが、NULL でない列の金額を月の名前に変換する必要がある。ある月の列名を「mon」だとすると、この変換ロジックは次のようになる（<Month>は特定の月を表す整数値である）。

```
NULLIF (COALESCE(NULLIF (0, mon-mon), <Month>), 0)
```

この式の変換ロジックを段階的に追うと次のようになる。

パターン1　monが数値だった場合

```
NULLIF(COALESCE(NULLIF(0, mon - mon), <Month>), 0)
NULLIF(COALESCE(NULLIF(0, 0), <Month>), 0)
NULLIF(COALESCE(NULL, <Month>), 0)
NULLIF(<Month>, 0)
(<Month>)
```

パターン2　monがNULLだった場合

```
NULLIF(COALESCE(NULLIF(0, mon-mon), <Month>), 0)
NULLIF(COALESCE(NULLIF(0, NULL-NULL), <Month>), 0)
NULLIF(COALESCE(NULLIF(0, NULL), <Month>), 0)
NULLIF(COALESCE(0, <Month>), 0)
NULLIF(0, 0)
(NULL)
```

あとは、この関数をリスト化すれば、支払いが生じた直近の月の名前が求められる。このように関数をネストすることでSQLではいろいろなことができる。LISPプログラマはこのような考え方に慣れているが、多くの手続き型言語のプログラマはそうではない。だが、少し練習すれば使いこなせるようになる。

10.6

数学関数

SQL:2003は、元来標準SQLで定義されていた四則演算を拡張して、いくつかの算術ライブラリを含めるようにした。もっとも、その大半はすでに何十年も前からベンダー製品ならサポートしているものだった。SQLは計算向きの言語ではない。そのため、SQLは算術関数や文字列関数のライブラリを持つべきではないし、地理情報を扱う手段はSQLという言語の一部ではなく、外部拡張に任せるのがスジだ。

10.6.1　数論演算子

MOD(n, m)はnをmで割ったときの剰余を返す。もしnかmのどちらかがNULLならば、結果はNULLになる。m が0 の場合は、ゼロ除算の例外が発生する。それ以外の場合は、以下の条件を満たす整数rを返す。

条件1　rはnと同じ符号を持つ
条件2　rの絶対値はmの絶対値よりも小さい
条件3　n=m*k+r（kは適当な整数）

nとmが基数（つまり正の0でない整数）でない場合の動作はトリッキーなことがあるので注意が必要だ。負数や小数に対する動作については、DBMS のパッケージ仕様を確認してほしい。これは手続き型言語においても、言語によって微妙に仕様が異なるところだ。もし他のプログラミング言語をDDLに埋め込むような「機能」を使って

しまうと、データの整合性が損なわれてしまうためやってはならない。これは特に、Pascalなどの言語において大きな問題となる。

1996 年9 月、レン・ギャラガーは、SQLにおけるMOD関数の仕様を見直すことを求めた。従来のワーキングドラフトでは、MOD(n, m) はnとm がともに正の場合のみ定義されており、どちらかが負の場合の動作は実装依存になっていた。n が負だと、特に数学的に要求される意味はないので、そもそも多くの実装はまったく定義していないか、プラットフォームの制約の中で最も簡単に計算できる結果を返していた。

だが、m が負の場合はきちんとした数学的解釈があるため、やはりSQL のMOD 関数においてもそれを維持することが求められた。そこで次のような定義を提案したい。

定義1

もしnが正であれば、結果rは一意な負でない整数になる。ただしrはmより小さく、n = (m * k) + r（kは適当な整数）。

定義2

nが負数の場合は、結果は実装依存の整数r。ただし、rはm以下かつ(_m)以上で、n=(m * k) + r（kは適当な整数）。この場合は、実装依存の結果を返すという警告も発生させる。

この定義は、適当な正数nに対して、MOD関数が、すべての整数群からの加法のもとで、整数{0, 1..., m-1}の剰余群への準同型写像

（homomorphism）になることを保証する。この写像は、次のような群の性質を保存する。

1. 加法的単位元が保存される。つまり、MOD(0, m)= 0
2. MOD(-MOD(n, m), m)= m-MOD(n, m)によって定義される剰余群において加法的逆元が保存される。つまり、MOD(-n, m)=-MOD(n, m)
3. 「⊕」をMOD((MOD(m, m)+ MOD(n, m)), m)によって定義される剰余加法とした場合、加法の性質が保存される。つまり、MOD((m + n), m)= MOD(m, m)⊕ MOD(n, m)
4. MOD((MOD(m, m)⊖ MOD(n, m)), m)で定義される剰余減法に対して、減法の性質が保存される。つまり、MOD(m-n, m)= MOD(m, m)⊖ MOD(n, m)この定義を採用した場合、次のような結果が得られる。

```
MOD(12, 5) = 2
MOD(-12, 5) = 3
```

MOD(-12, 5)の結果に驚くかもしれない。これは、m=5なので、k=(-3)になり、-12=5 * (-3)+3ということだ。アプリケーションによっては、MOD(-12, 5)の結果が3よりは-2や-3のほうが望ましい場合もあるかもしれない。だからこそ、MOD関数の定義は実装ごとに大きく異なっているのだろう。だが前述の数学的性質を使えることのメリットは、他のいかなる要件よりも重要だ。

SQL の結果がどうなるかを知っておくことは、特定のアプリケーションで望ましい結果を得るために式を変更するのに役立つ。表

10.1に、実装別のMOD関数の動作パターンを示す。

表10.1　実装別の MOD(n,m)の動作パターン

パターン	n	m	Oracle10g	SQL Server 2008	DB2 9.7	PostgreSQL 9.1	MySQL 5.6
a	12	5	2	2	2	2	2
b	-12	5	-2	-2	-2	-2	-2
c	-12	-5	-2	-2	-2	-2	-2
d	-12	-5	-2	-2	-2	-2	-2
e	NULL	5	NULL	NULL	NULL	NULL	NULL
f	NULL	NULL	NULL	NULL	NULL	エラー	NULL
g	12	NULL	NULL	NULL	NULL	NULL	NULL
h	12	0	12	エラー	エラー	エラー	NULL
i	-12	0	-12	エラー	エラー	エラー	NULL
j	0	5	0	0	0	0	0
k	0	-5	0	0	0	0	0

MOD関数以外の数論関数も簡単に見ておこう。他は特に難しいことはない。

- **ABS(n)**：nの絶対値を返す。nがNULLの場合は、NULLを返す。
- **SIGN(n)**：n が負の場合は-1を、0 の場合は0を、正の場合は1を返す。n がNULLの場合は、NULLを返す。数学の用語で言う「符号関数（signum function）」である

10.6.2　指数関数

- **POWER(x, n)**：数xをn乗する。もしいずれかのパラメータがNULLの場合はNULLを返す。xが0かつnが負の場合は、例外が発生する。xが0かつnが0の場合は1を返す。xが0かつnが正の場合は0を返す。
- **SQRT(x)**：xの平方根を返す。これはPOWER(x, 0.5)の省略形である。
- **LN(x)**：xの自然対数を返す。もしxが0か負の場合は、例外が発生する。
- **EXP(x)**：x 乗に対する自然対数の底*e*（2.718281828459045235 36…）を返す。もしx がNULLの場合は、結果はNULLになる。また、もし結果を宣言されたデータ型で表現できなかった場合は、例外が発生する。

10.6.3　小数を扱う関数

- **FLOOR(x)**：x以下の最大の整数を返す。xがNULLの場合はNULLを返す。

- **CEILING(x)**：xx以上の最小の整数を返す。xxがNULLの場合はNULLを返す。

この2 つは標準SQL には含まれていないが、たいていの実装はサポートしている。また、仮にサポートされていなくても、乗算と除算、および以下に示す丸めと切り捨ての関数を使えば簡単に書くことができる。

- **ROUND(x, p)**：数xx を小数桁p で丸める。もしいずれかのパラメータがNULL の場合はNULLを返す。
- **TRUNCATE(x, p)**：数xxを小数桁pで切り捨てる。もしいずれかのパラメータがNULLの場合はNULLを返す。

次の2 つの関数はMySQL、Oracle、およびSQL:2003 でサポートされているが、サポートしていない実装においてもCASE式を使えば記述できる。

- **LEAST(<expression list>)**：入力のリストのうち、最小値を返す。数値、時間、文字列に対して適用できる。
- **GREATEST(<expression list>)**：入力のリストのうち、最大値を返す。数値、時間、文字列に対して適用できる。

この2つの関数は第29章で詳しく取り上げる。

10.6.4　数値から文字への変換

帳票出力を行うアプリケーションにおいて、数値を文字表記に変換する関数は標準的な機能である。数値を文字列に変換する理由は

さまざまだが、たとえば見た目上のチェックや、法的あるいは公的な文書に必要なフォーマットに整形したりするのが主な理由だ。こういう変換は、一般にSQLで行うことではないし、標準SQLでもこうした機能は定義していない。

だがそれでもあえて、標準SQLだけを使って数値から文字列への変換を行おうとする人がいる。その1 人がスツ・ブルームである。これは彼が2002 年2 月にSQL Server ニュースグループに投稿した質問だ。まず、テーブルを作ろう。

```
CREATE TABLE NbrWords
(number INTEGER PRIMARY KEY,
 word VARCHAR(30) NOT NULL);
```

NbrWords：数値と文字列テーブル　　number：数値　　word：文字列

このNbrWordsテーブルに0から999までの文字列を投入する。

```
INSERT INTO NbrWords VALUES(1, 'one');
INSERT INTO NbrWords VALUES(2, 'two');
INSERT INTO NbrWords VALUES(3, 'three');
...
INSERT INTO NbrWords VALUES(999, 'nine hundred ninety nine');
```

さて、いま考慮すべき数値の範囲を1～999,999,999とすると、数値から文字列への変換は次のクエリで可能になる。

```
SELECT
CASE WHEN :num < 1000
     THEN (SELECT word FROM NbrWords
             WHERE number = :num)
     WHEN :num < 1000000
     THEN (SELECT word FROM NbrWords
             WHERE number = :num / 1000)
          || ' thousand '
          || (SELECT word FROM NbrWords
               WHERE number = MOD (:num, 1000))
     WHEN :num < 1000000000
     THEN (SELECT word FROM NbrWords
             WHERE number = :num / 1000000)
          || ' million '
          || (SELECT word FROM NbrWords
               WHERE number = MOD((:num / 1000), 1000))
          || CASE WHEN MOD((:num / 1000), 1000) > 0
                  THEN ' thousand '
                  ELSE '' END
          || (SELECT word FROM NbrWords
               WHERE number = MOD(:num, 1000))
      END;
```

もっと大きな数と小数を扱えるよう拡張するにはどうすればよいかは、自明だろう。なお、2500 を「100 が25 個ある（= twenty-five hundred）」と考えるか「1000 が2 個と100 が5 個ある（＝two thousand five hundred）」と考えるかは、趣味の問題であって、計算上はどうでもいい。このクエリをもっと簡潔に書くことも可能だが、コードとテーブルのサイズを考えると、これが最もバランスの取れた解であろう。

10.7

IPアドレス

現在、IPアドレスはIPv4 からIPv6 への過渡期にある。IPv4 は、0から255までの4 つの整数から構成されており、ドット（.）で区切られる。IPv4 の問題点は、遠からずこれが枯渇すると見られていることである。IPv6 は、4 桁の16 進数8 つから構成されており、コロン（:）で区切られている。IPv6 は技術的には数値型だが、IPアドレスはバイナリとして保存され、文字列として表示される。IPv6の主な仕様が確定したのは、1998年12月のことで、IETFのタスクフォースにおいてだった（RFC 2460）。余談だが、IPv5というのはない。いや正確にはあるのだが、動画ストリーミングの実験的なプロトコルで、一般向けに利用されるものではなかった。IPv4は32 ビットを使っていたが、IPv6は128ビットのアドレス空間を使っている。ほかにもいろいろ細かい話はあるのだが、とりあえず本章に関係するのは、その表現形式についてである。

10.7.1　CHAR(39)形式での格納

IPv6のアドレスを16進数の形式で格納するには、`CHAR(39)`の列があればよい（内訳は、4 桁の文字が8 つと、7 つのコロン）。フォーマットをチェックするには、DDL に正規表現を使った`CHECK`制約を付与すれば簡単だ。IPアドレスの良いところは、バイナリデータと違って、人間が読解できる点だ。だが、可読性のために各グループの前ゼロを省略するのは推奨できない。たしかに可読性はよくなるが、アドレス同士を機械的に比較するのが面倒になる。

10.7.2　バイナリ形式での格納

現在主流のハードウェアの多くは64ビットの整数をサポートしているが、まだ128ビットはサポートしていない。そのため、バイナリのIPv6を格納できる整数型はない。ただラッキーなことに、IPv6はインターフェイス識別子として64ビットを使っている。これはリンクレイヤーからの媒体アドレス情報（MACアドレス）からインターフェイス識別子を自動生成できるようにするためである。したがって、インターフェイス識別子とそれ以外の部分で分ければ、2つの`BIGINT`（64ビットの精度を持つ）を使えば足りる。

10.7.3　独立したSMALLINT での格納

IP アドレスは、それぞれが意味を持った部分から構成されている。それゆえ、IP アドレスを分解して個別の列として保持するようにモデル化することも可能だ。もちろん、IPアドレスはそれ全体でも原子的なデータ要素であるが、スカラ値の集合としてモデリングすることもできるということだ。こうした用途に向いているデータ型がDBMSにあれば便利だが、なくてもドメインで次のように作ればよい。

```
CREATE DOMAIN SmallInt
AS INTEGER DEFAULT 0 CHECK (VALUE BETWEEN 0 AND 65535);
```

次にこのデータ型を（非スカラ的ではあるが）原子的なデータ要素を宣言するのに使用する。

```
ip1 SMALLINT NOT NULL,
ip2 SMALLINT NOT NULL,
...
ip8 SMALLINT NOT NULL
```

この使い方をするときの気の利いたトリックは、インデックスを作成するときに、`ip8`を先頭列にして逆順に作成することだ。というのは、IPアドレスでは最後尾のグループが最もカーディナリティが高く、絞り込みが効くからだ。

【訳注1】 基数（radix）は、記数法においていくつの数字を使って数を表現するかを示す数。基数が2であれば、0と1 の2個、10であれば、0～9までの10 個の数字を使ってあらゆる数を表現する。「n 進法」というときのnは基数を表す。基数は「底（base）」とも呼ばれる。

【訳注2】 `TINYINT`はSQL Server、MySQLなどがサポートしている。MySQLでは`TINYINT`は–128～127の範囲である。

【訳注3】 浮動小数点のフォーマットは複数あるが、近年最も広く採用されているのがIEEE 754である。符号s（0または1）、仮数c、指数qの3つの整数で数値を表現する。その形式は、(–1) s×c×bqという形を取る。bは基数で2または10である。

【訳注4】 指数表記において、正規化された形式とは、仮数部が1以上10 未満になるよう表記することである。たとえば、34.56×10^11は、仮数部「34.56」が10より大きいため、正規化されていない。これを3.456×10^12とするのが、正規化である。非正規化数とは、仮数部が1未満になってしまうケースである。

【訳注5】 非正規化数が定義された背景には、アンダーフローの防止という理由がある。極めて微小な差異しかない2つの

数値a、bがあり、その差異は浮動小数点の正規形では表現できないほど小さいとき、a–bの結果は0になってしまう。これをアンダーフローと呼び、予期せぬところでゼロ除算が生じるなどの危険がある。これを防止するために、「0ではないが、最小の正規数よりも小さい数」として非正規化数が導入された。a–bの結果を非正規化数とすることで、結果が0になることを防止できる。

【訳注6】 とはいえ、bit型をサポートしているDBMSは多く、SQL Server、PostgreSQL、MySQLなどでこのデータ型を使用することができる。もちろん、著者が言うように、このようなデータ型は極力使うべきではないし、やむなく使う場合もNOT NULL制約を付加して、2値のみの演算にとどめるべきだろう。

【訳注7】 本来、NULLは値ではなく、「値が存在しない状態」を指し示すためのマークにすぎない。それゆえ、3 値論理における真理値の1つであるUNKNOWNとNULLは、同値ではありえない。しかし、標準SQLではUNKNOWNをNULLで代用するという混乱を招く仕様になっている。そして、この標準SQLのブール値をサポートしているPostgreSQLなどでも、NULLが真理値の1つとして扱われるという混同が起きている。

【訳注8】 「20の質問」は2人以上でやる、もの当てゲームの一種。出題者は心の中に1単語で表現できるもの（生物で

もよい）を思い浮かべる。ただし、回答者には秘密にしておく。回答者は「はい／いいえ」で答えられる質問を出題者にして、出題者は正直に答える。20問以内で出題者の考えたものを言い当てられれば、回答者の勝利。最近では人工知能とデータベースを利用したゲームアプリも存在する。

【訳注9】　実際、宣言した小数の桁数を超えるデータが列に投入された場合、エラーを返すのではなく、無言で保持可能な桁数に切り捨てたり丸めてテーブルに入れる動作がデフォルトというDBMSは多い。これに気づかないで開発を続けていると、どこかのタイミングでSQLの結果がおかしいことに面食らうことになる。

【訳注10】　商業的方法は「偶数丸め」とも呼ばれる。丸めの対象桁の数値が5 の場合、切り上げられたり切り下げされたりして、結果が常に偶数になる。この方式を採用しているソフトウェアとして、Microsoft Accessの ROUND 関数がある。以下のMicrosoft社のドキュメントも参照。

- [OFFXP] VBAのRound関数について
  http://support.microsoft.com/default.aspx?scid=kb;ja;418216

【訳注11】　「My Dear Aunt Sally」の頭文字はそれぞれ、Multiplication（乗算）、Division（除算）、Addition

（加算）、Substraction（減算）の頭文字と一致する。つまり語呂合わせの記憶法である。

【訳注12】　たとえば、INTEGER 型（厳密な数値型）の数値aとREAL 型（近似的な数値型）の数値bがあった場合、a+bの結果の型はREAL型になる。

11

時間型

Temporal Data Types

SF 作家クリフォード・シマックに、『時は最も単純（Time is the Simplest Thing）』（1961）という作品がある。彼は間違っていた。といっても別に、2000 年問題の話ではない。時にまつわる問題はそれよりも前に始まっていたのだ。カレンダーは規則性を持たないので、ISO標準で決められている時間の唯一の単位は、秒だけである【訳注1】。年、月、週、時間、分は、いずれも国際単位系には含まれておらず、ISO 標準では規約として言及されているだけだ。SQLに時間型が追加されたのは、SQL-92においてである。もっとも、当時もほとんどの実装が何らかの形で時間を扱うデータ型は持っていた。問題は、どのベンダーも内部で独自のトレードオフを行っていたことである。具体的なSQLコードはあとで見ることにして、まずは時間についての基礎から話を始めよう。

11.1

カレンダー標準についての注意事項

AUC 708 年以前のローマとエジプトの太陽暦には閏年は存在しなかった（AUC はラテン語「ab urbe condita（ローマ建国以来）」の略である【訳注2】）。困ったことに、太陽暦の1年に含まれる日数はキリがよくない。1年で約365.2422日であるため、時とともに端数が積み重なって誤差となる。そのため、太陽暦はおよそ4 年で1日ずれる。たとえば、エジプトの暦では約1,461 年で丸1年ずれた【訳注3】。その結果、太陽暦は農業にまったく役に立たず、やむなくエジプト人はナイル川の氾濫を予測するために星の動きを使った。すでに紀元前1世紀頃、アレクサンドリアのソシゲネスは太陽暦が導入されてから、2回り以上ずれていることに気づいていた。

ユリウス・カエサルが暦の刷新を命じたのが、AUC 708 年（紀元前46 年）である。この年は調整のため1年が445日あった。このときカエサルはソシゲネスのアドバイスを受けて、閏年を導入した。多くのローマ人は、AUC 708年を「混乱の年」と呼んだが、このとき採用されたユリウス暦は、その後の世界標準となった。

ユリウス暦は4 年につき1日の閏年を持っており、このおかげで短いスパンではかなり正確な暦になった。それでもまだ、400 年で3日程度の誤差は残っていたが、これは1日につき0.0022日の誤差が合算された結果である。

その結果、ユリウス暦は1582 年までに10 日ほどのずれが生じた（閏年のない暦であれば、AUC 708～2335 年（1682 年）までに1回りほどはずれていただろうから、それでも当時としてはユリウス暦は正確なほうであった）。農作物を作るうえで、正確に知ることが

重要な夏至が、すでに7 月21日ではなくなっていた。そこで学者たちは教皇グレゴリウスを説得して、1582 年10月からおよそ2週間分の日付を省略して調節するようにした。

CE 800 年と1200 年はキリスト教世界では閏年であった。1600 年が閏年かどうかは地域による。ヨーロッパ諸国は同時に新暦に移行したわけではないので地域差があるのだ【原注1】。

暦の修正は、経済的社会的な混乱を招いたこともある。大英帝国とその植民地では、1752年9月2日の次が同年の9月14日になった。前年に施行された暦の修正法案は、「年の開始を規制するため、また現在使用されている暦を是正するための法律」という【訳注4】。この法案には、地代、リース、債券などから得られる収益の額を調整するための条項が含まれていた。これにより、地代などの支払額は暦の修正による影響を直に受けることになった。暦上の日数に正確に対応して日割り計算されることとされたため、1752 年の9 月に関しては、通常のひと月分の利子を払う必要はないし、年利についても同様とされたのである。

この法案が施行されたことで、市民の間では広範囲にわたる暴動が起きたという。その理由は主に宗教的なもので、人間の寿命は神により厳密に定められており、いつ生まれていつ死ぬかは決まっている。それを人間の都合で変えることは許されない、9月から11日を奪うことは、地上に生きるすべての人間の寿命を11日奪うものだ、というものだった。同じことが、新年の始まりを3 月25日から1 月1日に変える際の83日間についても起きた。この暴動を気が狂ってい

ると思う人は、今日サマータイムのためにあわてて時計を1 時間調節する人が何人いるか考えてみてほしい。

　やっかいなことに、新年の始まりは国によって違っていた。大英帝国は3 月25日から始まっているのに対して、別の国では12月25日や3月1日、1月1日という調子だった。大英帝国と植民地では、1750年は3 月25日に始まり、3 月25日に終わった。つまり、1750 年の3月24日の次は、同年の3 月25 日だったのである。この年は閏年だった。この年にはもう1 つ閏日が2 月にあったが、これは閏日に関する暦の構造は変わっていなかったからだ。

　ラテン語ではseptem は7 番目を意味する。これが9 月（September）の語源だ。同様に、octem は8 番目、novem は9 番目、decem は10 番目を意味する。したがって本来、September は7月、October は8 月、November は9 月、December は10 月を意味するはずである。なぜSeptemberが9月になったのか？ 実は、Septemberは1752年までは7月だったのだ――そう、この名前は1年の開始が1月1日ではなく3月25日だった頃の名残りなのだ。

　かなり最近まで、日付をどう表記するのが適切かという点についての一般的な合意は存在しなかった。国ごとに独自の商取引の習慣があり、アメリカでは、日の前に月を書くが、イギリスでは逆である。他の国の習慣もバラバラだ。こうした日付フォーマットの相違は、国際的な商取引の場では混乱のもとになる。ボストンでは「12/16/95」と書くのが、ロンドンでは「16/12/95」、ベルリンでは「16.12.95」、ストックホルムでは「95-12-16」となる。しかも、

1 つの国の中でも業界が違うとフォーマットが異なることが、混乱に拍車をかけている。

　このような状況に対してソフトウェアベンダーの採用した対策は、あらゆる可能性を考慮することだった。普通の表示フォーマットは、年を2 桁または4 桁の数字、月を2 文字または3 文字の文字列、日を2桁の数字で表す。区切り文字には'/'、'–'、空白が使われる。

　かつて、NATOは月を表すのにローマ数字を使おうと試みたことがあった。それによって委員会メンバー間の言語の問題を回避しようとしたのだ。米国軍もこのフォーマットについて研究を行い、英語話者においては、年は4 桁、月は3文字、日は2桁で表すのが最も誤読誤記が少なくソートするときも便利だということを発見した。これが、現在軍事用途の「24時間表記」の表記法の基礎になっている。

　今日では、日付データのフォーマットをISO 8601標準に変換するプログラムは簡単に入手できる（“Data Elements and Interchange Formats -Information Interchange – Representation of Dates and Times”）。このISO 標準は、企業の商取引標準であり、電子データ交換プロトコル（EDIFACT）の標準でもある。いわゆるyyyy-mm-ddフォーマットで、当然標準SQLの一部でもある。他の標準的なプログラミング言語も、今後日付型を追加するときはこれに準拠することになるだろう。

　フルバージョンのISO 8601 のタイムスタンプは、ローカルタイムとUTC（Universal Coordinated Time：協定世界時）のどちらも採用

できる。UTCはかつてのGMT（グリニッジ標準時）の後継だ。

1970年、UTCがITUの専門家からなる委員会によって採択された。ITUは「UTC」がすべての言語において混乱を最小限にする唯一の省略形だと考えていた。UTCの代わりに提案されたものとして、英語表記のCUT（Coordinated Universal Time）と仏語表記のTUC（Temps Universel Coordinné）の2つがあった。UTCはこの英語とフランス語の折衷案として採用されたものだ。その理由は、最後の文字がCなら、すでにあったUT0、UT1などとも整合が取れることと、数学的な表記法のほうが国際的な用途を考えると優れていると判断されたためだ。

UTCは天文時と完全に同じではない。地球の自転は一定速度ではないため、かつてUTCは閏秒を使って太陽暦と調整しなければならなかった。現在、UTCは原子時計によって定義されるようになり、これは非常に正確なので閏秒による調整を必要としない。

米国特有の問題として、4 つのタイムゾーンと「法定時間（lawful time）」がある。法定時間は商法で使われる法律用語で、通常はサマータイムを使うかどうかと、サマータイムを地域ごとにどう定義するか、という規約を指す。

データベースにおけるUTCと表示および入力における法定時間の必要性は、一般的にまだそれほど認識されていない。だが、EDIと分散データベースは、タイムスタンプの比較にUTC を使わざるを得ないだろう。タイムゾーンなしの日付は、分散システム環境では解釈があいまいになる危険がある。1995-12-17にロンドンで開始されたトランザクションは、1995-12-16にボストンで開始されたトランザク

ションよりも開始時刻が早いかもしれない。こうした問題は、2 つの日付にタイムゾーンを付けて内部的にUTCに変換して解決する必要がある【訳注5】。

11.2

SQL の時間型

SQL-86とSQL-89 には時間を扱う型が存在しなかった。日付および期間を表すデータ型が追加されたのはSQL-92だ。標準SQLは、経時的に変化するデータを時間型によってサポートしているが、SQLは時間依存的なテーブルを持っていない。SQLはまた、順序づけられた制約、クエリ、ビュー更新を持っていないし、適用時間（現実の業務動作のモデリング）とトランザクション時間（格納されたデータの展開を把握すること）の明確な区別もしていない【訳注6】。SQLがサポートしているのは、いわゆる非順序的な演算だけである。これはつまり、SQLでは物事は特定の実行順序に従うのではなく、「すべて同時に」起きるということだ。すべての制約は、すべてのタイミングにおいて真でなければならないし、すべての更新、挿入、削除は同時にテーブル全体を操作するのであって、一度に1行ずつ操作するのではない。たとえば、もし1つのタイムスタンプ列を持つ行を1 つの文で複数行挿入する場合、すべての行が同じタイムスタンプを持つ。行がテーブルにロードされる順序は何の意味も持たないのだ。

標準SQLは、時間を扱うデータ型はほぼ完璧に備えている。数値や文字列からこれらのデータ型に変換するためにはルールがあり、時間のデータ型を同期させるグローバルタイムゾーンのためのスキーマテーブルもある。もっとも、標準SQL の取り決めはあまりに完璧で精巧すぎるため、小規模のSQL製品はこうした機能を実装していないこともある。標準SQLは、国際標準として全世界で共通して使えるやり方で時間を扱わなければならないが、実際、我々の多くはロー

カル時間の中だけで仕事している。もし読者に国際電話をかけた経験があれば、この問題の重要性がちょっとわかるだろう。

時間に関する共通の規約と用語もまた混乱している。我々は、「時間」と言うとき、この言葉を1日のサイクルの中における特定の点としても（「電車は13時に到着する」）、異なる単位と結び付いた時間間隔を指すとき（「そこまで電車で3時間かかる」）にも使う【訳注7】。それ以外にも、ひと月を構成する日数も月によって違うし、1年間の日数ですら一様ではない。ひと月が何週かも固定的ではない。

標準SQLは2つの基本的な時間のデータ型を持っている。特定の時刻を表すデータ型（DATE、TIME、TIMESTAMP）と持続的な間隔を表すデータ型（DAY、HOUR、MINUTE、SECOND）である。標準SQLはまた、これらのデータ型に対する演算子を一通り揃えている。だが、時間に関しては、いまだベンダー依存の構文も多く残っていることに注意してほしい。

11.2.1　データベース内部での時間の表現

一般的に、システム内部で時間を表現する方法は2つある。1つはUNIXで採用している方法で、ある開始時刻を基準にして、コンピュータの時計が刻む間隔をカウントする64 ビット以上のバイナリ文字列を保持する。もう1 つは、（私はこれをCOBOLメソッドと呼んでいるが）年、月、時間、分、秒のフィールドを分離する方法だ。これらのフィールドは文字列やBCDなど何らかの内部的なフォーマットで保持される。

UNIXメソッドの利点は、計算が簡単なことだ。しかしエンジンは外部とのやりとりの際、必ずISO 8601と内部フォーマットの変換を行う必要がある。COBOLメソッドはこの逆で、表示には適しているが、計算が面倒だ。

11.2.2　表示フォーマットの標準

ANSIの日付フォーマットはANSI X3.30 で規定されている。これはISO 標準を包含しつつ、年を4桁、月（01～12）と日（01～31）を2桁で表示するフォーマットを追加している。このオプションはカレンダー日付フォーマットと呼ばれている。標準SQLは、ISO 8601準拠のすべて数値で表すyyyy-mm-dd hh:mm:ss[ss...]フォーマットを採用している。これは、月の表示に略称の文字（SepとかNov）を使った場合に起きる言語間の相違を回避するためだ。

いまでも、年を2桁表示するフォーマットを採用しているプログラムは多い。もともとこれは、スペースを省略するための古い習慣だ（パンチカードの時代、表示に80 列しか使えなかった）。当時のプログラマは、1900年と2000年は非常に離れているので、区別する必要性を感じていなかっただろう（2000年問題については第38章参照）。COBOLプログラムは、こうしたフォーマットに対して日付演算を行うと、エラーを返した。もしCOBOLがDATE型を持っていれば、こんなことは起きなかっただろう。リレーショナルデータベースおよび4GLのプログラミング言語はDATE型を言語に組み込んでいるので、すでにこの問題を克服している。

11.2.3　タイムスタンプの扱い

TIMESTAMP(n)は、小数点n 桁の時刻と定義されている（たとえば、TIMESTAMP(9)はナノ秒の精度を持つ）。どの程度の精度を定義できるかはハードウェア依存である。FIPS-127標準は、少なくとも小数点5桁をカバーすることを要求しているが、最近の製品は普通、7桁まではカバーする。

TIMESTAMPの使い道は2つある。1つは、あるイベントと結び付いた本当のタイムスタンプである。もう1つは、一意キーを生成する一種のシーケンシャルな連番としての使い方だ。こちらは、時間本来の性質とは関係しない。たとえばDB2 のプログラムは、タイムスタンプのマイクロ秒の数値をビット反転させて「ランダム」な数を生成してキーに使っている。この方法はもちろん、重複値の生成を防止するものではないが、ランダムに近いデータ作る簡便な方法ではある。このような方法が役に立つのは、タイムスタンプをそのまま使うと、テーブル領域において「ホットスポット」を生成してしまう場合だ【訳注8】。たとえば、口座に対する支払い日時が重要で、法的な理由から正確なタイムスタンプが求められる場合である。口座番号は口座別に一意でありさえすればいいとすれば、タイムスタンプはこれを得る簡単な方法だ。

タイムスタンプはシステムの時計を一度だけ読み、同一トランザクション内のすべての項目に同じ時間を適用する。トランザクションが完了までにかかった本当の時間が数日だったとしても問題ではない。SQLにおけるトランザクションは、それ自身1つの単位として完了するか、まったく行われないかのどちらかである。これは通常

小さなトランザクションの場合は問題にならない。だが、複雑な更新を行う必要のある大きなバッチの場合は問題になることもある。

一意な識別子のソースとしてタイムスタンプを使うことは、シングルユーザーシステムにおいては問題ない。なぜならすべてのトランザクションは直列化されており、実行時間も十分な長さを持っているため、トランザクションの実行中に時計は進み、一意性が崩れることはないからだ。だが、マルチユーザーシステムでは、それほど問題は簡単ではない。クライアントサーバーシステムでは2 つのトランザクションが異なるローカルマシンで、同時に発生することがある。そのため、ローカルマシンの時計を使うと、キー重複が生じる。これに加えて、すべてのクライアントの時刻を同期する必要もある。この同期の問題には2つの側面がある。

問題1　どうやってすべての時計を同時に開始するか？

私が言いたいのは、複数のマシンをマイクロ秒単位で同期させる技術的な問題のことではなく、サマータイムのことを忘れたクライアントが存在した場合のことだ。

問題2　すべての時計が同じ進み方をすることをどのようにして保証するか？

サーバーの時計を使ってタイムスタンプをクライアントに送信する方法は、ネットワークトラフィックを増大させるうえに、常に問題を解決するわけではない。現在採用されている解決策はNIST 時間をシグナルとしてすべての時計をセットし、同期させることだ。米国政府の公的な時間はNIST とUSNO（United States Naval Observatory）によって提供されている。両者は以

下のサイトで利用できる。

http://www.time.gov/

NISTはまたITSとACTSというサービスも提供している。これを使うことで、インターネットまたは電話回線を使ってコンピュータの時計をセットできる。これらのサービスのためのフリーソフトも上記サイトからダウンロードできる。また、NISTのWebサイトでは、時間と頻度の標準および研究についての情報を掲載している。

多くのOSにおいてシステム時間はロングバイナリの文字列として表現されている。これは、開始日からのマシンサイクルをカウントして算出している。一意なキーを生成するトリックの1つは、この値の最下位の数字をいくつか抜き出してキーに使うことだ。だが、これはトランザクション規模が大きくなるにつれてうまくいかなくなる。タイムスタンプにより多くの小数桁を追加しても根本的な問題は解決されない。というのも、問題の本質は統計的なものであり、この方法だとどの程度の確率で重複が発生するかという点にあるからだ。

電話帳の適当なページをランダムに開いてほしい。そして、適当に見つけた13桁の電話番号のうち下2桁をマークしてほしい。こうすると0から99までのランダムな数のサンプルが得られる。このとき、まったく同じ数のペアが得られる確率はどのくらいだろうか。100分の1ということはないだろう。まず1 つ目の数を得る。次に2 つ目の数を得る。このとき、2 つ目の数が最初の数と一致しない確立は100 分の99である。次に3 つ目の数を得る。これが1 つ目の数とも2つ目の数とも一致しない確立は100分の98である。この調子で計算を

続けていくと12個の数を得たときにはそのいずれもが一致していない確立は0.99×0.98×……×0.88＝0.4427……。したがって、どれか1つのペアができあがる確率は0.5572である。これは五分五分よりもやや高い。20個の数を得るまでにペアができる確率は87%にもなる。30個の数字を得るときにはペアができる確率は99%を超える。このモデルを使えば、数字を3 桁に増やしたときにもペアができる確率が50%を超えるのがいつかわかる。

キーを生成する方法の条件は、同じキーが生成されることを排除するか、または極力小さくすることだ。そして、生成されるキーの値がなるべく一様に分散することでインデックス再構築のオーバーヘッドを小さくできることだ。システム時計を使ったキー生成のアルゴリズムは、似通ったキーの値に頼っているので、こうした問題を引き起こしやすい。

こうした目的のアルゴリズムとしては、一般にハッシュが使われている。ハッシュもまた一意な値の一様な分散を狙ったものだ。違うのはハッシュアルゴリズムは結果が（コリジョンを解決したあとならば）一意であることと、再現可能であることの両方を満たさなければならないことだ。そうでないと、保存したデータを再び見つけることができない。一方、キー生成は結果が一意でさえあればいい。これが、システム時計をキー生成には使うことができて、ハッシュには使うことができない理由である。

読者もよくデータベースにデータを登録するとき、ホスト言語において擬似的な乱数発生モジュールを使うだろう。多くの乱数発生モジュールは、特定の初期値から開始する（この初期値は種（シード）と呼ば

れる）。シードを使って一連の数列を生成していく。モジュールを呼ぶたびに数列における次の値が返される。この数列は、正しく発生させた乱数列の統計的性質を部分的に持つだろうが、同じシードからは同じ数列が毎回生成される。これが「擬似」乱数と呼ばれる理由である。このことはまた、もし数列の中である数が再現したら、その数列自身が循環を始めるということだ（もっとも、これは普通問題にならない。というのは、サイクルのサイズは非常に大きく、何百何千あるいは何百万といったサイズの数を用意できるからである）。

11.2.4　時間の扱い

昔の小さなデータベースは1 つのタイムゾーンの中で動いていた。複数のタイムゾーンをカバーするデータベースは時間をUTCで保存し、UTC からローカルのタイムゾーンが何時間ずれているのかを格納するオフセットの列も必要となる。タイムゾーンはUTCを起点とするので、UTCのずれは0である。これが、標準SQLがシステムレベルのタイムゾーンテーブルを定義する方法である。ほかにもタイムゾーンの定義としては世界のタイムゾーンを3文字のコードで分類するISO標準のやり方（たとえば、米国はEST〔Eastern Standard Time〕に属する）もある。タイムゾーンからの差分を表す数字は正または負で、時間単位で設定される。だが中には15分刻みで差分を持つ奇妙なゾーンもある。

さらにサマータイムについて考える必要もある。特に商取引における「法定時間」を考える際には重要だ。米国政府はサマータイム

を採用していない州であっても、その内部における連邦所有地ではサマータイムを採用している。

もしデータベースがコンピュータ内部のハードウェア時計からタイムスタンプを得ていたとしたら、タイムスタンプには年中、重複や欠落が生じてしまう。これが、標準SQLではUTCを内部的に使っている理由だ。

時間の表記には、24 時間表記のフォーマットを使うべきだ。これはAM、PM で分けた12 時間表記よりも間違いにくい。ソートするのも簡単だし、計算上の間違いも少ない。アメリカは時間、分、秒の区切り文字にコロンを使う。これはISO標準である。だが、ヨーロッパのいくつかの国では区切り文字にピリオドを使う。そのような国では小数点にはカンマを使うので、ピリオドを使うのは自然な方法なのだ。ほとんどのデータベースではどちらを選ぶか表示用のオプションで選択できる。

時間の種類は大きく3 つある。1 つ目がある特定のイベントに結び付く時刻（「彼は13 時に着く」）。2 つ目が持続期間（「この旅行は3 時間かかる」）。3 つ目がインターバル（「電車は10 時に出発し13時に着く」）。もちろんこれら3つの概念は相互に関係している。SQL-92は3つ目の概念を表すデータ型として、`INTERVAL`データ型を導入した。このデータ型は、ある特定の時刻を表すものではなく、期間の単位である。その単位は年であったり月、日、時間、分、秒であったりする。期間には2 種類ある。1つは年月の期間。これは年と月以外の精度は持たない。ただしこの両方を使わなければいけないというわけではない（年だけ／月だけという期間もありう

る）。もう1つが日時の期間である。こちらは年と月以外のすべての精度を持つ。すなわち、日、時間、分、秒である。

11.2.5　タイムゾーンとサマータイム

法律で定められたタイムゾーンを採用した最初の例は、1880 年代の英国国有鉄道（British Railways、現British Rail）である。カナダ、米国政府がそれに続いた。このタイムゾーンの確立には、カナダの鉄道技術者だったサンフォード・フレミングの貢献が大きかった。フレミングは、1884 年にワシントンで開かれた子午線国際会議の開催に尽力した人物である。この会議では、現在まで続く国際標準時のシステムが作られた。タイムゾーンにおける標準時は、米国では1918 年に標準時法として成立した。それ以後、1966 年までは同法は運輸省の管轄だった。タイムゾーンの境界は、最初に導入されたときから大きく変わっており、現在でもときおり変更されている。

サマータイムは第1次世界大戦中に欧州で採用されたのが最初だ。これが最初に導入された理由は、戦時におけるエネルギー節約の手段としてだったが、その後米国などにも広まり、第2 次世界大戦後も制度として残った。これらの規約も時間とともに変化してきたが、システム構築において必要なのは、歴史的経緯よりも、単純に現行制度がどうなっているか、どのようにデータベースのスキーマ情報テーブルを変更するかという情報である。

とりわけ、地球上にはタイムゾーンのオフセットが1時間単位ではない地域がある点に注意が必要だ。これに対応するため、標準SQLはタイムゾーンを分単位で設定する構文を持っている。願わくば、こ

のような変則的なルールは、10 進でない通貨単位とあわせて、私の生きている間に消えてほしいものだ。

タイムゾーンが分単位で設定される地域については、以下のサイトで確認できる。

http://www.timeanddate.com/time/

タイムゾーンの「時」のパートは–12 から14、「分」のパートは–59 から59 の間の値となる。ただし、「時」パートが–12 か14 のときは「分」パートは00 である。タイムゾーンの宣言は、TIMEまたはTIMESTAMP列で、次の構文を使って行う。

```
[WITH | WITHOUT] TIME ZONE
```

たとえば、DDLは以下のようになる。

```
CREATE TABLE Foobar
(start_date TIME WITH TIME ZONE NOT NULL,
 end_time   TIMESTAMP WITHOUT TIME ZONE);
```

11.3

INTERVAL 型

INTERVAL 型は、年月または日時の期間を表すデータ型だ。DAY やHOURといった1 語だけで記述することもあれば、HOUR TO SECONDのような範囲を複数の語で記述することもある。整数ではなく文字列で表記されるのはそのためだ。以下にこのデータ型のBNF定義を示す。

INTERVAL型は期間を表すデータ型だ。INTERVAL型は2種類に分類される。1つがカレンダー日付を扱うもの、もう1つが時刻を扱うものだ。SQLにおいて、時間型の値の単位は「フィールド」と呼ばれる。これは非リレーショナルなファイルシステムにおける「フィールド」とは異なるものなので、混同しないでもらいたい。

INTERVAL型で使用される予約語の一覧は、表11.1のとおりだ。

表11.1 ANSI/ISO標準におけるINTERVAL型

フィールド	値の取りうる範囲
YEAR	0001 ～ 9999。これはISO-8601標準に従っている
MONTH	01 ～ 12。12は1年とみなすこともできる
DAY	01 ～ 31。各月の妥当な日付でなくてはならない
HOUR	00 ～ 23。24は1日とみなすこともできる
MINUTE	00 ～ 59
SECOND	00 ～ 59.999...。スケールは実装依存

INTERVAL 型の値は、数値ではなく文字列として記述する。つまりシングルクォーテーションで囲むのだ。これは、ISO-8601 標準における日付と時刻の表記（日付と時刻をダッシュでつないだ文字列）を考えれば不自然なものではない。

INTERVAL 型は、YEAR やMONTH のように1 つの予約語を単独の単位として使うこともできるし、粒度に応じて、YEAR TO MONTH のように2 つの予約語を組み合わせて使うこともできる。年月レベルの粒度では次のような組み合わせがありうる。

表11.2　年月単位のインターバル

予約語	粒度
YEAR	年
MONTH	月
YEAR TO MONTH	年月

YEAR TO MONTHは、たとえば'0007 03'と指定すれば「7年3か月」の意味になる。

同様に、日と時刻の粒度に対しては、次のような組み合わせになる。

表11.3　日と時刻単位のインターバル

予約語	粒度
DAY	日

HOUR	時間
MINUTE	分
SECOND(s)	秒。小数も定義できる
DAY TO HOUR	日、時間
DAY TO MINUTE	日、時間、分
DAY TO SECOND(s)	日、時間、分、秒。秒には小数も定義できる
HOUR TO MINUTE	時間、分
HOUR TO SECOND(s)	時間、分、秒。秒には小数も定義できる
MINUTE TO SECOND(s)	分、秒。秒には小数も定義できる

INTERVAL型を使うときのサンプルを次に示す。

```
SELECT CURRENT_TIMESTAMP + INTERVAL '+7' YEAR,
       CURRENT_TIMESTAMP + INTERVAL '-3' MONTH,
       CURRENT_TIMESTAMP + INTERVAL '0007 03' YEAR TO MONTH,
       CURRENT_TIMESTAMP + INTERVAL '+5' DAY,
       CURRENT_TIMESTAMP + INTERVAL '-5' HOUR,
       CURRENT_TIMESTAMP + INTERVAL '12' MINUTE,
       CURRENT_TIMESTAMP + INTERVAL '3' SECOND,
       CURRENT_TIMESTAMP + INTERVAL '1 12' DAY TO HOUR,
       CURRENT_TIMESTAMP + INTERVAL '1 12:35' DAY TO MINUTE,
       CURRENT_TIMESTAMP + INTERVAL '1 12:35:45' DAY TO SECOND,
       CURRENT_TIMESTAMP + INTERVAL '01:12' HOUR TO MINUTE,
       CURRENT_TIMESTAMP + INTERVAL '01:12:35' HOUR TO SECOND,
       CURRENT_TIMESTAMP + INTERVAL '01:12' MINUTE TO SECOND
  FROM Dummy;
```

HOUR TO MINUTEとMINUTE TO SECONDの文字列はどちらも'01:12'だが、これは異なる意味を持つことに注意してほしい。前者は「1時間12分」だが、後者は「1分12 秒」だ。タイムスタンプの文字列にはまた、UTCからローカル時間に変換するためにタイムゾーンのインターバルを含めることもできる。

11.4

時間の計算

ほとんどの製品はDATE型を持っている。だが使用できる関数にはかなり差がある。一般的な関数として、整数や文字列から日付を構成する関数、特定の日付から月、日、年といった構成要素を抽出する関数、および表示用の若干のフォーマッティング関数がある。製品によって構文に違いはあるが、日付の算術を行うための関数もある。たとえば、日付に日数を足したり引いたりして新しい日付を作る、日付同士を引き算して間の日数を得る、といった関数だ。

表11.4は、標準SQLにおける<datetime>と<interval>型を算術に使うときの意味のある組み合わせの一覧だ。日付の計算には、タイムゾーンと計算対象の日付の相対的な精度についてのルールもあるが、これは直観的に明らかなので省略する。

表11.4　<datetime>と<interval>の有意味な組み合わせ

<datetime> - <datetime> = <interval>
<datetime> + <interval> = <datetime>
<interval> (* または/) <numeric> = <interval>
<interval> + <datetime> =<datetime>
<interval> + <interval> = <interval>
<numeric> * <interval> = <interval>

システム時計から現在時刻を取得して返すCURRENT_DATEという標準関数がある。ただし、まだ実装依存のTODAY、SYSDATE、NOW()、getdateといった名前の関数も使われている。また、製品によっては、日付を入力すると曜日を返す関数を持っている製品もあるかもしれない。これはDOWやWEEKDAYと呼ばれている【訳注9】。

なお、標準SQLは、「現在」を得るために以下の関数を用意している。これらの機能はいずれも自明だろう。

```
CURRENT_DATE
CURRENT_TIME [(<time precision>)]
CURRENT_TIMESTAMP [(<timestamp precision>)]
```

11.5

時間データモデルの性質

アリゾナ大学のリチャード・T・スノッドグラスは、時間データベースの世界的権威である。彼の絶版の著書『Developing Time-Oriented Database Appli cations in SQL』（Morgan Kaufmann、1999）は、大学のWebサイトで読むことができる。

http://www.cs.arizona.edu/~rts/tdbbook.pdf

彼はまた1998 年に現場のエンジニア向けに『Database Programming and Design』誌（Volume 11, Issues 6–10）に一連の記事を寄稿した。現在、Morgan-Kaufmann 社のデータベースシリーズには、時間データを扱ったものが含まれている。『How to Design, Update and Query Temporal Data』（Tom Johnston、Randall Weis 著、Morgan Kaufmann、2010）。本書では、この2冊の優れた本の複雑な内容を要約するつもりはないが、このデータのモデルについて、一般的な解説を行おうと思う。

時間データはどこにでも存在する。データベースアプリケーションのコードには、50行に1回時間データが使われるという推計もある。データウェアハウスは、その定義上、時間によってデータが変化していく。ラルフ・キンボールは、すべてのデータウェアハウスは時間ディメンションを持つと断言している。時間によって変化することが、データに価値を与えているとも言えるくらいだ（誰が変化しないデータを逐一分析するだろう？）。

11.5.1　期間のモデリング

時間は連続体である。これは、任意の2つの時刻の間に無限個の時刻の点を含むということだ。ISO標準において採用されているモデルは半開区間のインターバルである。これは、開始時刻の点は含むが、終了時刻の点は含まない。このインターバルを表すのには、通常、開始点に● 、終了点に○を持つ線分で表す。たいていの商取引においては、日単位の粒度があれば十分だろう。

サンプルデータとして、商品価格の履歴を管理するテーブルを考えよう。価格は開始日から終了日の期間において有効になり、開始日と終了日のペアでその期間を表す。また特別に、現時点の価格については終了日にNULLを使う。

```
CREATE TABLE PriceHistory
(upc CHAR(13) NOT NULL -- 業界標準コード
     REFERENCES Inventory(upc),
 price_prev_date DATE NOT NULL,
 price_start_date DATE DEFAULT CURRENT_DATE NOT NULL,
 price_end_date DATE, -- NULLは現在を意味する
     CHECK(price_start_date < price_end_date),
     CHECK (price_start_date = price_prev_date + INTERVAL '1' DAY), -- 歯抜けを防ぐ
     PRIMARY KEY (upc, price_start_date), item_price DECIMAL (12,2) NOT NULL
     CHECK (item_price > 0.00),
 etc.);
```

PriceHistory：価格履歴テーブル　upc：商品コード　price_prev_date：1つ前の開始日　price_start_date：開始日　price_end_date：終了日　item_price：価格

明らかなことだが、現時点の価格は、1つの商品につき1つしか存在しない。このビジネスロジックを実装することを考えよう。「1 つの商品は1 つしかNULL を持たない」という制約は、技術的にはトリガーを使っても書けるが、本来は次のように制約で実現するべきだろう。

```
CHECK (COUNT(*) OVER (PARTITION BY upc)
     = COUNT(price_end_date) OVER (PARTITION BY upc) +1)
```

ある時点における価格を得るには、BETWEEN述語を使えばよい。

```
SELECT *
  FROM PriceHistory AS H
 WHERE :sales_date BETWEEN H.price_start_date
                       AND COALESCE (price_end_date, CURRENT_DATE);
```

現在のデータをビューで持つのも良い考えだ。

```
CREATE VIEW CurrentPrices
AS
SELECT *
  FROM PriceHistory
 WHERE price_end_date IS NULL;
```

さて残る問題は、テーブルを更新し、新しい行を挿入するストアドプロシージャのコードを書くことだ。これは単一のMERGE文で行うこともできるし、SQL/PSMで書いても短い。

```
CREATE PROCEDURE UpdateItemPrice
(in_upc IN CHAR, new_item_price IN DECIMAL)
IS
BEGIN
    UPDATE PriceHistory
       SET price_end_date = CURRENT_DATE
     WHERE upc = in_upc;

    INSERT INTO PriceHistory
                (upc, price_prev_date, price_start_date, price_end_date, item_price)
           VALUES (in_upc, CURRENT_DATE, CURRENT_DATE + INTERVAL '1' DAY, NULL,
                   new_item_price);
END;
```

このプロシージャは、新しい価格の開始日として「CURRENT_DATE + INTERVAL '1' DAY」を指定しているので、

明日以降で新しい価格が有効になる。このように前の期間の終了日と次の期間の開始日が重ならないようにすることで、比較条件でBETWEENを使うことができる。

履歴テーブルのありがちな設計ミスとして、ジャーナルファイルをそのままモデリングしたテーブルというのがある。ジャーナルというのは、1 枚に1 つの商品の処理が書かれたクリップボード用紙であり、データの歯抜けなどもある。毎日細かい入出庫の記録を取る場合には、ジャーナルに記録されることが多いのだが、これをそのまま履歴として保存すると、しょせんは差分の積み重ねでしかないので、ある時点の状態を把握することが極めて難しくなる。

これを、きちんとした――つまり、(start_date, end_date)という期間の列を持ち、各商品が同数の在庫を持つような――履歴テーブルに変換したい。以下はアレハンドロ・メサの考えた方法だ。

```
CREATE TABLE InventoryJournal
(journal_date DATE NOT NULL,
 item_id CHAR(2) NOT NULL,
     PRIMARY KEY (journal_date, item_id),
 onhand_qty INTEGER NOT NULL);
```

InventoryJournal：在庫ジャーナルテーブル　item_id：商品ID　onhand_qty：現在の在庫数

```
WITH ItemGroups
AS
(SELECT journal_date, item_id, onhand_qty,
        ROW_NUMBER() OVER(ORDER BY item_id, journal_date, onhand_qty)
         - ROW_NUMBER() OVER(PARTITION BY item_id, onhand_qty
                             ORDER BY journal_date) AS item_grp_nbr
   FROM InventoryJournal),
QtyByDateRanges
AS
(SELECT MIN(journal_date) AS start_date,
        MAX(journal_date) AS end_date,
        item_id, onhand_qty
   FROM ItemGroups
  GROUP BY item_id, onhand_qty, item_grp_nbr)
SELECT start_date, end_date, item_id, onhand_qty
  FROM QtyByDateRanges;
```

このクエリのやっていることは、サンプルデータを使って具体的に見たほうがわかりやすいだろう。在庫数に変化が生じた場合（商品「BB」）は、異なる行にデータを出力する。

```
INSERT INTO InventoryJournal
VALUES('2015-01-01', 'AA', 100),('2015-01-01', 'BB', 200),
      ('2015-01-02', 'AA', 100),('2015-01-02', 'BB', 200),
      ('2015-01-03', 'AA', 100),('2015-01-03', 'BB', 300);
```

結果

```
start_date    end_date      item_id  onhand_qty
=============================================
'2015-01-01'  '2015-01-03'  'AA'     100
'2015-01-01'  '2015-01-02'  'BB'     200
'2015-01-03'  '2015-01-03'  'BB'     300
```

11.5.2　期間同士の関係

複数の(start_time, end_time)ペアの間には、いくつかの関係が成立する。タイムラインの線分を描いて確認してみよう。T1と

T2というペアを考えると、関係のパターンには以下のようなものがある。

1. T1とT2はオーバーラップしない（図11.1）。つまり、いかなる時刻の点も共有していない。コードで書けば次のように表せる。

```
((T1.end_time < T2.start_time) OR (T1.start_time > T2.end_time))
```

2. 2つの期間が隣接する（図11.2）。これは、

```
(T1.end_time = T2.start_time) OR (T2.end_time = T1.start_time)
```

   である。このケースは多くの場合、`(T1.start_time, T2.end_time)`または`(T2.start_time, T1.end_time)`という単一の期間にまとめられる。
3. T2がT1に含まれる（図11.3）。これはコードで書けばこうだ。

```
((T1.start_time <= T2.start_time) AND (T2.end_time <= T1.end_time))
```

4. T1とT2 がオーバーラップする（図11.4）。これは`OVERLAPS` 述語で判定できる。この述語については第20 章で詳しい定義を見る。この述語の定義を自分で書いてみるのも良い練習になる。

もしこうした関係を頻繁に扱うことがあるなら、あらかじめ関数を作っておくと、コーディングもそれを読むのも便利だろう。その関数の一般的な形式は次のようになるだろう。

```
<function name> (IN start_time_1 DATE, IN end_time_1 DATE, IN start_time_2 DATE,
IN end_time_2 DATE)
```

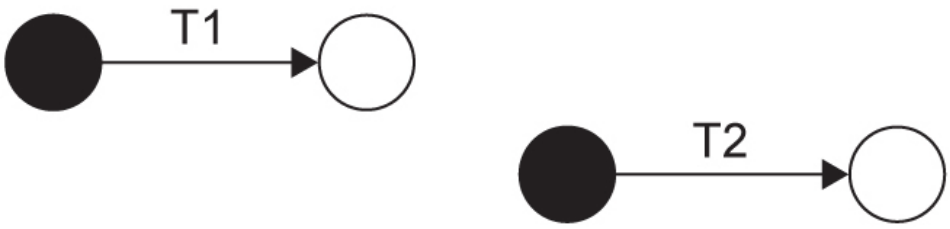


図11.1　T1とT2はオーバーラップしない



図11.2　2つの期間が隣接する

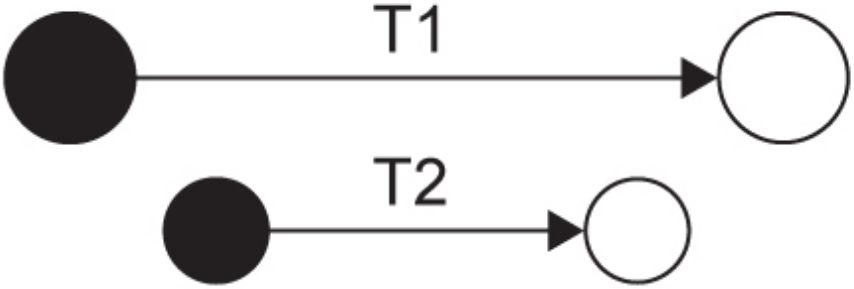


図11.3　T2がT1に含まれる

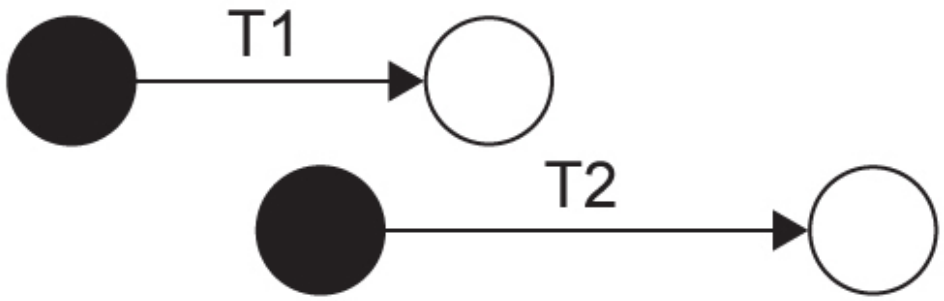


図11.4　T1とT2が部分的に重なる

【訳注1】　国際単位系では、1 秒は「セシウム133 原子の基底状態の2 つの超微細準位（F = 4、M = 0、およびF = 3、M＝0）間の遷移に対応する放射の周期の9192631770倍の継続時間」と定義されている。

【訳注2】　AUCはローマ建国紀元で、古代ローマが建国されたとする紀元前753年を元年とする紀年法。

【訳注3】　4年で約1日ずれることから、365×4≒1,461である。

【原注1】　省略形AD（Anno Domini：ラテン語で"我が王の年"）とBC（Before Christ）は、ISO 標準ではそれぞれCE（Common Era、西暦紀元）およびBCE（Before Common Era）に置き換えられる。これは、標準を脱宗教化するためだ。

【訳注4】　この法律が起案された理由の1つは、大英帝国と近隣諸国の暦が統一されていないため、商取引において混乱が多かったことである。当時、大英帝国は3 月25日から新年が開始されていたが、近隣諸国は1 月1日を開始にしていることが多かった。そのため、大英帝国も1月

1日を開始とすることが主な狙いだった。この法律によって、大英帝国の1751年は282日しかなかった。

【訳注5】　共通の標準時を使う地域全体を「等時帯」または「タイムゾーン（time zone）」と呼ぶ。ある地域の標準時は、UTCとオフセットで示すことが多い。日本はUTC＋9である。オフセットが1時間単位にならない地域の例としては、ニュージーランド（UTC+12:45）、ベネズエラ（UTC–4:30）などがある。

【訳注6】　SQL:2011では、テーブル定義のDDLに、適用時間とトランザクション時間を定義するための機能が追加された。この概念の区別はスノッドグラスらに由来する

【訳注7】　日本語では特定のタイムスタンプを表す言葉として「時刻」という便利な言葉がある。本書でもこの意味での時間を、なるべく「時刻」という言葉で表記することで、間隔を表す「時間」と区別している。

【訳注8】　タイムスタンプなど、近隣の値がよく似たシーケンスを一意キーとして使うと、物理的に同じ領域へのI/Oが集中して性能的なボトルネックが発生することがある。このI/Oの集中する領域のことを「ホットスポット」と呼ぶ。このような問題を避けるために、あえて一意キーをランダム化することでI/O分散を図るテクニックがある。著者がここで解説しているのはその方法である。ただし、これは物理レベルの問題をモデリングと

いう論理領域で解決するタイプの方法であり、著者本来のポリシーには反していると思われる。

【訳注9】　たとえば、`DOW`関数はPostgreSQLが、`WEEKDAY`関数はMySQLが持っている。

12

文字列型

Character Data Types

SQL-89は、CHARACTER(n)およびCHAR(n)のデータ型を定義した。これはどちらも印字可能なn 個の固定長文字列を表す。いくつかの実装では制御文字列をこのデータ型に含むことができるようにしているが、あまり一般的な拡張とは言えない。通常、許される文字はASCIIまたはUnicode のキャラクタセットに限られる。そして、多くの実装はソートのための照合順序を持っている。

SQL-92 はVARYING CHARACTER(n)およびVARCHAR(n)を追加した。これもいまは多くの実装に存在する。VARCHAR(n)は1～n個までの印字可能な文字を表す。これは重要なことだ。というのも、SQLは長さがゼロの文字型の列を許していないことを意味するからだ。だが、ベンダーによってはこれを許しているものもある。そうした製品では空文字を格納することができる【訳注1】。

SQL-92 では、NATIONAL CHARACTER(n)およびNATIONALARYING CHARACTER(n)のデータ型も追加している（それぞれNCHAR(n)およびNVARCHAR(n)と略す）。これらはISOで定義されているUnicode のキャラクタセットのうち印字可能な文字を格納する。これらのデータ型においてリテラル値はN'<string>'という構文を使う。

SQL-92はまたDBAが照合順序を独自に定義する権限も許している。Unicodeコンソーシアム（http://www.unicode.org/）はUnicode標準を維持しており、内容を書籍にまとめている（『The Unicode Standard, Version 5.2.0』The Unicode Consortium、2009）。あるいは次のサイトでも読める。

http://www.unicode.org/versions/Unicode5.2.0/

SQL:2006 になると、UnicodeとXMLを扱うために多くの機能を追加している。だが、基本的な文字列操作はベンダーの実装している機能に比べると極めてシンプルなままである。本章では、UnicodeとXMLについては多くを触れない。というのも、ほとんどの現場のSQLプログラマはASCIIまたは、その国のキャラクタセットだけをデータベースで扱うことが多いからだ。

12.1

SQL の文字列における問題

文字列を扱う方法はプログラミング言語によって違う。そのため、SQLを学ぶときは他の言語で覚えたことをいったんすべて忘れてほしい。文字列というのは、プログラマにとって問題の多い分野なのだ。

SQLでは、文字列はシングルクォーテーション（'）で囲まれた印字可能な文字のことである。昔の製品ではダブルクォーテーション（"）とシングルクォーテーション（'）のどちらも使うこともできた。SQL では2 つの連続したシングルクォーテーションを使うと1 つのシングルクォーテーションとみなす。ダブルクォーテーションは予約語であっても文字列として使用可能にするために使われる。

キャラクタセットは3 つに分類できる。1 つが国家または国際標準が定義するもの。2 つ目が実装が提供するもの。3 つ目がアプリケーションが提供するもの。すべてのキャラクタセットが必ずデフォルトで空白文字を持っている。アプリケーションによって定義されるキャラクタセットは、アプリケーションによって選ばれた任意のスキーマに存在するよう定義することができる。標準または実装によって定義されたキャラクタセットはカタログのインフォメーションスキーマに存在する。これは、標準および実装によって定義された照合順序の場合も同様である。それぞれのキャラクタセットに対してデフォルトの照合順序が存在するが、追加の照合順序を定義することもできる。これは、2つ以上の照合順序が使われている言語においては重要なことだ。たとえば、ドイツ語の辞書ではウムラウトの付いた「öf」は「of」より前に来るが、電話帳では逆の順序が使われ

ている。現在のUnicodeの照合順序のルールについては、以下のWebサイトを見ておくといいだろう。

http://userguide.icu-project.org/collation

12.1.1　文字列の同値性に関する問題

どのようなプログラミング言語においても長さと各文字の位置が正確に対応している場合を除いて、文字列の比較方法は異なる。

最初の問題は大文字と小文字を等しいとみなすか否かである。文字の大小の概念を持っているのはラテン文字、ギリシャ文字、キリル文字、アラビア文字である。このうち大文字と小文字だけを持っているのが前三者、アラビア語は頭字、中字、尾字の3 種類を持つ【訳注2】。SQLを含むほとんどのプログラミング言語は、プログラムのテキストにおける大文字小文字の区別を無視する。ただしデータの中における大文字小文字の区別はこの限りではない。いくつかの実装は、DBAが大文字と小文字をどのように扱うかシステムの設定パラメータで選べるようにしている。

標準SQLは大文字と小文字を変換する2つの関数を持っている。

- **LOWER(<string expression>)**：引数で渡された文字列に含まれるすべての文字を小文字に変換する
- **UPPER(<string expression>)**：引数で渡された文字列に含まれるすべての文字を大文字に変換する

ほとんどの実装では、この2つ関数は昔から存在するが、名前が異なっていることが多い。

長さの違う文字列の同値性テストは、まず最初に短い文字列の末尾（右側）に空白を埋めて、文字列同士が同じ長さになる状態にする。次に、1 文字ずつ同じ位置の文字を比較していき、すべての位置について完全に一致したならば同一文字列とみなす。1 つでも異なる箇所があれば、不一致とみなす【訳注3】。

12.1.2　文字列の順序に関する問題

SQL-89 は照合順序については何も語っていない。実際にはほとんどすべての実装がASCII またはUnicodeを使っており、どちらもISOの用語でいうRoman Iキャラクタセットである。中には順序のオプションのディクショナリまたはライブラリ（たとえば、アルファベットの中で大文字小文字を混在させた順序：A, a, B, b, C, c, ……）を持つ実装もある。多くの実装は、適切なISO標準に基づいたその国の言語オプションを提供している。

このオプションは、言語によっては非常に複雑になる。たとえば、ノルウェー語はキャラクタセットという点では普通のISO標準とまったく同じだが、ソート順序は標準と異なる。ドイツ語はかつてドイツとオーストリアでソート順序が異なっていた。スペイン語はchとllを、あたかも2 つで1 つの文字であるかのように扱い、ソートしない。そのため、読者の使う製品のISO Unicodeの実装はよく調べたほうがいい。

標準SQLは、比較に使う照合順序をDBA が定義できるようにしている。この機能はグローバル化が進むにつれ、よりメジャーになっていくだろう。しかし、ベンダーが実際にサポートしている範囲については、注意が必要だ。

12.1.3　文字列をグループ化するときの問題

SQLの同値性テストは2 つの文字列の短いほうを空白で埋めてから行われる。一方、グルーピングにおいては空白に対するルールはあいまいだ。そのためVARCHAR(n)に対してGROUP BY を行うと思わぬ結果が生じることがある。サンプルを使って確認してみよう。

```
CREATE TABLE Foobar (x VARCHAR(5) NOT NULL);
INSERT INTO Foobar VALUES ('a'), ('a '), ('a '), ('a ');
```

このテーブルに次のようなクエリを実行する。

```
SELECT x, CHAR_LENGTH(x), COUNT(*)
  FROM Foobar
 GROUP BY x;
```

CHAR_LENGTH(x)の値は製品によって変わる。主な製品の結果は以下のようになっている（なお、CHAR_LENGTH関数は製品によって名称が違う）。

Oracle、PostgreSQL

```
x   CHAR_LENGTH(x)  COUNT(*)
============================
a   2               3
a   1               1
```

DB2、SQL Server、MySQL

```
x   CHAR_LENGTH(x)  COUNT(*)
============================
a   1               4
```

1が返されるケースとしては、'a'がそのまま含まれるほか、'a'の空白が削除されて長さ1 とみなされる場合がある。一方、2が返るのは、'a 'の空白を削除せずそのまま長さを測ったからだ。この例からわかるように、同値性テストと違ってグルーピングの際には、ルールがあいまいなため製品間で異なる動作になる。この問題を回避したければ、テーブルに空白を登録できないような制約を付与する必要がある。

12.2

標準の文字列関数

SQL-92 は文字列関数を定義しており、それらは多くの実装でサポートされている。ただし、いくらかベンダー独自の構文にアレンジされている。こうした独自構文は今後も維持されていくと思うが、それ以上に、ベンダーは新しいリリースのたびに標準SQL の構文を追加していく。読者は可能な限り標準の構文を使うよう心がけてほしい。以下、代表的な機能を見ていこう。

文字列連結には||という演算子を使う（この記号はPL/I から借りてきた）。Sybase/SQL Serverファミリーではこの代わりに+演算子、MySQLではCONCAT(s1, s2)関数を使う。

SUBSTRING(<string> FROM <start> FOR <length>)関数は3 つの引数を取る。左の文字列がオリジナルの文字列、中央の<start>が開始文字列、右の<length>が抽出する文字列の長さである。SQL:2006では、対象をバイナリ文字列に拡張した関数が追加されたが、よほど特殊な用途でないと使わないだろう。またDB2などの実装は、LEFT関数とRIGHT関数を持っている。LEFT 関数は、文字列を左から指定された文字数分抽出して返す。RIGHT 関数はその反対バージョンだ。

前節でも触れたUPPER(<string>)とLOWER(<string>)は、与えられた文字列をすべて大文字、またはすべて小文字に変換する。

TRIM([[<trim specification>][<trim character>] FROM]<trim source>)関数は、<trim source>の文字列から文字列の左端または右端の<trim character>を削除した結果を返

す。<trim specification>には、LEADING、TRAILING、BOTHのいずれかのキーワードを指定する。それぞれ、左端、右端、両方を意味する。もし<trim character>を指定しなければ、デフォルトで空白が指定されることになる。多くの製品は<trim character>オプションを持っておらず、空白だけしか削除しない。かつては、左端の空白を削除するLTRIM、右端の空白を削除するRTRIMを持っていた製品も多かった。

TRANSLATE(<string expression> USING <translation>)は、ある文字を別の文字に変換する関数である。この関数を使うには、TRANSLATIONという特別なスキーマオブジェクトを作る必要があり、そこに変換ルールを記述する。

CHAR_LENGTH(<string>)あるいはCHARACTER_LENGTH(<string>)は、文字列長を整数で返す。多くの製品は、この名前ではなく、単純にLENGTH という名前になっている【訳注4】。標準SQLには、これから派生した次のような関数も一応あるのだが、ほとんどの製品は持っていない（ベンダーはデータベースはASCIIかUnicodeしか扱わないと考えているようだ）。

BIT_LENGTH(<string>)は文字列長をビット数で返す。

OCTET_LENGTH(<string>)は文字列長をオクテットで返す。オクテットは8ビットを1つとする単位である。これは単純にTRUNCATE (BIT_LENGTH (<string>)/8)と同じだ。

POSITION(<search string> IN <source string>)は、文字列の中で検索したい文字列が最初に登場する位置を整数で返す。

もし検索したい文字列が見つからなければ、0 を返す。<search string>と<source string>が同じだったら1を返す。同じ機能を持つ実装依存の関数に、DB2のLOCATE関数とSQL ServerのCHARINDEX関数がある。

12.3

一般的なベンダー拡張

SQL-89では文字列型に関する関数は定義されなかった。標準SQLが多くの実装の間で当たり前になっていた関数を追加するのはもう少しあとのことである。だが、実装依存の関数にも有用なものがあるため、それをここで見ていくことにしよう。

多くのベンダーが、内部フォーマットを文字列に変換することによって日付フォーマットを表示用に整形する関数を持っている。4GLと結び付けられたSQLを持つベンダーは、4GLがこうした機能を持っているからという単純な理由で、こうした拡張を行っている。

これらの関数は、一般的にはCOBOLスタイルのパラメータかグローバルにセットするデフォルトのフォーマットを使う。こうした変換はある程度は標準SQLのCAST関数を使って行われる。しかし、SQLは出力文を持っていないので、表示用の整形はベンダー拡張によって実現されることになるだろう。

REVERSE関数は、入力された文字列を逆の順序に変換する。

REPEAT関数またはREPLICATE関数は、指定された文字列を複数回繰り返した文字列を出力する。この特殊版として、空白を繰り返すSPACE関数もある。

ベンダーが提供する機能にはないのだが、あれば便利な機能として、連続した空白文字を1つの空白に変換する、というものがある。これは次のようなユーザー定義関数で記述できる。

```
CREATE FUNCTION BigSqueeze (original_str IN VARCHAR)
RETURN VARCHAR
IS
    marker_char CHAR(1);
    return_str  VARCHAR(2000);
BEGIN
    marker_char := '$';  -- 入力文字列には現れないマーカー

    return_str :=
           CASE WHEN INSTR(original_str, '  ') = 0
                THEN TRIM(BOTH ' ' FROM original_str)
                ELSE REPLACE (
                        REPLACE (
                          REPLACE (TRIM(BOTH ' ' FROM original_str),
                                    '  ', ' ' || marker_char),
                          marker_char || ' ', ''),
                        marker_char, '') -- マーカーを削除する
            END;
    RETURN return_str;
END;
```

この関数を使うと、たとえば「a b c d」といった入力文字列に対して、「a b c d」という空白を1つに「短縮」した文字列を返す。この関数の動作は以下のとおりだ。もし、入力の文字列に2 つ以上の空白が存在した場合は、両端の空白を取り除くとともに三重の入れ子になったREPLACE関数を呼ぶ。REPLACE関数によって、2つの空白は空白＋マーカーに変換され、最後に、マーカー＋空白は空文字になる。

マーカーは入力文字列の中には含まれない文字でなければならない。したがって、マーカーは変数よりは定数として宣言しておくほうがよい。

12.3.1　音声マッチング

人名はデータベースの設計者にとって常に悩みの種である。名前は可変長だし、奇妙なスペリングのものもあるし、一意でもない。

アメリカ人の名前は民族的出自によって非常に多様で、同じ発音なのにスペルが違う名前やその逆のケースも多い。およそ規則性に欠けるため、プログラムで処理しにくいのだ。

この多様性を無視すると、読み取りや聞き取りにおける間違いの多くはつづりによるものだ。ジャンクメールを受け取ったことのある人ならわかるだろう。私が受け取ったことのあるジャンクメールの中には正しいつづりのもの以外にSelco、Selko、Celcoというものがあった。これが音声上の間違いである。また、タイプミスから生じるものにCellro、Chelco、Chelkoというのもある。このような問題を解決するには、似た発音の名前を見分ける音声アルゴリズムが必要になる。

Soundex 関数

Soundex アルゴリズムを使った関数は、そのアルゴリズムの名前に由来する名称を持っている。Soundexアルゴリズムは、人の名前を入力してその名前が大雑把にどんな発音をするかを特定する文字型のコードを返す。

多くの製品は、ライブラリ関数としてSoundex アルゴリズムを持っている。また、標準SQL でも、文字列関数とCASE式を使ってSoundexアルゴリズムを計算することができる。

似たような名前が必ずしも同じSoundexコードを持つわけではない。たとえば、LeeとLeigh は似たような発音をするが、異なるSoundexコードを持つ。なぜなら、このLeigh の中の無声のgはコードを持っているからである。また、似たような発音をする名前で

も、最初の文字が異なっていれば必ず異なるSoundexコードを持つ。たとえば、CarrとKarrは異なるコードを持つ。

結局のところ、Soundexは英語の発音に基づいているのでヨーロッパとアジアの人名は正しくコード化できていない。たとえば、フランス人の姓BeauxとBeauは、発音はまったく同じなので、本来は同じコードが割り当てられるべきだが、実際はそうなっていない。

また逆に、似た発音の名前ではないのに、同じSoundexコードを持つ場合もある。比較的ありふれた姓であるPowers、Pierce、Price、Perez、Parkは、すべて同じSoundexコードである。他方、100 年ほど前はPowers をPower と書くことが多かったのだが、両者は異なるSoundex コードになる。

最初のSoundexアルゴリズム

マーガレット・オデルとロバート・ラッセルが、最初のSoundexアルゴリズムの特許を取得したのが1918 年である。現在のSoundexも、彼らの作った音声分類に基づいている。彼らは、音声を両唇音、唇歯音、歯摩擦音、歯茎音、軟口蓋音、声門音の6種類に分類した。アルゴリズムはかなり明快で、入力された単語をバックトラックしたり複数回の変換を行う必要がない。これは、彼らの時代にはコンピュータがまだなかったので、人力でコード化する必要があったからだ。そのアルゴリズムは次のルールに基づいている。

1.0 単語のすべての文字を大文字に変換し、各ステップで必要なだけ右端に空白を埋める。

2.0 単語の先頭文字を保存する。

3.0　次の文字が先頭以外の場所に現れたら、すべて削除する。

A、E、H、I、O、　U、W、Y

4.0　以下のルールに従って文字を数値に変換する。ただし先頭文字はそのままとする。

1＝B、F、P、V

2＝C、G、J、K、Q、S、X、Z

3＝D、T

4＝L

5＝M、N

6＝R

5.0　ステップ4.0 で生成された文字列の中に同じ数字が連続して存在した場合は、そのうちの1つだけを保持する。

6.0　ステップ5.0 で作られた文字列から最初の4 文字を切り出す。結果は、<大文字> <数字> <数字> <数字>という形式になる。これがSoundexコードである。

この手続きに従うと、たとえば「Robert」と「Rupert」はともに「R163」というコードに変換される。

- Robertの例

1.0　Robert → ROBERT

3.0　ROBERT → RBRT

4.0　R163

- Rupertの例

1.0 Rupert → RUPERT
3.0 RUPERT → RPRT **4**
4.0 RUPERT → R163

なお、ラッセルは、次のような別バージョンのアルゴリズムも提案していた。それは、2.0において(A, E, H, I, O, U, W, Y)という文字を「9」に変換し、5.0を次のルールで置き換えるものだ。

5.1 冗長な数字を削除する。たとえば「22992345」は、「29245」になる。
5.2 すべての9を削除し、スペースを詰める。たとえば「29245」は「2245」になる。

このルールに従うと、結果の文字列の中に同じ数字が連続することがある。このバージョンの利点は、より粒度が細かくなり、名前の種類が増えたときもよく区別できるようになることだ。

Soundexの問題は、コンピュータが登場するはるか前に使われていたため、国勢調査局の職員が手で行っていたことだった。また、アルゴリズムの運用ルールも地域によって少しずつ異なっていた。たとえば、La、De、von、vanといった姓に付けられる前置詞は、通常Soundexの姓からは削除されていたが、そうしない地域もあった。DiCaprioやLaBiancaといった姓について、前置詞がある場合とない場合でSoundexコードがどう違うか、試してみてほしい。同様に、Mc、Mac、Oといった姓の最初に来る単語も削除されることが多かった。

さらに、子音とともに使われるHとWをどう扱うか、という問題もあった。1920 年のニューヨークの国勢調査では、AshcraftはいったんAscrftに変換してから、最終的にA261というコードに変換された。一方、1880 年、1900 年、1910 年に行われた国勢調査では、hを残す地域と削除する地域が分かれたほか、ラッセルの2番目のアルゴリズムを使うかどうかでも割れた。そのため、Ashcraft は地域によってはA226 に変換された。Soundex の信頼性は、名前調査の選択率0.213%に対して95.99%である。

このバージョンのSoundexアルゴリズムは、SQLのユーザー定義関数で記述することができる。複数のTRANSLATEは1つの文で書くこともできるが、分けて書いたほうがわかりやすいだろう。REPLACE関数もネストすれば1文で書けるが、あえて分けて書いている。

```
CREATE FUNCTION SoundexSQL(in_name IN VARCHAR)
RETURN CHAR
IS
    tmp_name VARCHAR(60);
    header_char CHAR(1);
    prior_name_size INTEGER;
BEGIN
    tmp_name := in_name;
    --名前を頭文字とそれ以外に分離する
    header_char := UPPER (SUBSTR (tmp_name, 1, 1));
    tmp_name := UPPER (SUBSTR (tmp_name, 2, LENGTH(tmp_name)));
    -- 母音を削除する
    tmp_name := TRANSLATE (tmp_name, 'AEHIOUWY', ' ');
    -- スペースを削除し、ゼロ埋めする
    tmp_name := REPLACE (tmp_name, ' ', '') || '0000';
    --子音を数値に変換する
    tmp_name := TRANSLATE(tmp_name, 'BFPV', '1111');
    tmp_name := TRANSLATE(tmp_name, 'CGJKQSXZ', '22222222');
    tmp_name := TRANSLATE(tmp_name, 'DT', '33');
    tmp_name := TRANSLATE(tmp_name, 'L', '4');
    tmp_name := TRANSLATE(tmp_name, 'MN', '55');
    tmp_name := TRANSLATE(tmp_name, 'R', '6');
    --重複する数値があれば削除する
    WHILE (1 = 1) LOOP
        prior_name_size := LENGTH (tmp_name);
        tmp_name := REPLACE(tmp_name, '11', '1');
        tmp_name := REPLACE(tmp_name, '22', '2');
        tmp_name := REPLACE(tmp_name, '33', '3');
        tmp_name := REPLACE(tmp_name, '44', '4');
        tmp_name := REPLACE(tmp_name, '55', '5');
        tmp_name := REPLACE(tmp_name, '66', '6');
        --文字列長が変わらなければ、重複値がなくなったため出力
        IF prior_name_size = LENGTH(tmp_name)
        THEN RETURN header_char || SUBSTR (tmp_name, 1, 3);
        END IF;
    END LOOP;
END;
```

Metaphone

MetaphoneはSoundexアルゴリズムの改良版である（初出は『Computer Language』誌，ローレンス・フィリップス著，1990）。テリー・スミスウィックがPascal で書いたものを、本人の許可を得てここに掲載する。

```
FUNCTION Metaphone (p: STRING): STRING;
CONST
VowelSet = ['A', 'E', 'I', 'O', 'U'];
FrontVSet = ['E', 'I', 'Y'];
VarSonSet = ['C', 'S', 'T', 'G'];
 { variable sound - modified by following 'h' }
FUNCTION SubStr (A : STRING;
```

```
Start, Len : INTEGER) : STRING;
BEGIN
SubStr := Copy (A, Start, Len);
END;
FUNCTION Metaphone (p: STRING): STRING;
VAR
 i, l, n: BYTE;
 silent, new: BOOLEAN;
 last, this, next, nnext: CHAR;
 m, d: STRING;
BEGIN {Metaphone}
IF (p = '')
THEN BEGIN
 Metaphone:= '';
 EXIT;
 END;
{Remove leading spaces}
FOR i:= 1 TO Length (p)
DO p[i]:= UpCase (p[i]);
{Assume all alphas}
{initial preparation of string}
d:= SubStr (p, 1, 2);
IF d IN ('KN', 'GN', 'PN', 'AE', 'WR')
THEN p:= SubStr (p, 2, Length (p) - 1);
IF (p[1] = 'X')
THEN p:= 's' + SubStr (p, 2, Length (p) - 1);
IF (d = 'WH')
THEN p:= 'W' + SubStr (p, 2, Length (p) - 1);
{Set up for Case statement}
l:= Length (p);
m:= '';
  {Initialize the main variable}
new:= TRUE;
  {this variable only used next 10 lines!!!}
n:= 1;
  {Position counter}
WHILE ((Length (m) < 6) AND (n <>l))
DO BEGIN { Set up the 'pointers' for this loop-around}
 IF (n > 1)
 THEN last:= p[n-1]
 ELSE last:= #0;
 { use a nul terminated string}
 this:= p[n];
 IF (n < l)
 THEN next:= p[n+1]
 ELSE next:= #0;
 IF ((n+1) < l)
 THEN nnext:= p[n+2]
 ELSE nnext:= #0;
 new:= (this = 'C') AND (n > 1) AND (last = 'C');
 {'CC' inside word}
 IF (new)
 THEN BEGIN
  IF ((this IN VowelSet) AND (n = 1))
  THEN m:= this;
```

```
CASE this OF
'B': IF NOT ((n = l) AND (last = 'M'))
  THEN m:= m + 'B';
 {-mb is silent}
'C': BEGIN {-sce, i, y = silent}
 IF NOT ((last = 's') AND (next IN FrontVSet))
 THEN BEGIN
   IF (next = 'i') AND (nnext = 'A')
   THEN m:= m + 'X'{ -cia- }
   ELSE IF (next IN FrontVSet)
    THEN m:= m + 's' { -ce, i, y = 's' }
    ELSE IF (next = 'H') AND (last = 's')
      THEN m:= m + 'K' { -sch- = 'K' }
      ELSE IF (next = 'H')
        THEN IF (n = 1) AND ((n+2) < = l)
        AND NOT (nnext IN VowelSet)
        THEN m:= m + 'K'
        ELSE m:= m + 'X';
  END {Else silent}
 END;
{Case C}
'D': IF (next = 'G') AND (nnext IN FrontVSet)
  THEN m:= m + 'J'
  ELSE m:= m + 'T';
'G': BEGIN
 silent:= (next = 'H') AND (nnext IN VowelSet);
 IF (n > 1) AND (((n+1) = l) OR ((next = 'n') AND
  (nnext = 'E') AND (p[n+3] = 'D') AND ((n+3) = l))
{Terminal -gned}
 AND (last = 'i') AND (next = 'n'))
 THEN silent:= TRUE;
{if not start and near -end or -gned.)}
 IF (n > 1) AND (last = 'D'gnuw) AND (next IN FrontVSet)
 THEN {-dge, i, y}
 silent:= TRUE;
 IF NOT silent
 THEN IF (next IN FrontVSet)
  THEN m:= m + 'J'
  ELSE m:= m + 'K';
 END;
'H': IF NOT ((n = l) OR (last IN VarSonSet)) AND (next IN VowelSet)
  THEN m:= m + 'H';
 {else silent (vowel follows)}
'F', 'J', 'L', 'M', 'N', 'R': m:= m + this;
'K': IF (last <> 'C')
  THEN m:= m + 'K';
'P': IF (next = 'H')
   THEN BEGIN
    m:= m + 'F';
    INC (n);
    END {Skip the 'H'}
   ELSE m:= m + 'P';
'Q': m:= m + 'K';
's': IF (next = 'H')
   OR ((n > 1) AND (next = 'i') AND (nnext IN ['O', 'A']))
```

```
 THEN m:= m + 'X'
 ELSE m:= m + 's';
'T': IF (n = 1) AND (next = 'H') AND (nnext = 'O')
 THEN m:= m + 'T' { Initial Tho- }
 ELSE IF (n > 1) AND (next = 'i') AND (nnext IN ['O', 'A'])
  THEN m:= m + 'X'
  ELSE IF (next = 'H')
   THEN m:= m + 'O'
   ELSE IF NOT ((next = 'C') AND (nnext = 'H'))
    THEN m:= m + 'T';
{-tch = silent}
'V': m:= m + 'F';
'W', 'Y': IF (next IN VowelSet)
  THEN m:= m + this;
 {else silent}
'X': m:= m + 'KS';
'Z': m:= m + 's';
END;
{Case}
INC (n);
END; {While}
END; {Metaphone}
Metaphone:= m
END;
```

NYSIISアルゴリズム

NYSIIS（The New York State Identification and Intelligence System）は、Soundex よりも信頼性が高く、選択率も良いアルゴリズムである。特に、グループ化された音声に対して効果的だ。これはYのグループをうまく扱えないが、それはYが変換されないからだ。NYSIISは10文字のアルファベットの文字列のキーを生成する。

1.0 名前の先頭文字を次のように変換する。

MAC => MCC

KN => NN

K => C

PH => FF

PF => FF

SCH => SSS

2.0 名前の末尾文字を次のように変換する。

EE => Y

IE => Y

DT, RT, RD, NT, ND => D

3.0 キーの先頭文字を名前の先頭文字とする。

4.0 残りの文字を次のルールで変換する。

a. EV =>AFまたはA, E, I, O, U =>A

b. Q $^{=>G}$ Z$^{=>S}$ M$^{=>N}$

c. KN =>Nまたは K =>C

d. SCH =>SSS PH=>FF

e. H =>もし前後の文字が子音だったら、前の文字へ変換する

f. W =>もし前の文字が母音だったら、前の文字へ変換する

g. 対象の文字がキーの最後の文字と異なれば、キーの末尾に対象の文字を追加する。

5.0 もし最後の文字がSならば取り除く。

6.0 もし最後の文字がAYならば、Yで置換する。

7.0 もし最後の文字がAならば取り除く。

NYSIIS の信頼性は、名前調査の選択率0.164%に対して、98.72%である。この方法は、ロバート・トラフトが考案したもので、ニューヨーク州の人名識別および情報システムで使われている。

12.4

カッター分類法

名前をコード化するもう1つの方法として、100年以上前から図書館司書の間で使われているものがある。書籍の分類においては、しばしば著者の名前を単純な固定長コードに変換する必要が生じる。その方法の1 つがカッター分類法（展開分類法）である。この分類の結果は、Soundexによく似ているが、目的は違っていた。図書館の性質上、コードはオリジナルの人名のアルファベット順序を保存する必要がある。だからといって、著者名をそのまま分類コードに使うには種類が多すぎるし、長さもバラバラで都合が悪い。そもそも、一意にならない。

司書がこの問題を解決するために考案したのが、カッター分類法である。これを実装したテーブルは、著者のフルネームを文字と数字から成るコードに変換する。いくつかのバージョンがあるが、昔のバージョンは、大文字と小文字を混在させていた。カッター分類による変換の例は次のようなものだ。

```
"Adams, J" → "A214"
"Adams, M" → "A215"
"Arnold" → "A752"
"Dana" → "D168"
"Sherman" → "S553"
"Scanlon" → "S283"
```

数字の分散は、（英語圏の国では）入力された著者名の分散に依存する。変換テーブルの使い方は単純で、テーブルを総なめして、

ヒットした行のコードを使えばいい。

　カッター分類法には重要な2 つの性質がある。1 つは、コードがオリジナルの名前のアルファベットのソート順序を保存していること。これはコードでソートを行うときに便利だ。もう1 つは、名前の集合が大きくなればなるほど、各コードのアルファベットのグループのサイズが、近似的に名前の集合に近くなることだ。この性質は、データベースでインデックスを作成するときに重要になる。

　カッター分類法の変換テーブルは、Webサイトで見つけることができる。プリンストン大学図書館のサイトには、名前や地名などを変換するルールが掲載されている。

http://library.princeton.edu/departments/tsd/katmandu/class/cutter.html

12.5

関数を入れ子にした文字列置換

REPLACE関数を使ったトリックの1つに、これを入れ子にするというものがある。多くの実装は、32 階層までのネストを許している（もちろん、それ以上のネストを許す実装もある）。したがって、次のように書くことができる。

```
REPLACE(
  REPLACE(
    ...
      REPLACE(foobar, 'Z', 'z'),
    ...
  'B', 'b'),
'A', 'a')
```

これはLOWER関数の実装としては読みにくくてほめられた方法ではないが、多段のネストを使うことで、ダッシュ（–）、カンマ、空白といった余計な要素を取り除くことができる。この方法の利点は、ループや再帰を必要としないため、実行が速いことだ。

【訳注1】 空文字の格納方法には、実装ごとに微妙な違いがあるため、しばしばコーディング時に混乱のもとになる。たとえば、PostgreSQLでは空文字を長さゼロの文字列（''）として格納するが、OracleはこれをNULLとして扱う。

【訳注2】 ただしアラビア語における頭字、中字、尾字は、UTF-8などにおいては文字コード上の区別は存在しない。そのため、アルファベットにおける大文字・小文字の区別という問題と必ずしも同列に論じることはできない。

【訳注3】　このルールはVARCHARを使っている場合の話だが、実装によってサポート状況が異なる。SQL Serverはこのとおりの動作をするが、OracleやPostgreSQLは、空白を埋める動作はしない。

【訳注4】　CHARACTER_LENGTHという名前なのは、PostgreSQL。SQL ServerはLEN、Oracle、DB2、MySQLはLENGTH という名前だ。

13

NULL：SQLにおける失われたデータ

NULLs: Missing Data in SQL

失われた値の扱い方についての議論は、リレーショナルデータベースの世界では取扱注意のテーマだ。リレーショナルモデルの創始者コッドは、最終的に、失われた値を表すトークンとして2 つの異なるものを使うことを支持した（『The Relational Model for Database Management, Version 2』）。1つは「未知（unknown）」（サングラスをかけた人の目の色）、もう1つが「適用不能（not applicable）」（自動車の目の色）である。一方、現在リレーショナルデータベースの指導的な論者であるクリス・デイトは、失われた値を表すいかなる汎用的トークンも使うべきではないと主張している。標準SQLは、コッドのオリジナルのリレーショナルモデルに基づいてNULLという唯一のトークンを採用している。

おそらく、コッドは――ここでもやはり――正しかったのだ。標準SQLは、ROLLUPやCUBEによって生成されるNULLも追加しているが、これはあくまで合計行を識別するためのマークであり、「本物のNULL」とは異なる「作られたNULL」である（こうしたNULLを本物のNULLと区別するための関数も存在する）【訳注1】。ディヴィッド・マクガヴァランとクリス・デイトは『A Guide to Sybase and

SQL Server』（Addison-Wesley、1992）において、次のように述べている。

> 少なくとも現在SQLで定義され実装されているNULLに関して言えば、利点よりも欠点のほうが大きい。したがって、これを使うべきではないというのが著者たちの考えだ。NULLの表示のしかたは非常に奇妙で、挙動も一貫性を欠いており、間違いと混乱の源泉である（この言葉は、SQL Serverのみならず、SQL的なNULLをサポートしているすべてのシステムに当てはまる）。

このように、NULLに関する議論においては、一方の極にNULLを厳密に表現するトークンを使うべきだと言う論者がいて、もう一方の極にNULL 完全撲滅派がいるわけだが、SQL は中道を行っている。すなわち、単一の汎用的なNULLを失われた値を表すために使っている。NULL のルール、とりわけSQL 文におけるルールについては、本書のさまざまな箇所で触れているが、本章ではNULLおよび失われた値全般について論じる。

何らかの意味において「存在しないもの」というのは、人間にとってやっかいなものだ。古代エジプト、マヤ、中国、ローマ、および数ある歴史的な文明のいかなる数体系においても、ゼロは存在しなかった。ヨーロッパでインド–アラビア式のゼロを備えた数体系が広まるには、数世紀がかかった。実際ゼロの概念は受け入れがたく、ルネッサンス初期のヨーロッパの会計事務所は「当事務所はわ

けのわからぬ新奇な表記法には飛びつかず、理解しやすいローマ数字を使い続けます」という宣伝をしていたぐらいだ。

　ゼロにまつわる概念的な問題の多くは、序数と基数の区別を知らないことから生じていた。序数は位置（順序）を測り、基数は量や大きさを測る。ゼロに対する批判的な議論は、こういうものだ。もしいかなる量も大きさも存在しないとしたら、どうやってそれを測ることができるのだろうか？ ゼロで掛けたり割ったりするのはどういう意味なのか？ こうしたことは何かが「ない」ということの表現に対する言語的混乱の一例だ。古代ギリシャには、この混乱をうまく言い表したパラドックスがある【訳注2】。

1. No cat has 12 tails.（12尾の尻尾を持つ猫は存在しない）
2. A cat has one more tail than no cat.（猫は存在しない猫よりも尻尾が1つ多い）
3. Therefore, a cat has 13 tails.（したがって、猫は13尾の尻尾を持つ）

　ゼロと同じく、空集合の概念が数学で認められるまでにも長い時間を要した。空集合において問題になったのは、以下のようなことだった。

- そもそも要素が1つもないなら、その集合というのはどうやって作るのか
- 空集合は自身の部分集合なのか
- 空集合はすべての集合の部分集合なのか

- 単一の普遍的な空集合があるのか、それとも各タイプの集合別にたくさんの空集合があるのか

現在のコンピュータサイエンスも、失われた値について独自の問題を持っている。ANSI X3（SPARC Study Group 1975）の中間報告75-02-08 では、クエリの結果または属性の値として現れうる不完全なデータを14 種類挙げている。たとえば、オーバーフロー、アンダーフロー、エラーなどなど。これらは、現実世界の失われた値をコンピュータの制約の範囲内で表現しようとしたときに起きるものだ。

本章では、さまざまなモデルにおける失われた値に対する理論とアプローチについて議論することは避けて、SQLにおけるNULLの扱い方とその根拠だけに集中する。なお、これ以降、本書ではたびたびNULLを使わないよう警告を発する。わざわざNULLの扱い方を説明しておいて、その態度は矛盾していると思うかもしれないが、NULLは薬だと思ってほしい。正しく使っている限りは有用だが、乱用すればすべてをぶち壊す。最良の選択は、可能な限り使用を避けて、どうしても使わざるを得ないときだけ、適切に使用することだ。

13.1

空のテーブルと失われたテーブル

空のテーブルまたは空のビューは、失われたテーブルとは異なる概念である。空のテーブルというのは列と制約が定義されているが、行を1つも持っていないテーブルのことだ。テーブルを作った直後はこういう状態になるし、テーブルからすべての行を削除しても空のテーブルになる。これは、完全に妥当なテーブルだ。その定義から、すべての制約は真の状態にある。

失われたテーブルは、DROP TABLE文によってスキーマから削除されたか、（名前のタイプミスなどによって）そもそも最初から存在していないテーブルである。失われたビューは、これとは少し違う。もちろん、DROP VIEW文によって削除された場合にも生じるが、ビューのもとになっているテーブルまたはビューが削除された場合にも生じる。つまり、ビューを実行時に作ることができず、クエリが失敗するケースである。もしテーブルを削除するときにCASCADEオプションを付けていれば、それに依存するビューも削除されるのでこの問題は起きない。この点については、あとでもう少し詳しく見る。

空のテーブルまたは空のビューの動作はその使われ方によって異なる。詳しくはサブクエリについて解説する章で取り上げる。一般に、空のテーブルはNULLか空集合のどちらかとして扱われるが、どちらになるかは文脈に依存する。

13.2

列における失われた値

NULL の一般的な用法は、現在は不明だがいずれ事実が判明したときには正しい値で置き換えられる状態を表現することだ。実際のところ、NULLの意味は非常に多岐にわたる。というのも、これだけが唯一SQLにおいて失われた値を表現する方法だからだ。ここでいったん基本に戻ると、データベースにおいて1 つの行は1 つのエンティティを表す。そのエンティティは1 つ以上の属性（列）を持つ。その列の値はドメインから得られる。ここで、E(A)＝Vという表記を導入しよう。Eはエンティティ、Aは属性、Vはその値を意味する。たとえば、「ジョン(髪の色)＝黒」と書けば、「ジョンの髪は黒い」という意味だ。

SQLの汎用的なNULLは、このモデルとうまく適合しない。髪の色に対するドメインと車の色に対するドメインを定義したとしよう。その場合、髪の色と、車の色は比較可能ではあり得ない。なぜなら両者は異なるドメインから得られるからである。2つのドメインを比較可能にするには、暗黙的にせよ明示的にせよ、型変換の関数を使う必要がある。これが、標準SQLにおけるドメインの仕様だが、多くの実装はまだドメインをサポートしていない。ロサンゼルスの郊外で髪の色と同じ色の車に乗っている社員を見つけようとするのは少し奇妙な要件だが、サンプルとして考えてみよう。ある社員の髪の色がNULL の場合、頭の禿げた人が通勤していた場合に一致とみなされるだろうか。あるいは、髪の毛を持っていない人が車を持っていなかった場合、その人の髪の色と車の色は同じなのだろうか。SQL-89 以降、このように2 つのNULLを直接比較したときは、エラーでは

なく、UNKNOWNを返すことになっている。ドメイン固有のNULLというのは、概念的には汎用的なNULLとは異なる。なぜなら前者の場合、少なくともどんなものがUNKNOWN であるかについてはわかっているからだ。これを先ほどの表記で表現すれば、E(A)＝NULLと書ける。意味は、我々はそのエンティティと属性を知っているが、属性の値を知らないということである。NULLのもう1つの含意は、適用不能である。これは、帳票やスプレッドシートではN/A という記号で表す。コッドはこれを"I-mark"と呼んだ。たとえば、禿げた人の髪の毛の色という属性は、髪の毛の色のドメインから得られる失われた値のNULLであるが、彼の羽の色の属性は適用不能のNULLである（羽を持っている人間はいない）。つまり、値だけでなく、属性そのものが失われている場合を示す概念だ。この失われた属性のNULLをE(A)＝V 表記法で表すとE(NULL)＝NULLとなる。

　しかし考えてみると、そもそもエンティティに属していない属性が、なぜ（エンティティをモデル化した存在である）テーブルに登場するのだろう。たとえば、医療記録を統合して統計的な目的のために1 つのテーブルに集めることを考えよう。こうしてできあがったテーブルには、男性の行においても妊娠回数という属性が存在することになる。このような場合、この適用不能の属性の扱い方を考えなければならない（医学の進歩による男性の妊娠可能性については、ここでは考慮しない――興味のある人は http://www.malepregnancy.com/を参照）。つまり、男性については妊娠回数をゼロとするか、NULLとするかである。ルールを決めたらCHECK制約でルールを強制する。

もう1 つの方法は、医学的状態を表す列と出現回数を表す列を持つ方法がある。これによって男性で妊娠回数を持つ行が現れないようにする。だが、会社のような法人ではなく個人であることがわかっている場合、性別はどう決めればよいだろう。たとえば、アレックス・モルガンのような名前の人は、男性か女性どちらだろうか。一度以上の妊娠があればアレックスは女性とわかるだろう。だが、一度も妊娠していない女性の場合はどうか。

NULL(A)＝Vというケースは少し奇妙だ。これはつまり、エンティティは知らないが探そうとしている属性は知っており、かつその値も判明している場合だ。こればいわば「赤いものは何か？」という質問をするのに似ている。日常言語の文法的にはまったく問題ないが、あまりにあいまいすぎて、データベースで処理するには難しい。

もしSQLでこういうクエリを書きたいと思うなら、テーブルと列の名前を保持したシステムテーブルを用意して、それを通常のテーブルと結合してテーブルの主キーを取得する必要がある。

厳密には、E(A)＝Vという公式について値が不明か既知かによって、2^3＝8通りの可能な組み合わせを考えることができる。しかし、そうした組み合わせの多くはほとんど役に立たないし、そもそもあまり意味がない。たとえば、NULL(NULL)＝NULLというのはまったく何もわからない、という意味のNULLだが、これはエンティティについても、その属性についても、属性の値についても何の情報も持っていないことを意味する。他方、NULL(NULL)＝Vは、値はわかっているがエンティティと属性はわからない。これは、ダグラ

ス・アダムスの小説『銀河ヒッチハイク・ガイド』に登場する有名なギャグ「人生、宇宙、およびすべてのものの意味は？」の答えが「42」であるのを彷彿とさせる。

13.3

文脈と失われた値

たとえば、「Tricolor（3色）」というドメインを作成する。含まれる値は、「赤」、「白」、「青」の3 色である。そして、このドメインに基づく列を持つテーブルを宣言し、その列に一意制約を付与する。もしこのテーブルの列が、「赤」と2つのNULLを持っていた場合、我々は、2つのNULL について情報を持っていることになる。つまり我々は、2 つのNULL が（「白」と「青」）か（「青」と「白」）のどちらかの組み合わせであることを知っている。クリス・デイトは、このようなNULL を「識別されたNULL」と呼んでいる。

もしテーブルに「赤」、「白」、NULLの3 行が存在していたら、一意制約の下ではNULLは「青」以外ではありえない。このとき、NULLが実際の値を取るタイミングまでNULLのままにしておくべきだろうか、それとも、早々とNULLを「青」に変えてしまってよいものだろうか。状況による、としか言いようがない。複数の値を1つの列に持つことはできないし、データベースがDDLの情報をもとに自動的に列の値を変えることもできない。

この「識別されたNULL」を発展させたものに、マーク付きのNULLがある。たとえば、ホテルの部屋を管理するテーブルがあるとする。このテーブルは、チェックインとチェックアウトの日付の列を持っている。チェックイン日はNOT NULL 制約が付与されているが、チェックアウト日はNULLを許可している。我々は、宿泊客が正確にいつチェックアウトするかを知らないが、その代わり、NULL同

士の関係を知っている。たとえば、団体客であれば、特定の客の集団は同じ日にチェックアウトするはずだ。こうした条件をDDLに付与することはできる。つまり、チェックイン日はチェックアウト日より必ず前だし、そうすると、特定のツアーグループは必ずチェックイン日よりあとに同時にチェックアウトするという制約である。こうしたルールをSQLを使ってデータベーススキーマに作ることは可能だが、非常に難しい。一般的な方法は、ホスト言語で手続き型のコードを使うことだ。

文脈依存型のNULLのもう1つの例は、統計的／確率的なものだ。先ほどあいまいな性別コードの例として出した「アレックス・モルガン」の例をもう一度使おう。いま、モルガン氏が生まれたのが1940～50年代だという事実がわかっているとする。この時代、アレックスという名前が女性に使われることはほとんどなかった。一方、1990～2000 年代では、アレックスという名前の半分は女性だ。こうした蓋然性に基づいた推測には、ファジー論理と確率的なデータが必要となる。だが本書では、この方法論については踏み込まない。

ディヴィッド・マクガヴァランは、失われた値を持つ列（要するにNULLを許可している列）は、なぜその値が失われているのかの理由をコード化した列もあわせて持つという方法論を提案している（McGoveran 1993, 1994a, b, c）。これは余計な手間がかかるが、たしかにこの理由列があると、状況に応じて値を含めたり除外したりするコードを書くときに便利だろう。実際、統計学者は以前か

ら、失われたデータに対して、平均や中央値で代替するという方法も使ってきた。

13.4

NULL の比較

NULLを別のNULLと比較することはできない。=、<>、>、<、いずれの演算子も結果はすべてUNKNOWNが返る。これは、SQLが2値論理ではなく3値論理を採用しているせいである。ほとんどのプログラマは3値論理でスムーズに考えることができない。しかし、少し考えてみてほしい。いま、読者の前に複数の茶色の紙袋が用意されていて、袋を開けずに中身を比較しろと言われたとする。次の文について、どう考えるだろう。「紙袋Aには紙袋Bよりも多くのツナ缶が入っている」――TRUE（正しい）かFALSE（間違い）か？「そんなの一概に答えられない」と言いたいだろう。そこで3値論理とUNKNOWNの登場である。

次のような2つのクエリを順に実行することを考える。

```
SELECT * FROM SomeTable WHERE SomeColumn = 2;
SELECT * FROM SomeTable WHERE SomeColumn <>2;
```

2値論理の世界であれば、これでSomeTableのレコードをすべて選択することができる。だが、3 値論理の世界ではそうはいかない。すべてのレコードを得るには、これに加えて次のクエリも実行する必要がある。

```
SELECT * FROM SomeTable WHERE SomeColumn IS NULL;
```

IS [NOT] NULL述語は、TRUEかFALSEしか返さない。

また、SQL:2003では、次のような特殊な述語が導入された。

```
<expression 1> IS DISTINCT FROM <expression 2>
```

これは、2 つの式が互いに非NULLで同値か、またはどちらもNULLの場合にTRUEになる。つまり、次と同値だ。

```
(<expression 1> = <expression 2> OR (<expression 1>IS NULL AND <expression 2> IS NULL))
```

この述語の否定バージョンは次のようになる。

```
<expression 1> IS NOT DISTINCT FROM <expression 2> =
NOT (<expression 1> IS DISTINCT FROM <expression 2>)
```

これは、次と同値である。

```
   (<expression 1> <> <expression 2>
OR (<expression 1>IS NULL AND <expression 2> IS NOT NULL)
OR (<expression 1>IS NOT NULL AND <expression 2> IS NULL))
```

IS [NOT] DISTINCT FROM 述語は、記述がシンプルでパフォーマンス上も最適化されるだけでなく、TRUEとFALSEしか返さないので、伝統的な2 値論理の考え方が通用するという利点がある。この述語の基礎は、SQLのもう1 つ考え方――すなわち、同値性とグルーピングの区別――にある。グルーピングは、GROUP BY 句などで使われる同値性関係であり、すべてのNULL を1つの同値なクラスとして扱う【訳注3】。

13.5

NULLと論理

ジョージ・ブールは代表作『思考の法則』において2値論理を発展させ、その名を冠したブール代数によって不滅の名を残した。とはいえブール代数は、決して唯一の論理体系というわけではない。2つよりも多い真理値を用いる多値論理の体系も多く考えられているし、SQLの検索条件は、ブール代数よりもヤン・ウカシェヴィチ（ポーランド記法の考案者でもある）の提案した3値論理の体系に近い。

2 値論理は、ONとOFFの2つしか状態を持たないバイナリコンピュータや多くの数学分野と極めて相性がいい。しかしSQLは3値論理を採用している。すなわち、TRUE、FALSE、UNKNOWN である。UNKNOWNという値は、比較述語や他の述語の戻り値として生じるものだが、れっきとした真理値であり、データがないことの目印であるNULLとは異なる。これが、SQLにおいて(x=NULL)ではなく(x IS [NOT] NULL)と書かねばならない理由である。NULLは値でも変数でもないので、値や変数にしか適用できない「=」という述語は適用できないのだ。表13.1に、SQL の3値論理における論理演算のマトリクスを示す。

表13.1　SQLの3値論理演算

x	NOT x
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

AND	TRUE	UNKNOWN	FALSE
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	UNKNOWN	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE

OR	TRUE	UNKNOWN	FALSE
TRUE	TRUE	TRUE	TRUE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	UNKNOWN	FALSE

既知の値同士の比較の結果は、TRUEかFALSEにしかならない。これはブール論理そのものであり、SQLの演算子においても変わりない。

だが、SQL は真理値UNKNOWNを発明――正確には再発見――した。当時、数学においてはすでに多値論理についての研究は多くの蓄積があった。離散的な値に基づく多値論理もあれば、連続的な値に基づく多値論理（たとえばファジー論理）なども考えられている。SQLの3値論理は、一見すると、ウカシェヴィチの発展させた体系に近い。彼の体系は、AND、OR、NOT の演算はSQLの3値論理とほぼ同じだ。

条件法は、普通の表記では矢印で表す。その2値論理における真理表は表13.2のとおりだ（表側がa、表頭がbの値である）。

表13.2**条件法（伝統的な2値論理）**

a → b	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	TRUE	TRUE

　これは伝統的に、「前提aが真の場合、bが偽であることはありえない」と読む【訳注4】。SQLの3 値論理では、以下のようになる。

表13.3**条件法（3値論理）**

a→b	TRUE	UNKNOWN	FALSE
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	TRUE	TRUE

　これは論理式で、¬(a∧¬b)と表すことができる。意味は、上の日本語の文と同じだ。ちなみに(¬a∨b)とも同値である。

　SQLのすべての述語は、`AND`、`OR`、`NOT`の3つの基本的な演算を組み合わせることで表現できる。だが、その挙動はすべてのケースにおいて直観的に明らかだとは限らない。特に、サブクエリにおいて述語が使用される場合は混乱しがちだ。その代表的なケースについて、以下で見てみよう。

13.5.1　サブクエリの述語におけるNULL

よく忘れがちなのが、サブクエリの中でもNULLの比較が頻繁に生じていることだ。たとえば、次の2つのテーブルを考えよう。

```
CREATE TABLE Table1 (col1 INTEGER);
INSERT INTO Table1 (col1) VALUES (1), (2);

CREATE TABLE Table2 (col1 INTEGER);
INSERT INTO Table2 (col1) VALUES (1), (2), (3), (4), (5);
```

列はすべてNULLを許可している。まず、次のクエリを実行する。

```
SELECT col1
  FROM Table2
 WHERE col1 NOT IN (SELECT col1 FROM Table1);
```

結果

```
col1
======
3
4
5
```

この結果に不思議な点はない。では次に、Table1にNULLを登録して、先ほどと同じクエリを実行する。

```
INSERT INTO Table1 (col1) VALUES (NULL);

SELECT col1
  FROM Table2
 WHERE col1 NOT IN (SELECT col1 FROM Table1);
```

今度は、結果は空になる。これは直観に反するが、しかし正しい動作である。あらゆる実装でこの結果が返る。このようになる理由は、NOT INの計算過程をステップごとに追ってみるとわかる。

```
SELECT col1
  FROM Table2
 WHERE NOT (col1 IN (SELECT col1 FROM Table1));
```

IN述語は次のようにANY述語に置換できる。

```
SELECT col1
  FROM Table2
 WHERE NOT (col1 = ANY (SELECT col1 FROM Table1));
```

ANY述語はORと同値である。

```
SELECT col1
  FROM Table2
 WHERE NOT (   (col1 = 1)
            OR (col1 = 2)
            OR (col1 = NULL));
```

最後の(col1 = NULL)は常にUNKNOWNであるから（NULLの比較は常にUNKNOWN！）、ド・モルガンの法則を適用すると以下のようになる。

```
SELECT col1
  FROM Table2
 WHERE ((col1 <> 1)
   AND (col1 <> 2)
   AND UNKNOWN);
```

さて、真理表を見てもらいたい。UNKNOWN がAND 演算に含まれていた場合、結果は必ずUNKNOWNまたはFALSEである。そして、UNKNOWNとFALSEはWHERE句の検索条件において必ず棄却され、選択結果は空になる。いかがだろう。NULLがいかに直観に反する奇妙な振る舞いを引き起こすか、その一端が理解いただけただろうか。

13.5.2　論理値述語

標準SQLは、新しい述語を追加することで、3値論理の問題を部分的に解決した。

```
<search condition> IS [NOT] TRUE | FALSE | UNKNOWN
```

この述語は、戻り値としてTRUEかFALSEしか返さないので、2 値論理の世界へ戻ることができる。たとえば、

```
((credit_score < 750) OR (eye_color = 'Blue')) IS NOT FALSE
```

という条件は、credit_scoreまたはeye_colorの少なくとも一方がNULLであれば、TRUEになる。たとえば、両方がNULLだった場合は、次のようになる。

```
((NULL < 750) OR (NULL = 'Blue')) IS NOT FALSE
((UNKNOWN) OR (UNKNOWN)) IS NOT FALSE
(UNKNOWN) IS NOT FALSE
TRUE
```

この述語は、PostgreSQLなどで実装されているが、まだ使える実装は多くない。

13.6

数学とNULL

これまでに、四則演算の中にNULL が現れると、伝播が起こることを学んできた。NULL の伝播は非常に強力なので、NULL/0でさえNULLを返す（詳細は第10章を参照）。

13.7

関数とNULL

すべての標準的な算術関数はNULLの伝播を発生させる。つまり、NULL が入力に含まれていると、結果がNULLになる。ほとんどのベンダーは、独自に拡張した関数においてもNULL 伝播のルールを守っている。たとえば、NULL のコサインはNULL である。NULLを入力にとって出力が非NULLになる関数は2つだけであり、いずれも変換関数だ。

1.NULLIF (V1, V2)は最初の引数が2 つ目の引数と等しければ、NULLを返す。この関数は次のようなCASE式と同値だ。

```
CASE WHEN (V1 = V2)
     THEN NULL
     ELSE V1 END
```

この式からわかるとおり、たとえば、V1 が何らかの値で、V2 がNULL だった場合、この関数の戻り値はV1になる。

2.COALESCE (V1, V2, V3, ..., Vn)は、引数のリストを左から右へスキャンし、最初のNULLでない値を返す。もしすべての引数がNULLの場合のみ、NULLを返す。これも、引数にNULLが含まれていても非NULLの戻り値を返す関数だ。

13.8

NULLとホスト言語

本書では、特定のホスト言語にSQL 文を埋め込んで使用する方法については取り上げない。そういう技術は適当なほかの書籍を当たってほしい。だが、NULL がホストプログラムに渡される際、どのような扱いを受けるかについては知っておく必要がある。埋め込みSQLが定義されていて、NULLを扱えるホスト言語であっても、NULLの扱い方については標準的な取り決めがない。これは、データベースにおいてNULLの使用を避けるべき理由の1つでもある。

大雑把に言って、ホスト言語ではSQL文はEXEC SQLで囲まれ、言語依存の終端子（セミコロンなど）を持っている。この混合言語プログラムは、SQLをホスト言語がコンパイルできるプロシージャコールに変換するSQLプリプロセッサを通じて実行される。その後、ホストプログラムが普通のやり方でコンパイルされる。

SQL の部分を識別するために、EXEC SQL BEGIN DECLARE SECTIONとEXEC SQL END DECL ARE SECTIONのペアが存在する。ホスト変数はデータベースからカーソル経由でデータを受け取る。これはホスト言語とデータベースが情報を受け渡すのに使う「中立領域」である。SQLは、自分がホスト変数を使っていることを知っている。SQLは、:varのようにコロンをプレフィックスに使う変数がSQL文に現れたら、それはホスト変数の印だと解釈しているからだ。SQLのクエリにおけるカーソルは、シーケンシャルファイルに似た構造を作る。カーソル内部のレコードは、FETCH 文を持った BEGIN DECLARE セクションの中で、一度に1 行ずつホスト言語に戻される。これは、レコード単位で処理するホスト言語と、集合指

向的なSQLの間のインピーダンスミスマッチを避けるための処置である。

NULLは、INDICATOR変数をホスト言語のBEGIN DECLAREセクションで宣言することによって処理する。INDICATORは厳密な数値型で、スケールはゼロである（つまり、ホスト言語では何らかの整数型として扱われる）。FETCH 文はカーソルから1 行取り出し、SQL のデータ型をホスト言語のデータ型に変換し、結果を適切なホスト変数に格納する。もしSQL 側の値がNULLだったら、INDICATORには負数がセットされる。INDICATOR が使われていない場合は、例外が発生する。おわかりのように、ホストプログラムは必ずINDICATOR の値をチェックしなければならない。そうしないと知らずにNULL が渡された場合に、パラメータの値がゴミになる危険がある。もしパラメータが何の問題もなくホスト言語に受け渡されれば、INDICATORは0にセットされる。ホスト言語に渡された値が非NULL の文字列で、それがINDICATOR を持っていれば、SQL 側の文字列長がセットされる。この情報は、文字列のオーバーフローをチェックするために使うことができる。

ODBC、JDBCといったほかのSQLインターフェイスも、カーソルは使わないが、ホストプログラムにNULLの情報を伝えるための似たメカニズムを持っている。

13.9

設計上のNULL の扱いに関するアドバイス

NULLの扱いに頭を悩ませるエンジニアにとって、ISNULL関数の使い方を知っておくことは重要だ。

——2009年12月20日、Microsoft SQL Server Programming Newsgroupにあるプログラマが投稿した文面

まず、このプログラマは、SQL Server でCOALESCE 関数が使えることを知らなかった。そのため、彼は実装依存のISNULL関数の構文を使って、ひどい方言まみれのSQLを書いていたし、コードにも間違いがあった【訳注5】。残念なことに、こういう例は枚挙に暇がない。優秀なSQLプログラマならば、NULLに対処するにあたり、まずはDDLにおいてこれを排除することに全力を傾けるだろう。すべてのテーブルのすべての列に原則としてNOT NULL制約を付与し、正当な理由がある場合にだけNULLの使用を許可する。間違っても、こういう努力をしないまま、（ニュースグループに投稿したプログラマのように）クエリで実装依存の変換関数に頼ればいいなどと考えないでほしい。NULLは、SQLを知らない人々を混乱させるのだ。

加えて、NULLを使うのはコストの高い行為でもある。というのも、通常、NULLはそれが現れる列の行に余計なビットフラグを持つことで実装されている。これは前述のとおり、NULL が列自身の値ではないためだ。これによって、ストレージ要件や、インデックスおよび検索条件において本来なら必要なかったことを考慮する必要がある【訳注6】。

NULL は主キーに含まれる列では許されない。これは、主キー（あるいは主キーの一部）がNULLであると何が起きるかを考えてみればわかる。キーにNULLが存在するということは、データモデルにおいて、テーブルにおけるエンティティが一意かどうかを判断できないということである。これはすなわち、RDBMSは、すでにテーブルに存在する主キーが重複しているかどうかを判断できないということになる。

また、外部キーにおけるNULLも同様に避けるべきである。SQLはこの点については、「疑わしきは罰せず」の原理を採用しており、外部キーにおけるNULL は禁止されていない。しかし、結合を使用するクエリにおいて情報の欠落を生じさせる原因になる。たとえば、在庫テーブルにおけるある部品コードが、受注テーブルの外部キーから参照されている場合、受注テーブルの部品コードがNULLの場合、その部品を在庫テーブルから結合によって選択することはできない。これは必ず対応が成立しなければならない関係の例だ。存在しない部品を注文することはできない。

だが、社員と仕事、および関連エンティティの仕事割り当てテーブルの間に成り立つ関係を定義する外部キーにおいてはそうとも言い切れない。新しく雇用された社員は、社員テーブルにおいてはその人についてすべての情報が登録される。一方、仕事テーブルのほうにも、まだ人が割り当てられていない仕事がたくさんある。この時点においては、仕事割り当てテーブルにおける新入社員の仕事はNULLとするのが適当だと思うかもしれない（「未知」のNULL）。

他方、情報の完全性が求められるエンコーディングスキームにおいては、NULLは許されるべきではない。たとえば、社員は人間であるから、社員は男か女かどちらかである。反対に、法的な人格（自然人、法人、その他の法的主体）の性別を記録する場合においては、ISO標準の性別コードに従って、0＝不明、1＝男性、2＝女性、9＝法人といったコード体系が必要になる。

実務の現場では、不明や適用不能のNULLの代わりに、（エンコーディングスキーマが数字だけから構成されているなら）すべて0 やすべて9 のコードを使うこともよく行われている。この慣習は、失われた値を空白のまま残しておいて、あとで判明したら穴を空けていたパンチカード時代の名残りだ（FORTRANは数値フィールドにおける空白を0と解釈する。同様に、すべて9のフィールドはファイルの終わりと解釈する）。

NULLを使わざるを得ないケースもある。たとえば、デフォルト値が意味を持たない日付列のようなケースだ。誰かの誕生日がわからない場合、適当なデフォルト値というのはあるだろうか？ 保険の失効日が不定の場合はどうだろう。この場合は、NULLに「永久」の意味を持たせるのが適切のように思われる。あるいは、イベント間の相対的な順序だけはわかっている場合も、一般にこれをデータベースで表現するのは難しい。たとえば、入社してからある程度の期間経過後に昇給が発生するルールがあった場合、具体的な昇給日は入社時点ではわからない。死刑囚の情報を持つテーブルでは、刑期の終了日は、刑の執行か自然死によってもたらされる。どちらの日付もあらかじめわかっているわけではないので、NULLを使うとも

に、現在の状態と推移を管理するための列を追加するといった工夫も必要になるだろう。

MySQLは日付型を拡張した機能を持っていて、もし年はわかっているが月日が不明の場合、'1949-00-00'のような値を登録できる。また、年月はわかっているが日が不明の場合は、'1949-09-00'のように登録できる。このタイプの値は、正確な日付の計算には向かないが、不完全な形であっても値を持っておくことが役に立つこともある。たとえば、人の誕生日をソートしたり、（だいたいの）年齢を計算するぐらいなら、おおよそ正しい結果が得られるだろう。

人名については、汎用的なNULL よりも、何らかのダミー値を使うほうがよい。特に、'John Doe #1'、'John Doe #2'といった具合に名前が不明の人であっても一人一人を区別する必要がある場合は、NULLでは対応できない。一方、量を表す列の場合は、NULLを使わざるを得ないケースがある。不明とゼロは違うからだ。所有している車のガソリンタンクが空なのと、車を持っていないのとは同じではない。なお、不明の値を表すのに負数を使うのは――たとえその列が正数しか意味を持たない属性を表していたとしても――やめたほうがいい。計算を混乱させるだけである。

ホスト言語が日付型を持っていなかった時代、「永久」や「最大期間」は'9999-12-31'という文字列で表現していた。実際これは、ISO 8601 で最大の日付と定義されている値だ。DATE 型を備えた4GL製品が市場に出回るようになり、プログラマはようやく「永久」を表すための最大値をデータベースに登録できるようになっ

た。だがいったんデータベースに登録されたデータは再び引き出され、ホスト言語における計算や統計的な集計に現れる。一番良い方法は、日付とフラグの2列を用意することだ。だがこれは、4GLではかなり複雑なコードになる。たとえば、正確な誕生日が不明の人物について、「±3年の範囲内」で誕生日が推定されたとすれば、その推定範囲を示すためのフラグが必要になるだろうし、こうした推定を含む母集団の平均年齢を計算するのには、特別な統計の計算を行わなければならないだろう。

13.9.1　ホスト言語においてもNULL を避けるべし

ホスト言語のプログラムからデータベースにデータを登録する場合も、なるべくNULLの登録は避けるべきである。そのために、以下のような原則を守ること。

原則1

ホスト言語において初期化を行うこと。すべてのデータ要素を初期化し、データベースに登録する前にクライアント側の画面に値を表示させること。このとき、デフォルト値を使って初期化することも必要だろうが、どのようなデフォルト値を使うかは別の問題である。

原則2

自動的な初期値。データベースにおいて列の初期値を設定しておけば確実にNULLを避けられる。

原則3

値を推測する。わかっている値から失われた値を導き出す。たとえば、妊娠経験のある患者は女性以外にあり得ない。前立腺癌の患者は男性である。このテクニックはユーザーに適切な値を選ぶよう選択肢を制限する場合にも使える。13.3 節で紹介した「識別されたNULL」のテクニックも有効だ。

原則4

失われたデータを追跡する。データに「不明」、「エラー」、「失効」などといった条件を付与することで、データがどのような理由によって失われているかを識別する。このようなタグをデータに付けるには、それ専用のコード体系を持った追加の列が必要になる。多くの商用のアプリケーションでは、ここまで行う必要はないが、データ品質を監査するような場合は、こうした詳細情報を使うのが便利である。

原則5

失われたデータがプログラミングと帳票作成に与える影響の大きさを判断する。NULLを含んだ数値列は集約関数を使うクエリにおいてミスリーディングな結果をもたらす。集約関数は計算の前にNULLを除外するからだ。そのため、これによって引き起こされる結果の相違を修正するには、プログラマはSQLSTATE=01003（集合関数内でNULL 値は除外されたことを示す）を検知しなければならない（これは警告なので、ロールバックは引き起こさない）。SQLSTATE状態コードについては、第7章も参照。

原則6

データが失われるのを防ぐ。基本的なことで、いまさら言うのもなんだが、データベースに登録する前にデータをざっと見直して適正な値に更新しておくこと。2000 年代前半、システム業界ではデータ品質に対する関心が急激に高まった。これは、監査を怠ったCEOが有罪判決を受けて刑務所にぶち込まれる事件が相次いだためだ。従来ニッチな分野だったデータ品質ツールの市場がにわかに活気づいたのも、同じ理由による。

原則7

データ型とNOT NULL 制約は、（広い意味での）データベース全体で整合的である必要がある。たとえば、会計の表はデスクトップ上のスプレッドシートと基幹レベルのデータベースの間で同じ方法で定義されていなければならない。

13.10

NULL の複数の意味を使い分ける方法

1996 年7 月、コンピュサーブ上の議論においてカール・C・フェダールはデータベースで複数の失われた値のトークンを扱うための面白いアイデアを思いついた。

もし読者がプログラムに埋め込んだSQLを使っているなら、INDICATOR列を使う必要に迫られたことがあるだろう。この列は主にデータベースにおけるSQL側の列がNULLかNULLでないかの情報をホスト言語に渡すために使う。ホスト言語はこの情報をプログラマに渡す。それでは、この概念をもう少し拡張して、テーブルに同じ機能の列を作ってはどうだろう。簡単な例を見て考えよう。

```
CREATE TABLE Bob
(keycol INTEGER NOT NULL PRIMARY KEY,
 valcol INTEGER NOT NULL,
 multi_indicator INTEGER NOT NULL
     CHECK (multi_indicator IN (0,      -- 既知の値
                                1,      -- 適用不能
                                2,      -- 失われた値
                                3)));   -- 近似値
```

このテーブルに次のようなルールを設ける。

- すべての列がわかっている場合→通常どおり合計値を求める。
- 値が適用不能の場合→合計は適用不能である。
- 適用不能な値はないが、失われた値がある場合→合計値は失われた値になる。
- 適用不能な値も失われた値もない場合→合計値は近似値であることを警告する。

クエリの一般的な形は次のようになる。

```
SELECT SUM (valcol),
       (CASE WHEN NOT EXISTS (SELECT multi_indicator
                                FROM Bob
                               WHERE multi_indicator > 0)
             THEN 0
             WHEN EXISTS (SELECT *
                            FROM Bob
                           WHERE multi_indicator = 1)
             THEN 1
             WHEN EXISTS (SELECT *
                            FROM Bob
                           WHERE multi_indicator = 2)
             THEN 2
             WHEN EXISTS (SELECT *
                            FROM Bob
                           WHERE multi_indicator = 3)
             THEN 3
             ELSE NULL END) AS totals_multi_indicator
  FROM Bob;
```

なぜvalcol列の合計値をこんなにいじり回すのか？ それは、合計値のステータスがmulti_indicator列でわかるからである。これはちょうど、オリジナルのテーブルでvalcol列のステータスがmulti_indicator列でわかったのと同じである。それでは、読者のための練習問題を出してみよう。

Q 1 複数の意味を持つ失われた値のためのルールを作り、SUM()、AVG()、MAX()、MIN()、COUNT()関数のそれぞれのクエリを書きなさい。

Q 2 multi_indicator列に近似の程度（±5、±10とか）を設定しなさい。valcolは常に中間にあるとする。また、状況に応じたあいまいさに対応するmulti_indicator 列を定義しなさい。

```
CREATE TABLE MultiNull
(groupcol INTEGER NOT NULL,
 keycol INTEGER NOT NULL,
 valcol INTEGER NOT NULL CHECK (valcol >= 0),
     valcol_null INTEGER NOT NULL DEFAULT 0
        CHECK(valcol_null IN
         (0,   -- 既知の値
          1,   -- 適用不能
          2,   -- 失われているが適用可能
          3,   -- 1%以内の近似
          4,   -- 5%以内の近似
          5,   -- 25%以内の近似
          6)), -- 25%以上の近似
     PRIMARY KEY (groupcol, keycol),
     CHECK (valcol = 0 OR valcol_null NOT IN (1,2)));
```

MultiNull：複数のNULLテーブル　groupcol：グループ列　keycol：キー列
valcol：値列

```
CREATE VIEW Group_MultiNull
(groupcol, valcol_sum, valcol_avg, valcol_max, valcol_min,
row_cnt, notnull_cnt, na_cnt, missing_cnt, approximate_cnt,
appr_1_cnt, approx_5_cnt, approx_25_cnt, approx_big_cnt)
AS
SELECT groupcol, SUM(valcol), AVG(valcol), MAX(valcol),
       MIN(valcol), COUNT(*),
       SUM (CASE WHEN valcol_null = 0 THEN 1 ELSE 0 END) AS notnull_cnt,
       SUM (CASE WHEN valcol_null = 1 THEN 1 ELSE 0 END) AS na_cnt,
       SUM (CASE WHEN valcol_null = 2 THEN 1 ELSE 0 END) AS missing_cnt,
       SUM (CASE WHEN valcol_null IN (3, 4, 5, 6) THEN 1 ELSE 0 END)
                                                         AS approximate_cnt,
       SUM (CASE WHEN valcol_null = 3 THEN 1 ELSE 0 END) AS appr_1_cnt,
       SUM (CASE WHEN valcol_null = 4 THEN 1 ELSE 0 END) AS approx_5_cnt,
       SUM (CASE WHEN valcol_null = 5 THEN 1 ELSE 0 END) AS approx_25_cnt,
       SUM (CASE WHEN valcol_null = 6 THEN 1 ELSE 0 END) AS approx_big_cnt
  FROM MultiNull
 GROUP BY groupcol;

SELECT groupcol, valcol_sum, valcol_avg, valcol_max, valcol_min,
      (CASE WHEN row_cnt = notnull_cnt
            THEN 'All are known'
            ELSE 'Not all are known' END) AS warning_message,
       row_cnt, notnull_cnt, na_cnt, missing_cnt,
       approximate_cnt,
       appr_1_cnt, approx_5_cnt, approx_25_cnt, approx_big_cnt
  FROM Group_MultiNull;
```

これは通常のアプリケーションで利用するには少し複雑すぎる方法だが、たとえばデータウェアハウスにデータを受け渡す「ステージング領域」でデータクレンジングを行う場合などには、有用だろう。

【訳注1】 ROLLUPやCUBEによって作られる合計行のことを超集合行と呼ぶ。超集合行の集約キーの箇所はNULLになる。またこれ以外にも、外部結合を行った際にキーが不一致だった場合にもNULLが生成される。これも「本物のNULL」とは意味が異なる。詳しくは第30章参照。

【訳注2】 日本語に訳してしまうとこのパラドックスの要点が伝わらないので、原文のまま示した。蛇足を承知で説明すると、「no cat」という語が、ヨーロッパ言語の構文上、あたかも何らかの実体――12尾の尻尾を持った化け猫のような――を指すかのように錯覚する点が、このパラドックスの面白いところである。日本語は、このような発想に結び付く構文を持っていない。

【訳注3】 GROUP BY 句の集約キーにNULLが含まれていた場合、それらのNULLはすべてまとめて「NULL」という1つのカテゴリとして、あたかも他の集約キーと同じ1つの値であるかのように扱われる（著者は「擬似同値性」と呼んでいる）。これは、いわばすべてのNULLは互いに等しいと言っているのと同じなので、比較述語においてNULL同士は等しいとも等しくないとも判断できないという3 値論理のルールに反しているように見える。だ

が実際問題、GROUP BYのキーにNULLが存在した場合、すべてのNULLを異なるとみなして異なるカテゴリとしてグループ化するのは現実的でないし、それもまた3値論理のルールに適合していないことに変わりはない。

【訳注4】 あるいは「aが真ならばbも真である」と読んでもよい。注目すべきは、前提aが偽の場合は、bは真でも偽でもかまわないことである。「もし新たな顧客を開拓したら給料倍増だ」と課長が部下に約束したとしよう。このとき、部下がめでたく新規顧客の開拓に成功すれば、もちろん課長は給料アップを実現しなければならない。だが、開拓に失敗したときは、課長は罰として減給しても、そのまま据え置いても（あるいは失敗したにもかかわらず給料をアップしても）、嘘をついたことにはならないということになる。

【訳注5】 ISNULLは、NULLを指定の値に変換するSQL Server独自の関数（OracleのNVLみたいなもの）。

【訳注6】 本文でも述べられているように、NULLは値ではなく、物理的な実装方法においても列の値として保持されていないことが多い。そのため、インデックスの中にもNULLを保持していない実装が多く、そのような実装ではIS NULL条件においてインデックスを使用することができない（DB2など一部のDBMSではNULLをインデッ

クスにも保持する仕様になっているが、あくまでベンダーの独自仕様のため一般性はない）。

14

複数列のデータ要素

Multiple Column Data Elements

　データ要素が原子的またはスカラ的でなければならないという概念は、データ要素がテーブルにおいて単一の列で表現されるという意味だと一般に理解されている。だがこれは必ずしも正しくない。データ要素が原子的であるということは、それ以上独立した、有意味な部分に分割できないということだ。もし原子的な属性を分割してしまえば、非常に深刻な設計上の誤りを犯すことになる。

　(x, y)の座標系を考えてみよう。xやyの値は、単独では、ある直線上の点を特定するのに対し、xとyのペアは平面上における単一の位置を与える。このとき、xとyを1つの列に入れてしまうのは不便だと思うだろう。そこで、2つの列に分けて管理する。だが、だからといって(x, y)が原子的なデータ要素ではないということにはならない。2 列に分けて管理するのは表記上の問題であって、抽象的なデータ要素のレベルでは、座標は――これ以上分割すれば意味が失われるという点で――原子的である。

14.1

距離関数

近年、地理データはその重要性を増している。緯度（latitude）と経度（longitude）を使って位置を特定し、地球上の2 点間の距離を計算できたら便利だと考える人は多いだろう。この計算を行う関数は、多くのSQL 製品ではまだ標準的ではないが、そのロジックを知っておくと何かと役立つ。この計算ではラジアンを使うが、これは、半径と弧の長さが等しくなる角度で、度数法で測ると約57.29578度（＝180/π）に相当する。つまり、10進で表された度数をラジアンに変換するには、度数に0.017453293（＝π/180）を掛け合わせればよい。ここでは円周率として3.14159265358979を用いるが、もっと精度の高い円周率を使えるなら、もちろんそのほうがよい。

いま、地球上に2 つの座標点が存在し、それを(Latitude1, Longitude1)、(Latitude2, Longi tude2)という座標で表すとしよう。この2 点間の距離をラジアンと三角関数で求める（読者の使うDBMSで三角関数が使えることを前提にしているが、最近では多くのDBMSで実装されている）。

```
CREATE FUNCTION Distance
(latitude1 IN FLOAT, longitude1 IN FLOAT,
 latitude2 IN FLOAT, longitude2 IN FLOAT)
RETURN FLOAT
IS
    r   FLOAT;
    lat FLOAT;
    lon FLOAT;
    a   FLOAT;
    c   FLOAT;
BEGIN
    --地球半径をkmからマイルに変換
    r := 6367.00 * 0.6214;

    --緯度・経度の差分を計算
    lon := longitude2 - longitude1;
    lat := latitude2 - latitude1;

    --中間値
    a := SIN(lat / 2) + COS(latitude1) * COS(latitude2) * SIN(lon / 2);

    --中間結果cはラジアンにおける大圏距離 [訳注1]
    c := 2 * ASIN(LEAST(1.00, SQRT(a)));

    --距離を求めるため、ラジアンに半径を乗じる
    RETURN (r * c);
END;
```

LEAST関数は与えられた2つの数の小さいほうを返すもので、丸めエラーを防止するためだ。これがないと、2点がほとんど地球の反対側であった場合、ASIN関数で逆正弦を求める計算が正しく行われない。LEAST 関数はOracle やMySQL の独自拡張だが、標準SQL でもCASE 式を使えば書ける（第29章参照）。スコット・コールマンは標準的な球面幾何学を使った2点の距離計算では、2点が非常に近い（約16 km以内）場合は不正確になることに気づいた。その理由は、非常に小さい角度に対するSIN関数の結果がゼロに近づくからだ。

この欠点を補うには、半正矢関数を用いる方法が有効だ【訳注2】。これは近距離の計算を非常に正確なものにするが（http://en.wikipedia.org/wiki/Haversine_formula）、今度はこのやり方だと地球の反対側の2点の距離を計算するときに、誤差が大きくなる（約16 km）という欠点がある。

```
CREATE FUNCTION HaversineDistance
(rlat1 IN FLOAT, rlon1 IN FLOAT, rlat2 IN FLOAT, rlon2 IN FLOAT)
RETURN FLOAT
IS
    --入力パラメータはラジアンで受け取る
    c FLOAT;
BEGIN
    c := POWER(SIN((rlat2 - rlat1) / 2.0), 2) +
           COS(rlat1) * COS(rlat2) * POWER (SIN((rlon2 - rlon1) / 2.0), 2);
    RETURN (3956.088331329 * (2.0 * ATAN2(SQRT(c), SQRT(1.0 - c))));
END;
```

ATAN2(x, y)関数は、デカルト座標(x, y)から極座標(r,θ)に変換するのに使っている【訳注3】。

14.2

IPv4 アドレスの格納方法

「IPアドレス型」というデータ型があるわけではないが、IPアドレスは人や会社の一意な識別子として使われる。もしある会社のIPアドレスが知りたければ、メールを送って担当部署に尋ねるか、`ping`を飛ばしてみればいい。IPアドレスを格納する方法は、大きく分けて3つある。文字列、整数、4つの8ビット数の集合である。

私がSQL Serverで実測してみたところ、どのやり方でも必要な時間、作業、データを文字列として返すのに必要なI/O量は大差なかった。整数と8ビット数を使う方法は、余計な計算が必要にはなるが、近年のマシンパワーを考えれば大したオーバーヘッドではない。

結論としては、4 つの`TINYINT` 列を使った8ビットモデルが、以下の3 つの利点を持っているという点で有利だ。

- プログラミングが簡単
- セクション別にインデックスを付与できる
- 人間が読みやすい

だが、`TINYINT`はSQL Server独自の1バイトの整数型であるため、他の環境でどのようなやり方がよいか考える必要がある。標準SQLで`TINYINT`に最も近いデータ型は、`SMALLINT`である【訳注4】。

以下、それぞれの方法を詳しく見ていこう。

14.2.1　単純なVARCHAR(15)列で格納する方法

IPアドレスを格納する最も単純な方法は、`'63.246.173.210'`のような形式で、`VARCHAR(15)`の列に格納することだ。この場合は、`CHECK`制約で`SIMILAR TO`述語を使い、「ドットと数字」が正しい位置に現れることを保証しなければならない。また、8ビットの数値における前ゼロをどう扱うかを決めなければならない。文字列比較を正確に行うためには、前ゼロは切り捨てるのがよいだろう。

この方法の利点は、プログラミングがかなりシンプルで、コードも読みやすいことだ。欠点は、ストレージの容量が大きくなることと、検索時にパターンマッチングの文字列関数を使う必要があることだ。また、IPアドレスは8ビットの数値で表現されていることを前提に作られているホストプログラムにデータを受け渡す際も面倒になる。文字列から8ビット数への変換は、カンマで区切られた文字列を個別の整数へ変換する文字列処理が必要だ。

14.2.2　1 つの整数列で格納する方法

これは3つの方法の中でストレージの容量は最小限に抑えられる。また付与するインデックスの大きさも最小限になり、検索コストも極めて低い。

欠点は、プログラミングが複雑になることだ。まず、8ビット数を整数に変換する関数が必要になる。かつ、結果として得られる整数値も人間には何がなんだかわからない。`2130706433`という整数を見せられて、これが`'127.0.0.1'`を意味するとすぐにわかる人はいないだろう。以下に、整数から8ビット数の文字列へ変換するユーザー定義関数を示す。

```
CREATE FUNCTION IPIntegerToString (in_ip IN INTEGER)
RETURN VARCHAR
IS
    o1 INTEGER;
    o2 INTEGER;
    o3 INTEGER;
    o4 INTEGER;
    ip INTEGER;
BEGIN
    ip := in_ip;

    IF ABS(ip) > 2147483647
        THEN RETURN '255.255.255.255';
    END IF;

    o1 := ip / 16777216;
    IF o1 = 0
    THEN o1 := 255;
        ip := ip + 16777216;
    ELSE IF o1 < 0
        THEN IF MOD(ip, 16777216) = 0
            THEN o1 := o1 + 256;
            ELSE o1 := o1 + 255;
                IF o1 = 128
                THEN ip := ip + 2147483648;
                ELSE ip := ip + (16777216 * (256 - o1));
                END IF;
            END IF;
        ELSE ip := ip - (16777216 * o1);
        END IF;
    END IF;

    ip := MOD(ip, 16777216);
    o2 := ip / 65536;
    ip := MOD(ip, 65536);
    o3 := ip / 256;
    ip := MOD(ip, 256);
    o4 := ip;

    --文字列を返す
    RETURN
    CAST(o1 AS VARCHAR) || '.' ||
    CAST(o2 AS VARCHAR) || '.' ||
    CAST(o3 AS VARCHAR) || '.' ||
    CAST(o4 AS VARCHAR);
END;
```

14.2.3　4 つのSMALLINT 列を使う方法

4つのSMALLINT列を使う方法の利点は、`VARCHAR(15)`よりはストレージの容量が少なくて済むこと、検索が簡単で比較的速いこと、4 つの8ビット数のそれぞれにインデックスを作成できることである。SQL Serverのように`TINYINT`（通常1バイト）のデータ型を持っていれば、ストレージコストはさらに節約できる。

欠点は、プログラミングが若干複雑になることだ。

```
CREATE TABLE FourColumnIP
(octet1 SMALLINT NOT NULL
     CHECK (octet1 BETWEEN 0 AND 255),
 octet2 SMALLINT NOT NULL
     CHECK (octet2 BETWEEN 0 AND 255),
 octet3 SMALLINT NOT NULL
     CHECK (octet3 BETWEEN 0 AND 255),
 octet4 SMALLINT NOT NULL
     CHECK (octet4 BETWEEN 0 AND 255),
 ...);
```

FourColumnIP：IPを4列で保持するテーブル　　octet1～octet4：8ビット列1～4

この方法を採用するなら、画面に表示する用途を考えてビューを作るのがいいだろう。このビューは見てのとおり、4つの列を1つにつなげるだけだ。

```
CREATE VIEW DisplayIP (IP_address_display)
AS
SELECT  CAST(octet1 AS VARCHAR(3))||'.'||
        CAST(octet2 AS VARCHAR(3))||'.'||
        CAST(octet3 AS VARCHAR(3))||'.'||
        CAST(octet4 AS VARCHAR(3))
  FROM FourColumnIP;
```

14.3

IPv6 アドレスの格納方法

最初にTCP/IPを設計した人々は、IPアドレスを32ビットで定義した。全世界のIPアドレスの割り当てはIANA（Internet Assigned Numbers Authority）が管理しており、5つの地域インターネットレジストリ（Regional Internet Registry：RIR）と協力して、各地域のインターネットサービスプロバイダなどの主体にIPアドレスのブロックを割り当てている。

IPv6アドレスは、IPv4よりもはるかに多くのキャパシティを持っており、当分は枯渇することはないと見られる。問題は、これがIPv4の単純な拡張ではなく、インターネットプロトコルの再設計であることだ。アドレスサイズが32ビットから128ビット（16 バイト）に拡張されたのはよいとして、それ以外にも、サブネットのルーティングプレフィックスを切り出すのが容易になり、効率的なルーティングのために大きなブロックを割り当てることができるなど、新しく設計された部分が多い【訳注5】。Vista 以降のWindows、Mac OS X 以降のMac、各種Linux ディストリビューションをはじめ、IPv6プロトコルに対するネイティブサポートを行っているOSは多い。

14.3.1　単一のCHAR(32)列

IPv6 アドレスを格納する最も明快な方法は、コロン抜きで`CHAR(32)`列を使うか、コロン込みで`CHAR(40)`列を使うことである。16進の表示フォーマットは、シンプルな固定長フォーマットである。`10(a)`から`15(f)`までの文字は、数字との混同を避けるため、通常小文字が使われる。「コロンと数字」が正しい位置に存在することを保証するために`CHECK`制約を使うのは、IPv4のときと同じだが、かなり複雑な正規表現になる。

文字列は固定長なので、一部だけを切り出すのは容易である【訳注6】。アドレスの一部だけを対象に検索するときのトリックは、`LIKE` 述語でワイルドカード（*）や正規表現の代わりに、アンダースコア（_）を使うことである。単純な`LIKE`述語は、対象文字列に対する有限オートマトンを生成することなく、文字と文字を比較するだけなので高速である。

14.4

通貨と他の単位との変換

通貨は貨幣の量と単位の2 つの意味を表現しなければならない。ISO 4217 の通貨コードは、各国または各地域の通貨単位を一意な3桁のコードで識別する。現在、通貨単位が10進でないのは、モーリタニアとマダガスカルだけである。両国では、主要通貨の価値が非常に低いため、補助通貨に10進の単位を採用するとあまりに小さくなりすぎて、実用に堪えないからだ【訳注7】。こうした通貨の小数位を計算する方法については、会計業務を担当する部署とよく相談する必要がある。

ユーロにおける通貨に関するルールはEUによって決められており、ドルに対するルールは米国会計基準（GAAP）によって決められている。

```
CREATE TABLE InternationalMoney
(...
 currency_code CHAR(3) NOT NULL,
 currency_amt DECIMAL (12,4) NOT NULL,
 ... );
```

InternationalMoney：国際通貨テーブル　currency_code：通貨コード　currency_amt：為替レート

この各国通貨が混在するテーブルは、そのままでは使うのが難しい。そのため、各地域のユーザーにはその地域だけの通貨を表示するためのビューを定義するのが望ましい。すると必然的に、そのビューでは為替レートを保持するための情報も必要になる。

```
CREATE VIEW EuroMoney (... euro_amt, ...)
AS
SELECT ... (M1.currency_amt * E1.conversion_factor), ...
  FROM InternationalMoney AS M1, ExchangeRate AS E1
 WHERE E1.to_currency_code = 'EUR'
   AND E1.from_currency_code = M1.curency_code;
```

ユーロが関係する換算を行う際は、1つ注意すべきポイントがある。ユーロから他の通貨、他の通貨からユーロへ換算する際の精度と丸めについては、厳密なルールがあるのだ。これは「三角換算」と呼ばれている【訳注8】。まず通貨1をユーロに換算し、その次にユーロを通貨2に換算しなければならない。換算の際は6 桁

の換算レートを使う必要がある。ただし、本書執筆後に規定が変更されている可能性もあるので、現時点で最新の情報を調べてほしい。

14.5

社会保障番号

米国において個人を一意に識別する番号に一番近いのは、社会保障番号（SSN）である【訳注9】。新たな社員を雇う際には、そのSSN が正しいものかどうかをチェックする必要がある。だが、多くのプログラマはこれをどうやってやればいいか知らない。といっても、ここでプライバシーや近年の立法の動向や法律的問題を論じるつもりはない。本書は現場のプログラマのためのものであり、我々が気にしなければならないのは、このSSNをコードの中でどう利用するかということだけだ。

SSNは3 つのパートから構成されている。すべて数字であり、XXX-XX-XXXX のようにダッシュで分けられている。それぞれのパートは、地域、グループ、通番と呼ばれている。多くの場合、地域は個人がSSN を申請した場所（1972 年以前）か、申請時に住んでいた場所（1972 年以後）を意味する。表14.1（次ページ参照）は米国で使われている地域コードの一覧である。

同じ地域コードが2回以上現れる場合があるが、これはそのコードが1つの州から別の州へ移行したか、その地域が分割されたことを意味している。ある場所がどの地域コードに該当するかを調べるには、郵便番号を使う。この表を見て、かなり人口の偏りがあると思うかもしれないが、いまさらどうしようもない。なお、1996 年以降、米国で就労するだけなら、必ずしもSSN は必要なくなった。アメリカ合衆国内国歳入庁は、SSNを持たない外国人労働者のために800 万個以上の納税者識別番号を発行している。2004年だけで、90万個の番号が発行された。

SSNのグループパートは、番号が割り当てられているか否か以外の意味は特にない。カード保持者のエスニシティがグループ番号にコード化されているというのは、ただの都市伝説であり、なぜそんな風説が広まったのか理解できない【訳注10】。社会保障局は、月に一度、各地域に最も多く割り当てられているグループのリストを公表している。

SSNが正当かチェックする唯一の方法は、グループ番号を調べることだ。最初の番号は1から9までの奇数で、その次に10 から98 までの偶数が続く。グループ98までの番号がすべて発行されたあとは、グループ2から8が発行され、11から99までの奇数が続く。

たとえば、地域XXXに対する最大のグループが72 の場合、XXX-04-XXXXという形式の番号が不正であることがわかる。というのも、9以下の偶数のグループがまだ割り当てられていなかったからだ。50～60年前、財布の中に偽のSSNカードが入っているのは決して珍しいことではなかった。というより端的に言って、多くの人が偽造カードを使っていた。番号は一見すると妥当に見えたが、当局は偽造カードの一覧を持っていて、摘発に当たっていた。

SSNの通番パートは何の意味も持っていない。通番はただ、0001から9999までの連番だが、必ずしも小さい数から割り当てられているわけではない。0000は決して使われない。

故人／存命を問わずSSN の妥当性チェックを行ってくれる営利企業や非営利のWeb サイトがある。こうしたサービスを利用すると、その番号に該当する人物が生きているのか死んでいるのか、発行後何年経つか、どこで発行したかといった情報を教えてくれる。こうしたサイトのいくつかは、政府機関や大学が雇用者、病院などのSSNチェックを支援するために運営している。商用サイトはファイルを提出すれば、大量のSSNを一括チェックしてくれる。コストは1つのSSN当たりおよそ1セントである。

以下にいくつかサイトを挙げる。もっとも、私から特定のサイトを推薦することはしない。

http://www.veris-ssn.com
http://www.searchbug.com/peoplefinder/ssn.aspx
http://info.dhhs.state.nc.us/olm/manuals/dma/eis/man/Eis1103.htm

表14.1　米国で使われている地域コードと所有者

000	不適切なコード	362-386	ミシガン	525, 585	ニューメキシコ
001-003	ニューハンプシャー	387-399	ウィスコンシン	648-649	

004-007	メイン	400-407	ケンタッキー	526-527	アリゾナ
008-009	ヴァーモント	408-415	テネシー	600-601	
010-034	マサチューセッツ	756-763		764-765	
035-039	ロードアイランド	416-424	アラバマ	528-529	ユタ
040-049	コネティカット	425-428	ミシシッピ	646-647	
050-134	ニューヨーク	587-588		530	ネヴァダ
135-158	ニュージャージー	752-755	割り当てられているが、まだ発行されていない	680	
159-211	ペンシルヴァニア	429-432	アーカンソー	531-539	ワシントン
212-220	メリーランド	676-679		540-544	オレゴン
221-222	デラウェア	433-439	ルイジアナ	545-573	カリフォルニア
223-231	ヴァージニア	659-665		602-626	
691-699		440-448	オクラホマ	574	アラスカ

232-236	ウェストヴァージニア	449-467	テキサス	575-576	ハワイ
232	ノースカロライナ	627-645		750-751	
237-246		468-477	ミネソタ	577-579	コロンビア地区
681-690		478-485	アイオワ	580	ヴァージン島
247-251	サウスカロライナ	486-500	ミズーリ	580-584	プエルトリコ
654-658		501-502	ノースダコタ	596-599	
252-260	ジョージア	503-504	サウスダコタ	586	グアム
667-675		505-508	ネブラスカ	586	アメリカンサモア
261-267	フロリダ	509-515	カンザス	586	フィリピン諸島
589-595		516-517	モンタナ	666	永久欠番
766-772		518-519	アイダホ	700-728	鉄道委員会 – 1963年に廃止
268-302	オハイオ	520	ワイオア	729-733	入国審査中［※1］

303-317	インディアナ	521-524	コロラド	734-899	予備
318-361	イリノイ	650-653		900-999	不適切なコード［※2］

※1　700番台において、728より大きな地域コードを持つSSNは存在しない（例外は729から733および764から772）。

※2　900–999は正当な地域コードではないが、プログラムでは使うことがある。州が高齢者、盲人や身体障害者を支援する際、社会保障局による連邦プログラムに変換される。また、この範囲の地域コードは、SSNが学生番号として使われていた時代、外国人学生に使われた。

14.6

有理数

有理数は、aとbを整数としたとき、(a/b)という分数で定義される数である。反対に、無理数は分数で表現できない数である。無理数の古典的な例は、2の平方根である。技術的には、2進法のコンピュータは有理数の部分集合しか表現することはできない。ある場合には、有理数を(分子, 分母)のペアとして表現するのが便利なこともある。たとえば、ヴァディム・トロパシュコは、階層構造をSQLで扱う入れ子区間モデルにおいて、有理数を利用している（以下の拙著を参照：『Trees and Hierarchies in SQL for Smarties』 Morgan-Kaufmann、2004）。

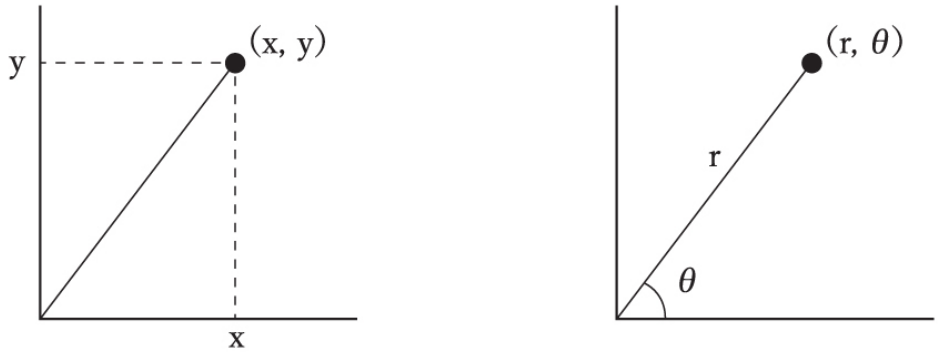
小学校の生徒なら、「1/2と1/4 を足すとどうなる？」と訊かれて、分子と分母をそれぞれ足して1/2＋1/4＝（1＋1）/（2＋4）＝ 2/ 6 ＝1/ 3 という計算をするかもしれない。これは和の計算としては間違いなのだが、この演算で得られる数は中間数といって、2つの分数の間の最も単純な数を返す（ここでの「単純」というのは、分母が小さいという意味だ）。実際、たとえば1/4と1/2の平均を既約分数で表すと、3/8であるが、この場合の分母は8で、中間数の3より大きい【訳注11】。

【訳注1】　大圏距離とは、地球の表面上の2 点を結んだ曲線の距離のこと。我々は普段、メルカトル図法で地球を平面化した地図を見慣れているが、実際は地球は球面なので、2点を結ぶ線は直線ではなく弧を描く。そのため、距離の算出にも三角関数を使う必要がある。

【訳注2】　半正矢関数（haversine）は、$\sin^2\frac{\theta}{2}=\frac{1}{2}(1-\cos\theta)$で定義される関数。

【訳注3】　我々が馴染み深い座標系(x, y)は、ある平面上の点を表すのに原点からの直交的な2 つの線分の長さxとy で表している。これは直交座標系とかデカルト座標系と呼ばれる。だが、ある点の位置を表す方法はほかにもあり、その1つが、ここで触れられている極座標系である。これは、原点からの角度（θ）と距離（r）で位置を表す。その表現のしかたから、円上の位置を表すのに便利なのはもちろん、これを3 次元に拡張すると空間上の位置を表現するのにも相性が良いた

め、地理情報を扱う際に利用される。



デカルト座標　　極座標

【訳注4】　ほかのDBMSでは、類似のデータ型として以下のものがある。だが普通は、この程度のストレージ消費を気にする必要はないだろう。

- MySQL：`TINYINT`（1バイト）
- PostgreSQL：`SMALLINT`（2バイト）
- DB2：`SMALLINT`（2バイト）

【訳注5】　IPv6のアドレスは、前半部（プレフィックスまたはネットワークID）と後半部（インターフェイス識別子）から構成される。第10章も参照。

【訳注6】　IPv6の表記ルールでは0を省略できるケースがあるが、フォーマットが複雑になるので、ここでは省略しないことを前提に考えてほしい。

【訳注7】　たとえばモーリタニアの通貨はウギアだが、これに対する補助通貨はコウムで、1ウギア=5コウムであり、10 の倍数ではない。また、マダガスカルの通貨はマダガスカルアリアリだが、1アリアリ＝5イライムビランゼと、こちらも10の倍数になっていない。

【訳注8】　SQLではないが、Microsoft Officeには、三角換算を行う`EuroConvert`という関数が存在する。

【訳注9】　社会保障番号（Social Security Number）は米国で市民や永住者に発行される番号で、個人を一意に特定できる。元来は徴税用だったが、最近は他の用途にも利用され、汎用的な国民識別番号になってい

る。日本におけるこうした個人識別番号としては、2013 年導入が検討されている国民総背番号制の「マイナンバー」が近い。

【訳注10】　エスニシティというのは、メキシコ系とかアイルランド系など、個人の出自を表す概念である。

【訳注11】　入れ子区間モデルは、木構造のような階層データをリレーショナルデータベースとSQLで扱うためのモデルである（基本的なアイデアは、第36章で解説する入れ子集合モデルと同じだが、ノードの座標を整数から有理数に拡張している）。このモデルの特徴は、2 つの有理数の間には、その区間がどれほど微小であっても、なおそこから無限の有理数の点を取り出すことができるという性質（稠密性）を利用して、ノードの更新処理を簡潔にしたことにある。中間数は、そのような2点の中間の点を取り出すアルゴリズムの1つである。入れ子区間モデルについては、拙著『達人に学ぶDB設計徹底指南書』でも解説しているので、興味ある方はそちらも参照してほしい。

15

テーブルの操作

Table Operations

リレーショナルデータベースのテーブルにおいて、行の集合に対してSQLが実行できる操作は4 つしかない。挿入、削除、更新、検索である。操作単位は常に基底テーブルの行集合全体である。ファイルシステムを使ってきた経験の長い人は、1回のアクセスで1つのレコードの1つの列を操作するという考え方に慣れている。繰り返しグループや可変長のレコード形式であれば、ファイル内の各レコードの構造を変更することさえ可能だったろう。

だがSQLにおける心理モデルは違う。操作単位は常に部分集合であり、挿入、更新、削除は一度に集合全体を扱う。ここで、テーブルにおける1つの行に1つのプロセッサを割り当てることができるぐらいのマシンパワーを持っていると想像してみてほしい。もしそうなら、読者が開始の笛を吹いた瞬間に、すべてのプロセッサがパラレルで一斉に処理を開始するのだ【訳注1】。

15.1

DELETE FROM 文

SQLにおけるDELETE FROM文は、テーブルから0 行以上の行を削除する。インタラクティブなSQLツールを使っていれば、ユーザーが何行を削除したかを知らせてくれる。標準SQLでも、もし1行も削除しなかった場合は、SQLエンジンが「該当データなし」の完了条件を通知することを要求している。SQL では、DELETE FROM 文には位置付け型と探索型の2 つの形式がある。位置付け型削除は、カーソルを使って行う。探索型削除は、SELECT文のように、WHERE句で検索条件を指定して使う。

15.1.1 DELETE FROM 句

探索型削除の構文は次のとおりである。

```
<delete statement: searched> ::=
DELETE FROM <target table> [[AS] <correlation name>]
[WHERE <search condition>]
```

DELETE FROM 句は単純に、更新可能なテーブルまたはビューの名前を指定する。SQL:2003 以降、DELETE FROM 句で相関名を指定することも可能になった点に注意してほしい。それ以前は、DELETE FROM 句で相関名を指定することはできなかったのだが、これは、SQL エンジンがテーブルの別名を扱うロジックに由来する制限だった。SQLエンジンは、テーブルの別名が与えられると、まずは、基底テーブルまたは更新可能なビューから、それと同じ行を持つ新しい一時テーブル、つまりコピーを作る。そして、そのコピーに対して操作を行った後、最後にコピーを削除する。この動作のため、

DELETE FROM 句で相関名を指定すると、結局削除対象になるのは、システムが生成した一時テーブルのほうで、元のテーブルは何ら変更されないということになってしまうのだ。

だが、SQL:2003ではこの制限が撤廃され、相関名が指定されたときは、一時テーブルの代わりに更新可能なビューを作るようになった。このおかげで、ビューに対する変更は元のテーブルにも反映されるようになったのだが、この新しい機能は使わないことを強く推奨する。

位置付け型削除では、基底テーブルから削除される行はカーソルのカレント行に限られる。構文は次のとおりである。

```
<delete statement: positioned> ::=
DELETE FROM <target table> [[AS] <correlation name>]
WHERE CURRENT OF <cursor name>
```

SQLにおけるカーソルは、一般に非手続き型のコードに比べ高コストで、標準の機能ではあるものの、実際は実装によって構文もバラバラだ。もしテーブルのキーが適切に設計されていれば、このタイプのDELETE FROM文を使う必要はないはずだ。

15.1.2 WHERE 句

WHERE句に関して最も重要なことは、これがオプションであって、必ずしも付けなくてもよいことだ。WHERE 句がなければ、テーブルのすべての行が削除される。テーブル構造そのものは存在し続けるが、行はすべてなくなる。

すべてではないが、多くのインタラクティブなSQL ツールは、WHERE 句なしのDELETE FROM文を実行しようとすると、親切にも

本当にやってよいか確認してくる。もし本当にテーブルを空っぽにするつもりでなければ、即座にロールバックしてテーブルを復元させることだ。もしコミットするか、あるいはツールの設定をオートコミットにしていたら、データを復元させることは難しい。DBAが何とかデータを復旧させるよう頑張ってくれるだろう……【訳注2】。もっとも、少なくともSQLを学習している間は、こういうミスをやってしまっても気に病む必要はない。行の集合全体を一度に削除するということは、1レコードずつ確認を取りながら削除することはできないということだ。もう一度言うが、SQLは行の集合全体を一度に処理する。

多くの実装において、テーブルに対する削除は2つの段階を踏んで行われる。

第1段階

WHERE句の条件に合致する行に削除フラグを立てる。この際に、削除することで制約違反が起きないかどうかもチェックされることが多い。一番ありがちな制約違反は、子テーブルから外部キーで参照されている親キーを削除してしまうことだ（「おい、オレたちはまだ『ピンク・フラミンゴ』の注文を受け付けてるんだ。勝手に在庫から削除するなよ」）。もちろん、表明、CHECK 制約、トリガーなどで実装された制約に引っかかった場合にも、ロールバックが発生する。

第2段階

もし対象の行集合を削除しても問題ないことがわかったら、実際に削除する。このときも、データを本当にきれいさっぱり消

してしまうケースと、とりあえずユーザーから非表示の状態にして、あとでクリーンアップ処理が物理的に削除するケースがある。その後、インデックスの再編成などの後処理が行われる。ここで重要なポイントは、行に削除フラグが立った状態でも、テーブル全体がまだ検索対象としては有効なことだ。

多くの場合において、この2 段階の削除方式は、第1段階でいきなり消してしまう方式と違いがない。おおかたのDELETE FROM文のWHERE句の条件は比較的単純で、定数を使うか、同一行内の列を使うぐらいだからだ。たとえば、人事テーブルから社員を削除する（つまりクビにする）ケースを考えよう。

```
DELETE FROM Personnel
 WHERE iq <= 100; -- 定数を使用した単純な条件
```

または、

```
DELETE FROM Personnel
 WHERE hat_size = iq; -- 同じ行の列を使用する
```

こうした単純な条件であれば、オプティマイザは検索条件がテーブル全体に影響しないことを理解し、一度しかスキャンしない。だが検索条件で自己参照する場合、2 段階を踏んで削除する方法は、1段階だけのときと異なる結果になりうる。たとえば、部門の平均よりもIQの低い社員をクビにするDELETE FROM文を考えよう。

```
DELETE FROM Personnel
WHERE iq < (SELECT AVG(P1.iq)
              FROM Personnel AS P1 -- ここは相関名を付けなければならない
             WHERE Personnel.dept_nbr = P1.dept_nbr);
```

当初のデータは次のような状態だとする。

```
CREATE TABLE Personnel
(emp_nbr   CHAR(16) PRIMARY KEY,
 dept_nbr  CHAR(16),
 iq        INTEGER);
```

Personnel：社員テーブル　emp_nbr：社員ID　dept_nbr：部署ID　iq：IQ

Personnel

emp_nbr	dept_nbr	iq
Able	Acct	101
Baker	Acct	105
Charles	Acct	106
Henry	Mkt	101
Celko	Mkt	170
Popkin	HR	120
…		

もし仮に、この処理を1 行ずつ実行したと仮定しよう。まず会計部門（'Acct'）の平均IQは次のように計算できる。

(101＋105＋106) / 3＝104

すると、クビになるのはAble だ。次に、テーブルをシーケンシャルにスキャンし、もう一度平均IQを求める。すると今度はこういう結果になる。

(105 + 106) / 2 = 105.5

今度クビになるのはBakerだ。残るのはCharlesだけである。

今度は、テーブルをアルファベットの降順で並べてみよう。今度はCharlesが先頭に来る。IQ の平均値は104だが、Charlesは合格なので残留。次にテーブルをシーケンシャルにスキャンすると、IQ の平均値は104 のままなので、Bakerも残る。だが、やはり哀れなAble はクビの運命を免れない。

こういうシーケンシャルな動作をするDBMSはいまはまだ存在しないが、初期のDB2 では、行の削除をストレージに格納されている物理的な順序に従ってシーケンシャルに実行していたことは、覚えておく価値があるだろう。また、SybaseのSQL Anywhere（昔のWatcom SQL）は、テーブルをソートするORDER BY 句をオプションで持っていた。これは、テーブルに対してまさにシーケンシャル削除を行うためのものだ。この機能は、順序が問題にならない場合にシーケンシャル削除を強制する場合に使われた。それによって、テーブルへの2 段階目の操作をスキップすることで高速化を図れたのだ。シーケンシャル削除はまた、カーソルとホスト言語を使ったときと同じ結果を得るためという目的でも使われた。

私は、アンダース・アルトバーグとヨハネス・ベッカーとともに、重複行を削除するDELETE FROM文で、どのようなロジックが最も高速かテストしたことがある。dup_cnt列はdup_id内の連番を意味する。テスト対象の3つのSQL文は以下のとおりだ。

```
CREATE TABLE Test
(dup_id   CHAR(3),
 dup_cnt  INTEGER);
```

Test：テストテーブル　dup_id：重複ID　dup_cnt：連番

Test

dup_id	dup_cnt
00A	1
00A	2
00A	3
00B	1
00C	1
00C	2

D1

```
DELETE FROM Test
WHERE EXISTS (SELECT *
                FROM Test AS T1
               WHERE T1.dup_id =  Test.dup_id
                 AND T1.dup_cnt < Test.dup_cnt);
```

D2

```
DELETE FROM Test
 WHERE dup_cnt > (SELECT MIN(T1.dup_cnt)
                    FROM Test AS T1
                   WHERE T1.dup_id = Test.dup_id);
```

D3

```
CREATE TABLE WorkingTable
(dup_id   CHAR(3),
 min_dup_cnt  INTEGER);

INSERT INTO WorkingTable(dup_id, min_dup_cnt)
SELECT dup_id, MIN(dup_cnt)
  FROM Test
 GROUP BY dup_id;

DELETE FROM Test
 WHERE dup_cnt > (SELECT min_dup_cnt
                    FROM WorkingTable AS Working
                   WHERE Working.dup_id = Test.dup_id);
```

それぞれの動作を簡単に説明しよう。D1は、相関サブクエリ内の条件によって、同じdup_id 内で、カレント行のdup_cnt よりも小さなdup_cnt が存在する行を削除する。これはつまり、次のようにdup_cntが最小の行だけを残すのと同義だ。

dup_id	dup_cnt
00A	1
00B	1
00C	1

D2 は、相関サブクエリ内のMIN 関数で、同じdup_id 内で最小のdup_cntを求め、それより大きいdup_cntの行を削除する。したがって結果はD1と同じになる。D3も、相関サブクエリでやっていたことを一時テーブルを使っているだけで、結果はやはり同じだ。

デスクトップマシンにインストールしたDBでの、それぞれの実行結果は以下のとおりだ。

- D1：3.20秒
- D2：31.22秒
- D3：0.17秒

この結果は、データ量、DBMS、マシンスペックによって異なるが、それぞれの実行計画は、だいたい以下のようになるだろう。

D1 は、EXISTS 述語内のdup_id の条件においてインデックスを使用できるだろう。D2 はdup_idのグループごとにスキャンを繰り返し、最小値を計算する必要があるため、コストが高くなりがちだ。D3は、dup_id 列を使った単純な結合で、当然、インデックスも使用できる点が有利だ。なお、D3は導出テーブルを使えば次のように1つの文で書き換えることも可能だ。

D3-2

```
DELETE FROM Test
 WHERE dup_cnt >
         (SELECT min_dup_cnt
            FROM (SELECT dup_id, MIN(dup_cnt)
                    FROM Test
                   GROUP BY dup_id) AS WorkingTable(dup_id, min_dup_cnt)
           WHERE WorkingTable.dup_id = Test.dup_id);
```

もっとも、条件によってはカーソルを使ったフルスキャンが、重複行を削除するには高速な場合もある【訳注3】。

15.1.3　別のテーブルに基づいた削除

WHERE 句はやろうと思えばいくらでも複雑な条件を記述できる。サブクエリを使えば、他のテーブルを使うことも可能だ。たとえば、借金のブラックリストを管理するテーブルから、負債を返した債務者を削除すること考えよう。これは次のようにEXISTS述語を使った相関サブクエリで可能だ。

```
DELETE FROM Deadbeats
 WHERE EXISTS (SELECT *
                 FROM Payments AS P1
                WHERE Deadbeats.cust_nbr = P1.cust_nbr
                  AND P1.paid_amt >= Deadbeats.due_amt);
```

サブクエリ内のSELECT文では、相関名（P1）を付けなくてもスコープのルールからP1の列であることは自動的に判断できるが、読みやすさのためすべての列にテーブル名または相関名を付与するのがよい。

15.1.4　同一テーブルのデータを削除する

DELETE FROM 文では、削除対象のテーブルの列、定数、集約関数を使うことができる。例として、クラスから成績が平均を下回る生徒を削除する文を見てみよう。

```
DELETE FROM Students
 WHERE grade < (SELECT AVG(grade) FROM Students);
```

だが、FROM句に存在するテーブルの相関名を使うことはできない。そのため、SELECT句では問題なく動作するWHERE 句であっても、DELETE FROM 句ではエラーになる場合がある。たとえば、自己結合を記述することはできない。一例として、上司よりも給料の良い部下を削除するという問題を考えよう。社員テーブルには、社員を識別するemp_nbrと、上司の社員番号を保持するboss_nbrという列を持っている。このとき、次のSQL文は動かない。

```
CREATE TABLE Personnel
(emp_nbr  CHAR(3) PRIMARY KEY,
 boss_nbr CHAR(3),
 salary   INTEGER);
```

Personnel：社員テーブル　emp_nbr：社員ID　boss_nbr：上司ID　salary:給与

```
DELETE FROM Personnel AS B1 --このような相関名は許されない
 WHERE Personnel.boss_nbr = B1.emp_nbr
   AND Personnel.salary > B1.salary;
```

この問題を回避する手段はいくつかある。1 つのトリックは、このテーブルそのままのビューを作り、相関名の代わりにそのビューを使うことだ。

```
CREATE VIEW Bosses
AS
SELECT emp_nbr, salary
  FROM Personnel;

DELETE FROM Personnel
WHERE EXISTS (SELECT *
                FROM Bosses AS B1
               WHERE Personnel.boss_nbr = B1.emp_nbr
                 AND Personnel.salary > B1.salary);
```

単純にPersonnelテーブルをサブクエリの中で使うだけだと、うまくいかない。というのも、WHERE句において外側でもサブクエリ内のPersonnelテーブルに対する参照が必要だが、サブクエリ内のテーブルを外側で操作することはできないからだ【訳注4】。こうしたビューは、SQLエンジンが主記憶の中に実体化できるよう、なるべく小さいほうが望ましい。

テーブルから重複行を削除する

重複行とは、テーブルにおいて本来不要な行のコピーである。重複行が発生する最もありがちな理由は、本来キーである列に一意制約を付与せずに、同じデータを二重登録してしまうケースだ。重複行をテーブルからSQLで削除するのは、思ったよりも難しい。実際、もし重複行が完全に全列について同じ値を持っている場合は、単純なDELETE FROM文では削除できない。重複行を削除するには、その中の1行だけを救いつつ残りのすべての行を削除しなければならない。SQL でそれらの区別ができなければ、すべての行を削除してしまう。もう1 つの問題は、基底テーブルから行を削除すると、参照整合性制約のアクションが発動される可能性があることだ。その結果、望ましくない副作用が発生することもありうる。

たとえば、Table1の行が削除されると、それに対応するTable2の行も削除するという参照整合性制約が存在した場合、Table1の重複行を削除すると、Table2の行が削除されて、残ったTable1 の行とマッチする行がなくなってしまうかもしれない。だが、参照整合性制約では、Table2には少なくとも1行はTable1の残った行とマッチする行が存在しなければならないのである【訳注5】。こうした問題

は、遅延制約を使うことで回避できるかもしれない。たとえば、重複行を持ったMessedUpTableから重複行を排除する次のトランザクションは、遅延制約を使うことでトランザクション終了時まで制約の施行を遅らせられる。

```
BEGIN
INSERT INTO WorkingTable --重複排除のためDISTINCTを使う
SELECT DISTINCT * FROM MessedUpTable;

DELETE FROM MessedUpTable; --MessedUpTableテーブルをクリアする
INSERT INTO MessedUpTable  --MessedUpTableにワークテーブルのデータを入れる
SELECT * FROM WorkingTable;

DROP TABLE WorkingTable; --ワークテーブルを削除する
END;
```

ROWIDを使った重複行削除

レオナルド・C・メデルは、Oracle で重複行を削除する面白い方法を考えついた。次のようなテーブルを考えよう。

```
CREATE TABLE Personnel
(emp_id INTEGER NOT NULL,
 name CHAR(30) NOT NULL,
 ...);
```

Personnel：社員テーブル　　emp_id：社員ID　　name：氏名

Oracleにおける古典的な重複行削除のSQL文は次のようなものだ。

```
DELETE FROM Personnel
 WHERE ROWID < (SELECT MAX(P1.ROWID)
                  FROM Personnel P1
                 WHERE P1.emp_id = Personnel.emp_id
                   AND P1.name = Personnel.name);
```

ROWIDという列は――正確には擬似列だが――ストレージにおける行の物理的な格納位置に基づいた識別子である。ユーザーセッションが終了すると変わる可能性があるが、セッション中は不変である。これは、物理ディスクのデータが格納されているアドレスをダイレクトに指定するため、Oracleのテーブルに物理的にアクセスする手段としては最速である。データの物理的表現はユーザーから隠蔽されるべきであるというコッドのルールには真っ向から反する機能ではあるが。

パフォーマンスは、10 万行ほどのデータをテーブルに用意してみれば、簡単にテストできる。メデルの報告によれば、上記のROWIDによる方法と、次に紹介する2 つの改善案を試したところ、改善案のほうが10倍ほど高速だったという。

最初の改善案は、重複行のグループの中で、最も大きなROWIDを見つけ、それ以外の行を削除する方法だ。

```
DELETE FROM Personnel
 WHERE ROWID IN (SELECT P2.ROWID
                   FROM Personnel P2,
                       (SELECT P3.emp_id, P3.name,
                               MAX(P3.ROWID) AS max_rowid
                          FROM Personnel P3
                         GROUP BY P3.emp_id, P3.name) P4
                  WHERE P2.ROWID <> P4.max_rowid
                    AND P2.emp_id = P4.emp_id
                    AND P2.name = P4.name);
```

この場合、重複行は全列の値が重複しているという前提なので、GROUP BY句のキーにはテーブルの全列を指定する必要がある。

2 つ目の改善案は、テーブルの全行から、削除対象の行集合を引き算する方法である。これは次のようなSQL文になる。

```
DELETE FROM Personnel
 WHERE ROWID IN (SELECT P2.ROWID
                   FROM Personnel P2
                 MINUS
                 SELECT MAX(P3.ROWID)
                   FROM Personnel P3
                  GROUP BY P3.emp_id, P3.name);
```

MINUSは標準SQLのEXCEPTに相当するOracleの方言である。この2つの方法が最初に紹介した古典的な方法よりも速い理由は、WHERE句で相関サブクエリの使用を回避しているからだ。より新しいやり方としては、ROWIDの代わりとなる識別子をROW_NUMBER関数を使って作成する方法もある。これならば実装非依存である。

```
CREATE VIEW OrderedPersonnel on delete do instead
AS
SELECT emp_id, name,
       ROW_NUMBER() OVER (PARTITION BY emp_id, name) AS r
  FROM Personnel ;

DELETE FROM OrderedPersonnel
 WHERE r > 1;
```

なお、この方法は、OrderedPersonnelビューが更新可能であることを前提としている。

15.1.5　参照整合性制約を使わずに複数のテーブルから削除する

DELETE FROM文で複数のテーブルから行を削除する直接的な方法はない。複数のテーブルから関連付けられた行を削除する方法は3つある。

1. 削除する値を持った一時テーブルを使う。

2. 参照整合性制約のCASCADEオプションを使う
3. ビューまたは基底テーブルのINSTEAD OFトリガーを使う。

たとえば、注文テーブルと在庫テーブルを持ったデータベースを考えよう。実装したいビジネスルールは、在庫が切れた製品について注文を削除するというものだ。参照整合性制約はまったく宣言されていないと仮定する。これをSQLで実現するには、まずは、削除するべき製品を集めた一時テーブルを作り、次に、そのテーブルをサブクエリで使用する。

```
CREATE MODULE Foobar
CREATE LOCAL TEMPORARY TABLE Discontinue
(part_nbr INTEGER NOT NULL UNIQUE)
ON COMMIT DELETE ROWS;

...
PROCEDURE CleanInventory(...)
BEGIN

INSERT INTO Discontinue
SELECT DISTINCT part_nbr --削除する項目
  FROM ...
 WHERE ... ;  --何か適当な条件

DELETE FROM Orders
 WHERE part_nbr IN (SELECT part_nbr FROM Discontinue);

DELETE FROM Inventory
 WHERE part_nbr IN (SELECT part_nbr FROM Discontinue);

COMMIT WORK;

END;
...
END MODULE;
```

標準SQLでは、一時テーブルはスキーマにおける永続的なオブジェクトだが、中身のデータは一時的である。セッションの開始時には

必ず空で、セッションを保持しているユーザーにしかデータは見えない。GLOBAL オプションを付けると、どのようなモジュールからもテーブルの1つのコピーにアクセスできる。LOCALオプションの場合は、宣言されたモジュールだけにデータのスコープが制限される。

15.2

INSERT INTO 文

INSERT INTO文は基底テーブルにデータを追加する唯一の方法である。大量データをテーブルにロードする場合は、別の方法も用意されているが、それらはベンダーの独自拡張である。

15.2.1 INSERT INTO 句

INSERT INTO句の構文は次のとおりである。

```
<insert statement> ::= INSERT INTO <insertion target>
<insert columns and source>
<insertion target> ::= <table name>
```

<insertion target>に指定できるのはテーブル名だけであって、別名を使うことはできない点に注意してほしい。

```
<insert columns and source> ::=
<from subquery> | <from constructor> | <from default>
<from subquery> ::=
[<left paren> <insert column list> <right paren>]
[<override clause>] <query expression>
<from constructor> ::=
[<left paren> <insert column list> <right paren>]
[<override clause>]
<contextually typed table value constructor>
<override clause> ::=
OVERRIDING USER VALUE | OVERRIDING SYSTEM VALUE
<from default> ::= DEFAULT VALUES
<insert column list> ::= <column name list>
```

INSERT INTOの基本的な形式は、（通常は1行のみを保持する）テーブル定数か、クエリインサートである。テーブル定数を使うインサートでは、VALUES句を使ってデータを指定する。挿入する値の

リストには、通常は定数か明示的なNULLを指定するが、理論的にはスカラサブクエリを含む任意の式を記述することができる。

DEFAULT VALUES句は、VALUES (DEFAULT, DEFAULT, ..., DEFAULT)の省略形である。これは特定の1行を挿入するために使われる。

テーブル定数を使ったINSERTは、ちょっとした用途に使える。たいていは、少量のデータを投入するために、インタラクティブなセッションから使用される。

他方、クエリインサートは、まずクエリ部を実行し、その結果を一時テーブルに保持する。その後、一時テーブルを一度に全行ターゲットのテーブルに挿入する。どちらの場合も、ターゲットテーブルの列のリストは任意だが、クエリの列リストまたはVALUES 句とUNION 互換でなければならない。リストに含まれていない列の値は、NULL か明示的なデフォルト値が指定されていればデフォルト値が使用される。

<override clause>は、SQL:2006 で定義された。これはまだ広く利用できる機能ではないが、IDENTITY 列をINSERT がどう制御するかを指定するためのオプションだ。このオプションは見かけよりも複雑な問題を抱えているため、ここでは詳細に立ち入らない。

15.2.2 INSERT の性質

理論上、クエリを使ったINSERTは、ターゲットのテーブルにクエリの結果行を一度に投入する。この集合指向的な性質から言えるのは、たとえば、次のようなINSERT 文において、Some Tableへの

INSERTにどれだけの時間がかかろうとも、transaction_timeの値は1つしかないということだ。

```
INSERT INTO SomeTable (somekey, transaction_time)
SELECT millions, CURRENT_TIMESTAMP
  FROM HugeTable;
```

　これは非常にシンプルな仕様だと思うだろうが、裏では挿入される各行に対してさまざまなチェックが行われている。1 行でもターゲットテーブルの制約に違反すれば、INSERT 全体が失敗し、ロールバックされる。一般的な実装では、行をターゲットテーブルに挿入する一方で、ワークテーブルのほうは、トランザクション全体が妥当とみなされるまで、未コミット状態に保っておく。INSERTがコミットされると、ターゲットテーブルの全インデックスが再構築される。この操作は、他ユーザーからのアクセスをロックするほか、もしクラスタ化インデックスであれば、テーブルをソートすることもありうる【訳注6】。

　ファイルシステムを使った開発に慣れている人は、むしろ最初にインデックスを削除し、データを挿入し、その後でテーブルのソートとインデックス再作成を行う、という順序を考えるかもしれない。実際、ファイルシステムでは、システムの運用を支援するユーティリティプログラムは、ファイルのレコードがどのようなキーでソートされているかを把握することで、効率的にインデックスを作成するものもある。だが、残念ながらSQLではその考えはうまくいかない。インデックスは一意性や参照整合性を維持する役割も持っており、簡単にテーブルから削除したりソートし直したりできないか

らだ。ファイルは互いに独立の存在だが、テーブルはデータベース全体の一部なのだ。

15.2.3　バルクロードとデータ抽出ツール

どんなバージョンのSQLにおいても、外部ファイルを読み込んで直接テーブルにデータを投入するための言語拡張やユーティリティプログラムがある。こうしたツールに特に標準はないので、実装ごとにバラバラだ。だが大半のツールでは、ファイルの名前とそのフォーマットを指定する必要がある。こうしたツールは、単純なものだと、ただファイルを読み、1 つのターゲットテーブルに投入するだけだが、Oracle が持つような高度なものになると、レコードを読むたびに簡単な編集ができるミニ言語を備えている。もしシンプルなツールを使っているなら、ターゲットテーブルに投入する前に、データクレンジング用のワークテーブルを作っておくのが賢明だ。このワークテーブル上で、編集用のプロシージャを適用したり、重複行を検出したり、不正なデータを除外すればいい。

ツールの中には、こうしたワークテーブルがない場合、自動的に作るものもある。こうした機能は、当然のことだが、ワークテーブル定義のメタデータが必要になるが、通常は前回のバルクアンロード（またはエクスポート）結果としてこうしたデータを保持できる。

入力用のユーティリティがあれば、出力用のユーティリティもある。これは、反対にテーブルをファイルに変換する。たいていはオプションでフォーマットを選択できる。こちらもやはり計算や選択はSQL 内で完結し、ホスト言語を使う必要がない。こうしたプログ

ラムの中には、SELECT文やビューを入力できるものもあるが、単純に基底テーブルをファイル変換するだけの機能しか持たないものもある。また、最近では多くのツールが、CREATE TABLE やCREATE INDEXのDDLおよびデータ投入用のINSERT文を出力する機能を持っている。

15.3

UPDATE 文

SQL のUPDATE 文の機能は、テーブルの0 行以上の行の0 列以上の列の値を変更することである。SQLの実装は、何行が更新対象になったか、あるいは少なくとも更新対象行がなかった場合はそのことをSQLSTATEの値で返す。UPDATEにも、DELETE文と同じく、位置付け型と探索型の2 つの形式がある。位置付け型UPDATEはカーソルと一緒に使用し、探索型はWHERE 句を使う点も同じだ。カーソルは、CURRENT OF <cursor name>で指定された行だけを更新対象にする。カーソルについては7.3節を参照してほしい。

15.3.1 UPDATE 句

探索型UPDATEの構文は次のとおりである。

```
<update statement> :: =
UPDATE <table name> [[AS] <correlation name>]
SET <set clause list>
[WHERE <search condition>]
<set clause list> :: =
<set clause> [{, <set clause>}...]
<set clause> :: = <object column> = <update source>
<update source> :: = <value expression> | NULL | DEFAULT
<object column> :: = <column name>
```

UPDATE 句は単純に変更対象の基底テーブルか更新可能ビューの名前を指定する。元来、SQL ではUPDATE 句で相関名を使うことはできなかった。これは、SQL においてテーブルの別名が使われた場合、内部的にはエンジンが新しいテーブルを生成し、基底テーブルや更新可能ビューと同じデータを投入していたからだ。そのため、相関名のテーブルに更新を行うと、システム生成された一時テーブ

ルにだけ変更が適用され、オリジナルのテーブルはまったく変更されないようになっていたのだ。だがSQL:2003からこの仕様が変わり、UPDATE句で相関名を使っても、オリジナルのテーブルに更新が反映されることになった。これは、一時テーブルの代わりに更新可能ビューが作られるように内部仕様が変更されたためである。

SET 句は更新対象の列リスト、WHERE句は更新対象行を制限する。なお、UPDATE文が正常に動作するには、ユーザーが更新対象の列に対する更新権限を持っていることが前提になる。SQL:2003では、SET句で行構築子を使うことができる。構文はこうである。

```
UPDATE Foobar
   SET (a, b, c) = (1, 2, 3)
 WHERE x < 12;
```

これは、列を列挙した通常の構文の短縮形である【訳注7】。

```
UPDATE Foobar
   SET a = 1,
       b = 2,
       c = 3
 WHERE x < 12;
```

15.3.2 WHERE 句

すでに述べたように、WHERE句について最も覚えておくべきことは、これが任意だということだ。WHERE句がない場合、テーブルの全行が更新対象となる。必要なWHERE句を付け忘れるというのは、非常によくある間違いである。もしやってしまった場合は、即座にロールバックするか、DBAに助けを求めること。

<search condition>で真と判定された行は、個別にではなく1つの部分集合としてマークされる（空集合であることもある）。次に、この部分集合は、テーブルから削除され、新たな行が挿入される。たとえ更新対象が0 行であっても、それは妥当な更新とみなされることに注意してほしい。

15.3.3　SET 句

<set clause list>における各列への値の割り当ては、パラレルで実施される。そのため、すべての列が一挙に更新される。少なくとも理論的なモデル上はそうなっている。実装上は、まずすべての対象行がWHERE句を使ってマークされ、次にSQLエンジンはマークされた行を一時領域にコピーし、SET 句はその古い行イメージをもとに実行され、更新結果が新しい行イメージとして作られる。最後に、古い行イメージが削除され、新しい行が挿入される。もしこのプロセスのどこかでエラーが発生すれば、ロールバックが行われ、テーブルは未変更のまま、ユーザーにエラーが返される。このパラレルな更新プロセスは、伝統的な第3世代プログラミング言語における変数の更新プロセスとは異なるため、初学者がよくつまずくところだ。たとえば、次のような文は、SQLでは構文的に正しくない。

```
UPDATE MyTable
   SET a = b, a = c;
```

伝統的なプログラミング言語では、更新は左から右へシーケンシャルに実行されるので、まずa にb の値が代入され、次にa にc の値が代入されるので、結局a はcと一致する。しかし、SQL では同時に

a にb とc を代入するという意味になるため、エラーが発生する【訳注8】。<set clause list>の左辺には、1つの列が複数回登場してはならないのだ。繰り返すが、SET 句の更新は同時に行われるので、複数の代入値が指定されると、どちらが正解かわからなくなってしまうのだ。

このSQLの特性から、2つの列の値を交換するUPDATE文は、逆にとても簡単に書ける。次のようにすればよい。

```
UPDATE MyTable
SET a = b, b = a;
```

これは、次のプロシージャと同値ではない。

```
BEGIN
UPDATE MyTable
   SET a = b;

UPDATE MyTable
   SET b = a;
END;
```

最初に紹介した単独のUPDATE文では、列aとbの値が入れ替わる。2 つ目のプロシージャでは、列aの値がbと同じになり、bの値は変化しない。

15.3.4　別のテーブルを使った更新

実のところ、別テーブルの値を使って更新する方法としては、今日ではMERGE文を使うのが適している（この方法は次節で見る）。だが、UPDATE文でこれを行っている古いコードを見る機会もあるだろう。SET句の右辺では、定数や列だけでなく、相関サブクエリを使

うこともできる。UPDATE句のテーブル名のスコープはUPDATE文全体に及ぶため、SET 句やWHERE句内のサブクエリでも参照が可能なのだ。言葉で説明するより実例を見たほうがわかりやすいだろう。まず次のようなテーブルを作る。

```
CREATE TABLE Customers
(cust_nbr INTEGER NOT NULL PRIMARY KEY,
 acct_amt DECIMAL(8,2) NOT NULL);
```

Customers：顧客テーブル　cust_nbr：顧客番号　acct_amt：口座残高

```
CREATE TABLE Payments
(trans_nbr INTEGER NOT NULL PRIMARY KEY,
 cust_nbr INTEGER NOT NULL,
 trans_amt DECIMAL(8,2) NOT NULL);
```

Payments：支払いテーブル　trans_nbr：取引番号　cust_nbr：顧客番号
trans_amt：支払額

問題は、Customersテーブルの口座残高から、Paymentsテーブルの支払額の合計を引いて更新することだ。こうした更新は、たいていバッチ処理で一括して行われるので、探索型UPDATE文を使うのが定石だ。

```
UPDATE Customers
SET acct_amt = acct_amt - (SELECT SUM(trans_amt)
                             FROM Payments AS P1
                            WHERE Customers.cust_nbr = P1.cust_nbr)
WHERE EXISTS
      (SELECT *
         FROM Payments AS P2
        WHERE Customers.cust_nbr = P2.cust_nbr);
```

もし支払いがない場合、スカラサブクエリは空集合を返す。SUM関数は空集合を引数に取ると、常にNULLを出力するので、結局サブクエリはNULLになる。この場合、NULLの伝播が起きて、

```
acct_amt = acct_amt - NULL
acct_amt = NULL
```

という代入式になってしまい、acct_amt がNULL になってしまう。これを防止するために、Paymentsに存在しない顧客は更新対象から除外しているのが、WHERE句のEXISTSを使った条件である。

さて、SET句でサブクエリを使うときの最もよくある間違いが2つある。1つ目が、サブクエリが複数行を返すケースだ。もし顧客番号（cust_nbr）でPayments テーブルが一意になって、この問題を考える必要がないなら、次のようにSUM関数を除外することが可能だ。

```
UPDATE Customers
SET acct_amt = acct_amt
                - (SELECT trans_amt
                     FROM Payments AS P1
                    WHERE Customers.cust_nbr = P1.cust_nbr)
WHERE EXISTS
      (SELECT *
         FROM Payments AS P2
        WHERE Customers.cust_nbr = P2.cust_nbr);
```

だが、顧客が複数の支払いを実施していて、Paymentsテーブルに複数行が存在していれば、このUPDATE文は失敗し、ロールバックが発生する。最初のコードならば、SUM(trans_amt)によって、顧客

1人につき1行だけしか返らないことが保証されており、この問題を回避できる。

2 つ目のよくある間違いは、UPDATE 文で集約関数を使った場合、空集合を引数に取ると0 ではなくNULL が返る点だ。たとえば、顧客ごとの平均支払額で更新することを考えよう。先のUPDATEのSUM関数を単純にAVG関数に置き換えると、次のようになる。

```
UPDATE Customers
   SET acct_amt = (SELECT AVG(P1.trans_amt)
                     FROM Payments AS P1
                    WHERE Customers.cust_nbr = P1.cust_nbr)
 WHERE EXISTS (SELECT *
                 FROM Payments AS P1
                WHERE Customers.cust_nbr = P1.cust_nbr);
```

ここでも、WHERE 句を使ってacct_amt がNULL に更新されることを防いでいる。もう1 つの解として、サブクエリが空集合だった場合にCOALESCE関数を使う方法もある。

```
UPDATE T1
   SET c1 = COALESCE ((SELECT c1
                         FROM T2
                        WHERE T1.keycol = T2.keycol), T1.c1),
       c2 = COALESCE ((SELECT c2
                         FROM T2
                        WHERE T1.keycol = T2.keycol), T1.c2),
       ...
 WHERE ... ;
```

こうすることで、マッチングしなかった行は値を変更せずにおける（正確には、同じ値で更新するのだが）。だが、このやり方だと、SET 句の列の数だけT2に対するスキャンが発生し、実行コスト

が高い。ジェミー・リッカードは、COALESCE関数をサブクエリの中に入れ込むことでこの問題を解決した。

```
UPDATE T1
   SET (c1, c2, ...) = (SELECT COALESCE (T2.c1, T1.c1),
                               COALESCE (T2.c2, T1.c2),
                               ...
                          FROM T2
                         WHERE T1.keycol = T2.keycol)
WHERE ... ;
```

もちろん、この方法を使うには、実装が行構築子をサポートしている必要がある。

15.3.5 UPDATE 文でCASE 式を使う

CASE 式をUPDATE 文の中で使うのは非常に便利である。このトリックの重要なポイントは、SET a = aが何も結果を変えない、という点にある。前節のUPDATE文は次のように記述できる。

```
UPDATE Customers
   SET acct_amt = CASE WHEN EXISTS
                            (SELECT *
                               FROM Payments AS P1
                              WHERE Customers.cust_nbr = P1.cust_nbr)
                       THEN (SELECT AVG(P1.trans_amt)
                               FROM Payments AS P1
                              WHERE Customers.cust_nbr = P1.cust_nbr)
                       ELSE acct_amt END; -- 条件に合致しなければ更新しない
```

このUPDATEはWHERE句を持っていないため、テーブルをフルスキャンする。特定の日に支払いを行う顧客は全体の一部であると想定すると、フルスキャンは非効率かもしれないし、反対にフルスキャンが高速なケースも多くあるだろう。これは業務仕様に依存するので一概に言えない。

だがCASE 式の本当の強力さは、複数のUPDATE 文を1 つにまとめられることにある。これによって、実行時間を大幅に短縮できるうえに、プロシージャや汚いSQLコードを読みやすく簡潔に記述できるのだ。その典型が、第6章で取り上げた、本の値引き問題だ。再掲すると、25ドル以上の本は一律10%オフとし、25ドル未満の本は一律15%値上げする。単純に考えると次のように書きたくなる。

```
BEGIN  -- 間違い！
UPDATE Books
   SET price = price * 1.15
 WHERE price < 25.00;

UPDATE Books
   SET price = price * 0.9
 WHERE price >= 25.00;
END;
```

だがこれはうまくいかない。現在の価格が24ドルの本は、最初のUPDATE 文で27.6ドルに上昇し、2 番目のUPDATE 文で24.84ドルに下落する。これはやりたかったこととは違う。

UPDATE 文の順序を入れ替えてもうまくいかない。

かつては、この問題を解決する手段はカーソルだった。1 行ずつループして、IF-THEN-ELSE 文で値下げするか値上げするか決める。このときのテーブルは、あたかもファイルのようだ。しかし今日ではこんな方法を使う必要はない。我々には宣言型の道具であるCASE 式がある。

```
UPDATE Books
SET price = CASE
            WHEN price < 25.00 THEN price * 1.15
            ELSE price * 0.9
            END;
```

これは速いし、何より結果が正しい。

CASE式を使う際は、同一の列に対して特定の順序で関数なり計算を適用する必要があるかを注意して考える必要がある。UPDATE文のSET句はパラレルで実行されるため、もしそういう場合は、CASE式で一度に実行するのではなく、UPDATE文を分割する必要がある。たとえば、次のような複数のUPDATE文ならば、CASE式で置き換えても問題ない。

```
BEGIN
UPDATE Foobar
   SET a = x
 WHERE r = 1;
UPDATE Foobar
   SET b = y
 WHERE s = 2;
UPDATE Foobar
   SET c = z
 WHERE t = 3;
UPDATE Foobar
   SET c = z + 1
 WHERE t = 4;
END;
```

これは次の文と同値である。

```
UPDATE Foobar
   SET a = CASE WHEN r = 1 THEN x ELSE a END,
       b = CASE WHEN s = 2 THEN y ELSE b END,
       c = CASE WHEN t = 3 THEN z
                WHEN t = 4 THEN z + 1
                           ELSE c END
 WHERE r = 1
    OR s = 2
    OR t IN (3, 4);
```

WHERE句はなくても結果は同じだが、インデックスが有効に使えればパフォーマンスは改善できる。ここで注目すべきは、複数の

UPDATE文を1つにまとめることができたのは、SET句において、更新対象の列が一度しか代入されないからだ。その意味で、この方法は更新対象の列を基準に分類しており、ターゲット駆動と言える。対して、複数のUPDATE文で実行するほうのコードは、変更する条件を基準に分類しているため、ソース駆動とでも言えるだろう。

　もし読者がこの問題を、シーケンシャルファイルを使って手続き型言語で解く場合、どちらの考え方をするか考えてみてほしい。

15.4

ベンダー拡張が持つ欠陥について

基本的に本書では実装依存のSQLについては取り上げないが、SybaseとMicrosoftのT-SQL に含まれるUPDATE 文には、ユーザーに警告しておくべき重大な欠陥がある。「パワーユーザー」の中には、この欠陥のゆえにMERGE文を使うことを主張する人もいるぐらいだ。

その欠陥とは、UPDATE文でFROM 句にテーブルを指定できる独自構文だ。もし更新対象の基底テーブルがFROM 句にも現れると、その行は複数回操作されることになり、リレーショナルモデルの原理に完全に違反する。1 つの列に複数の値を代入しようとしたときの正しい振る舞いは、カーディナリティ違反のエラーを返すことである。次の簡単なサンプルを見てもらいたい。

```
CREATE TABLE T1 (x INTEGER NOT NULL);
INSERT INTO T1 VALUES (1), (2), (3), (4);
CREATE TABLE T2 (x INTEGER NOT NULL);
INSERT INTO T2 VALUES (1), (1), (1), (1);
```

ここで、T2とマッチするT1のすべての行を2 倍するUPDATE 文を作る。オリジナルのSybase バージョンのFROM句の使い方は、次のようなものだ。

```
UPDATE T1
   SET T1.x = 2 * T1.x
  FROM T2
 WHERE T1.x = T2.x;
```

T1

```
 x
====
 16
 2
 3
 4
```

この結果の意味がわかるだろうか？ T1とT2でxの値が一致するのは、x=1の行だけである。だが、T2にはx=1の行が4行あるので、T1でx=1のxは4回2倍されるのだ。結果、1×2×2×2 ×2 ＝16 である。この種の問題の一部は、Sybase の現行バージョンでは修正されているが、この構文はまだ使える（もちろん非標準である）。Microsoft は、単純に、複数の行から現在の物理的配置に基づいた値を1つ取り出すことで、このカーディナリティ問題は解消している。アダム・マチャニクの考えた簡単な例で見てみてよう。

```
CREATE TABLE Foo
(col_a CHAR(1) NOT NULL,
 col_b INTEGER NOT NULL);

INSERT INTO Foo VALUES ('A', 0), ('B', 0), ('C', 0);

CREATE TABLE Bar
(col_a CHAR(1) NOT NULL,
 col_b INTEGER NOT NULL);

INSERT INTO Bar
VALUES ('A', 1), ('A', 2), ('B', 1), ('C', 1);
```

次に、SQL Server の実装依存の構文を使って、次のようなFROM句付きのUPDATE 文を実行する。

```
UPDATE Foo
   SET Foo.col_b = Bar.col_b
  FROM Foo INNER JOIN Bar
    ON Foo.col_a = Bar.col_a;
```

このUPDATE文の結果は一意に定まらない。Barテーブルには、col_a='A'の行が('A', 1)、('A', 2)の2つ存在するため、どちらが使われるかは、（もしcol_bにクラスタ化インデックスが存在しなければ）行の物理的配置によって決まるからだ。

この問題を回避するには、アレックス・クズネツォフの考えたCOUNT関数を使うトリックが必要になる。

```
UPDATE Foo
   SET Foo.col_b = Bar.col_b
  FROM Foo, Bar
 WHERE Foo.col_a = Bar.col_a
   AND (SELECT COUNT(*)
          FROM Bar
         WHERE Foo.col_a = Bar.col_a) = 1;
```

こうすると、更新される値の候補が複数ある行は更新対象から除外される（それが業務要件を満たすのか、というのはまた別問題だが）。

15.5

MERGE 文

SQL-99 において、磁気テープのファイルシステム時代から使われていた「マージ＆インサート」のプロシージャを真似する構文がSQLにも追加された。これはUPDATEとINSERTを同時に行うということで"upsert"とも呼ばれる。基本的なビジネスロジックは次のとおりである。

```
EACH FOR トランザクションテーブル
    DO IF   現在行がマスタテーブルに存在しない
       THEN 現在行をマスタテーブルに挿入する;
       ELSE マスタテーブルをトランザクションテーブルの検索条件に合致する行の値で更新する
    END IF;
END FOR;
```

1950年代には、トランザクションテープとマスタテープを突合するために、ソートしておく必要があった。だが、SQL バージョンのMERGE文ではそのような処理を記述する必要はない。構文は以下のとおりである。

```
MERGE INTO <table name> [AS [<correlation name>]]
USING <table reference> ON <search condition>
{WHEN [NOT] MATCHED [AND <search condition>]
THEN <modification operation>} ...
[ELSE IGNORE];
```

UPDATE文で相関名を使うことができなかったように、MERGE文でも相関名は使えないのではないかと思う読者もいるかもしれない。だが、やはりSQL:2003でMERGE文でも相関名を使うことが可能になった。とはいえ、わざわざイレギュラーな構文で書く必要はないだろう。

行がターゲットテーブルとマッチした場合でもしなかった場合でも、WHEN 句ではさらに<search condition>を付けることができる。実装によっては、この<search condition>を付けられないものもあるので、注意する必要がある。こういう制限がある製品の場合は、ON 句やWHERE句にロジックを書く、あるいはCASE式を使うという方法で回避できる。

<modification operation>には、INSERT、UPDATE、DELETEを記述する。構文はそれぞれ単体で記述する場合と変わらない。これによって、複雑なプログラミングロジックを、1 つの文の中に隠すことができる。一方、NOT MATCHED の場合、ON 句の検索条件がFALSE またはUNKNOWNの行に対する操作を記述する。THENの後ろには、INSERTまたは例外を発生させる単一の文だけを記述できる。

いま、Personnelテーブルの給与列を、PersonnelChangesテーブルの列で更新することを考える。どちらのテーブルも主キーにemp_nbrを使っている。これは典型的なMERGE文の用例である。

```
CREATE TABLE Personnel
(emp_nbr  CHAR(6) PRIMARY KEY,
 emp_type CHAR(12),
 salary   DECIMAL(8,2),
 bonus    DECIMAL(8,2),
 comm     DECIMAL(8,2));
```

Personnel：社員テーブル　emp_nbr：社員ID　salary：給与　bonus：ボーナス
comm：手当て

```
CREATE TABLE PersonnelChanges
(emp_nbr CHAR(6) PRIMARY KEY,
 salary  DECIMAL(8,2),
 bonus   DECIMAL(8,2),
 comm    DECIMAL(8,2));
```

PersonnelChanges：給与変更テーブル　emp_nbr：社員ID　salary：給与
bonus：ボーナス　comm：手当て

```
MERGE INTO Personnel
USING (SELECT emp_nbr, salary, bonus, comm FROM PersonnelChanges) AS C
   ON (Personnel.emp_nbr = C.emp_nbr)
WHEN MATCHED
THEN UPDATE
        SET salary = C.salary,
            bonus  = C.bonus,
            comm   = C.comm
WHEN NOT MATCHED
THEN INSERT (Personnel.emp_nbr, Personnel.salary, Personnel.bonus, Personnel.comm)
     VALUES (C.emp_nbr, C.salary, C.bonus, C.comm);
```

もし役職によって給与体系が異なる場合は、CASE式の出番である。

```
MERGE INTO Personnel
USING (SELECT emp_nbr, salary, bonus, comm FROM PersonnelChanges) AS C
   ON (Personnel.emp_nbr = C.emp_nbr)
WHEN MATCHED
THEN UPDATE
        SET salary = C.salary,
            bonus  = CASE WHEN emp_type IN ('sales', 'executive')
                                THEN C.bonus
                                ELSE 0.00 END,
            comm   = CASE WHEN emp_type = 'sales'
                                THEN C.comm
                                ELSE 0.00 END
WHEN NOT MATCHED
THEN INSERT (Personnel.emp_nbr, Personnel.salary, Personnel.bonus, Personnel.comm)
     VALUES (C.emp_nbr, C.salary, C.bonus, C.comm);
```

MERGE 文に相当する実装依存の機能も存在するので、そうした機能については、使っているDBMS のマニュアルを調べてほしい。多くは、データウェアハウスにデータを追加するために使われることが多い。

MERGE文を最初に見た人は、次のようなUPDATE文とINSERT文の組み合わせと同じだと思う。

```
BEGIN
UPDATE T1
   SET (a, b, c, ...
        = (SELECT a, b, c, ...
             FROM T2
            WHERE T1.somekey = T2.somekey),
 WHERE EXISTS
        (SELECT *
           FROM T2
          WHERE T1.somekey = T2.somekey);

INSERT INTO T1
SELECT *
  FROM T2
 WHERE NOT EXISTS
        (SELECT *
           FROM T2
          WHERE T1.somekey = T2.somekey);
END;
```

だが両者は完全に同じではない。MERGE文は単一の文であるため、全体として最適化することが可能だが、UPDATE＆INSERTは2つの分離した文であるため、個別最適化しかできないし、書いた順番でしか実行されない。MERGE文で追加の検索条件付きのWHEN [NOT] MATCHED句を使えば、オプティマイザはパラレル、あるいは最新の統計に基づいた実行順序で実行できる。

【訳注1】　1行に1プロセッサというわけにはいかないが、現在でもOracleやSQL Serverのようにパラレル SQLをサポートする実装はある。こうした実装においては、複数の行が文字どおり同時並行的に処理されるため、CPU やストレージといったハードウェアリソースに余裕があれば、大きなパフォーマンス向上が期待できる。

【訳注2】　もし万が一、意図せぬデータを削除し、かつコミットしてしまった場合、復旧の手段はあるだろうか？ フルバックアップやテーブル単位のダンプファイルが取得されていれば、その時点まで戻すことができるし、トランザクションログの差分バックアップを有効にしている環境であれば、削除を実行する直前の状態まで復旧させることが可能だ。もっとも、リストアという大掛かりな作業が発生することは言うまでもない。

【訳注3】　フルスキャンが性能的に有利な条件は、一般に、テーブルの件数が少ない場合やストレージレベルでのデータ分散が十分に行われている場合である。こうしたケースでは、カーソルでフルスキャンする方法が相関サブクエリや結合でインデックスを使用する方法よりもパフォーマンスが良いことはありうる（あまり多くはないが）。

【訳注4】　実際には、この制限を持っておらず、ビューを作らなくても相関サブクエリ内で基底テーブルを参照できる実装も多い。

【訳注5】　多くの実装で、参照整合性制約における親キーとして指定できる列は、主キーまたは一意制約の存在する列 に限られるという条件が付いているのは、この問題を回避するためであろう。

【訳注6】　クラスタ化インデックスはSQL ServerやMySQL（InnoDB）に存在するインデックスの種類。テーブルに1 つだけ作成することが可能で、インデックスキーでテーブルの行をソートする。通常主キーのインデックスに使用される。

【訳注7】　この機能は、一見するとトリビアルに見えるかもしれないが、これが真価を発揮するのは、`SET` 句の右辺がサブクエリ、特に相関サブクエリの場合である。列を1列ずつ指定する従来の構文では、そのたびにサブクエリを記述せねばないが、行構築子を使えば、サブクエリの記述を1か所にまとめることが可能となり、可読性、実行コストともに有利である。なお、サブクエリと行構築子を組み合わせる構文は、Oracle、PostgreSQLなどがサポートしている。

【訳注8】　MySQLのようにエラーにならず、この構文を左から右へシーケンシャルに実行する実装もある。

16

比較演算子またはシータ演算子

Comparison or Theta Operators

コッドは初期の論文で「θ（シータ）演算子」という術語を導入した【訳注1】。これはプログラマが比較演算子と呼ぶものだ。SQLの大半のデータ型は、他のプログラミング言語のデータ型と比べると、比較を行うのが少し難しい。あるデータ型の値を他のデータ型の値と比較するには、必ず型変換が必要だが、SQLは強い型付け言語であるため、許されている型変換が少ない――クリスマスを数値に変換して平方根を求めることはできないのだ【訳注2】。

比較演算子をオーバーロードすることにより、数値、文字列、日付の3つのデータ型に適用することができる。比較演算子の記号と意味を表16.1に示す。古い実装では、「～と等しくない」という意味の!=や¬=という記号も使われていたが、これはC 言語やPL/I から借りたもので、いまでは標準SQLに入っていない。こうした古い実装依存の機能を使うとコードの移植性と可読性が下がるので、もし読者がまだ使っているようなら、ここで習慣を改めてほしい。比較演算子は述語でもあるので、戻り値にTRUE、FALSE、UNKNOWN の3 つの真理値を取る。TRUE とFALSEは通常どおりのルールに従うが、UNKNOWNが返されるのは、比較対象の少なくとも1つがNULLだった場合だ。ただし、比較的新しい演算子IS [NOT] DISTINCT FROM

には注意してほしい。これはNULLを引数にとってもTRUEとFALSEしか返さない（第13章参照）。

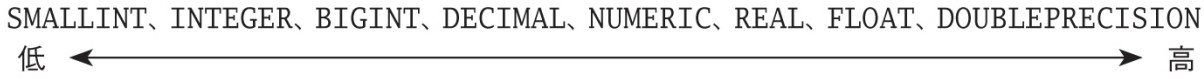
表16.1　比較演算子の記号と意味

演算子	数値	文字列	日付
<	～より小さい	～より照合順序が前	～以前
=	～と同じ	～と照合順序が同じ	～と同時
>	～より大きい	～より照合順序が後	～以後
<=	～以下	～より照合順序が前か等しい	～以前または同時
<>	～と等しくない	～と照合順序が等しくない	～と同時ではない
>=	～以上	～より照合順序が後か等しい	～以後または同時

16.1

型変換

数値型はいずれも相互に比較可能かつ代入可能である。違う数値型同士で代入を行った結果、有効桁数が失われると、例外条件が発生する。ただし小さいほうの有効桁が失われる場合は、実装はどんなやり方で丸めまたは切り捨てを行うかを定義しており、あえて例外条件を通知することはしない。ほとんどの場合、ある値は比較対象と同じデータ型に変換され、通常のやり方で比較が行われる。そのとき選ばれるデータ型は、2 つのうち「高い」ほうである。優先順位は次の順序で決まっている。



浮動小数点のハードウェア仕様が、REAL、FLOAT、DOUBLEPRECISIONの数値の比較に影響を与えることは多い。いつもDECIMALやNUMERICで代用できるわけではないので、これを避ける良案は特にない。ホスト言語もおそらく同じ浮動小数点の仕様を持つハードウェアで動作しているはずなので、少なくともSQL側とホスト言語側のエラーは互いに整合的なのが救いだ。

CHARACTERとCHARACTER VARYINGデータ型は、互いが同じキャラクタセットから構成されている文字列である場合に限り、比較可能である。すなわち、ASCII 文字は図形文字とは比較できないし、英語はアラビア語とは比較できないということだ。多くの実装では使うキャラクタセットは1つなので、この点は問題にならない。

比較する際には、短いほうの文字列は空白で埋められる。文字列は左から右に文字位置同士の文字が比較され、特定のキャラクタセ

ットの照合順序――たいていはASCIIまたはEBCDIC ――が使われる。

時間型（または標準SQL での呼び方では日付型〔<datetime>〕）は、ソースとターゲットのデータ型が同じ<datetime>のフィールドを持っている場合に限り、相互に比較可能である。すなわち、日付と時刻を比較することはできない。

いずれの場合も、比較の前にCAST関数を使うことで、明示的な型変換を行うことができる。表16.2 は、標準SQLの型変換におけるソース（表の行）とターゲット（列）の妥当な組み合わせを示す。Y（yes）は、組み合わせに制限がなく構文的に妥当であることを意味する。M （maybe）は、組み合わせが他の構文ルールに妥当であれば許されることを示す。N（no）は、組み合わせが不当であることを示す。

表16.2　標準SQLにおけるソースとターゲットのデータ型の妥当な組み合わせ

	EN	AN	VC	FC	VB	FB	D	T	TS	YM	DT
EN	Y	Y	Y	Y	N	N	N	N	N	N	M
AN	Y	Y	Y	Y	N	N	N	N	N	N	N
C	Y	Y	M	M	Y	Y	Y	Y	Y	Y	Y
B	N	N	Y	Y	Y	Y	N	N	N	N	N
D	N	N	Y	Y	N	N	Y	N	Y	N	N
T	N	N	Y	Y	N	N	N	Y	Y	N	N

TS	N	N	Y	Y	N	N	Y	Y	Y	N	N
YM	M	N	Y	Y	N	N	N	N	N	Y	N
DT	M	N	Y	Y	N	N	N	N	N	N	Y

※ EN＝厳密な数値　　AN＝近似的な数値　　C＝固定長または可変長の文字列　　FC＝固定長文字列　　VC＝可変長文字列　　B＝固定長または可変長のビット文字列　　FB＝固定長ビット文字列　　VB＝可変長ビット文字列　　D＝日付　　T＝時刻　　TS＝タイムスタンプ　　YM＝年月インターバル　　DT＝日時インターバル

16.1.1　データ表示フォーマット

標準SQLは表示のためのフォーマットについては、何も決めていないし、それがあるべき姿である。日付は国によってフォーマットが異なるし、時刻のデータを文字列に変換してさまざまなフォーマットで表示するためのベンダー拡張も多い。唯一、ISO 8601フォーマットのサブセットだけが、標準SQL の時間データのために使われている。SQL は日付としてはyyyy-mm-dd、時間としては24時間の（いわゆる「軍事用」）表記しか持たない。

16.1.2　他の表示フォーマット

文字列と厳密な数値型は、普通は何も考えなくても期待どおりに表示される。近似的なデータ型は、10 進数または指数フォーマットのどちらかで表示される。これは実装依存だが、標準では近似的数値の文字列は次のように定義されている。

```
<approximate numeric literal>::= <mantissa> E <exponent>
<mantissa>::= <exact numeric literal>
<exponent>::= <signed integer>
```

ホスト言語の中には仮数（<mantissa>）は不要だったり、小文字のeを区切り文字に許しているものもある。SQLでは前ゼロが必要だが、他の言語では必ずしもそうではない。

16.2

SQL における行比較

標準SQLは、シータ演算子をスカラ値にとどまらず、行式にまで適用できるよう一般化している。これはまだ一般的な機能ではないが、キーが複数列から構成されていて、キー全体で比較したい場合など便利な機能だ【訳注3】。この機能はSQLの直交性をより高め、直観的に扱いやすくする【訳注4】。次の3つの行定数を見てみよう。

```
A = (10, 20, 30, 40);
B = (10, NULL, 30, 40);
C = (10, NULL, 30, 100);
```

このような行定数に対する比較が妥当であるのは、行内の各列のデータ型がUNION互換である場合と定義するのが合理的だ。そうでなければ、演算はエラーになり、警告が発生するべきだ。また、行比較の結果は、各列をANDでつなげた比較の結果と同じだと定義するのが合理的である。つまり、(A=B)を考えると、

```
((10, 20, 30, 40) = (10, NULL, 30, 40));
```

これは、次と同じということである。

```
((10 = 10) AND (20 = NULL) AND (30 = 30) AND (40 = 40))
```

したがって次のように評価される。

```
(TRUE AND UNKNOWN AND TRUE AND TRUE);
```

最終的な戻り値は、次のようになる。

```
(UNKNOWN);
```

NULLを、現在はわからないが、いずれ値が明らかになるものとして考えれば、この比較の結論をいま引き出すことはできない。したがって、戻り値がUNKNOWNになるのは納得のいく話だ。

今度は、(A = C)という比較を考えよう。

```
((10, 20, 30, 40) = (10, NULL, 30, 100));
```

これは次と同値である。

```
((10 = 10) AND (20 = NULL) AND (30 = 30) AND (40 = 100));
```

次のように評価される。

```
(TRUE AND UNKNOWN AND TRUE AND FALSE);
```

最終的な戻り値は、

```
(FALSE);
```

4 番目の列同士の比較(40 = 100)は常にFALSEである。したがって、Cの2列目のNULLが将来どんな値になろうとも、この比較全体がTRUEになることはあり得ない。この結論は、直観に合致するものだ。最初の比較は、NULL の値によってTRUEにもFALSEにもなる可能性があるが、2番目の比較は永遠にFALSEである。シータ演算子について、標準SQLは次のような適用ルールを定めている。式「RX <comp op> RY」は、「行RXを行RYと比較する」の略である。同様

に、RXiは行RXのi 番目の列を意味する。式の結果は、型マッチングのエラーが起きない限り、TRUE、FALSE、UNKNOWNのいずれかである。ルールは可能な限りTRUEかFALSEに決まるよう作られており、UNKNOWNは最後の手段という扱いだ。式は左から右へ読み、行内の値がもう一方の行の値よりも常に大きいまたは小さいかを判断する。これは単語をアルファベット順に並べるときと同じ方法である。

1. RX = RYは、すべてのiについてRXi = RYiの場合に限りTRUE。
2. RX <> RYは、あるiについてRXi <> RYiの場合に限りTRUE。
3. RX < RYは、あるnについて、n未満のiについてRXi = RYiかつRXn < RYnの場合に限りTRUE。
4. RX > RYは、あるnについて、n未満のiについてRXi = RYiかつRXn > RYnの場合に限りTRUE。
5. RX <= RYは、RX = RYまたはRX < RYの場合に限りTRUE。
6. RX >= RYは、RX = RYまたはRX > RYの場合に限りTRUE。
7. RX = RYは、RX <> RYがTRUEの場合に限りFALSE。
8. RX <> RYは、RX = R
9. RX < RYは、RX >= RYがTRUEの場合に限りFALSE。
10. RX > RYは、RX <= RYがTRUEの場合に限りFALSE。
11. RX <= RYは、RX > RYがTRUEの場合に限りFALSE。
12. RX >= RYは、RX < RYがTRUEの場合に限りFALSE。

13. RX <comp op> RYは、RX <comp op> RYがTRUEでもFALSEでもない場合に限りUNKNOWN。

=と<>、<と>=のように、比較演算子の否定関係は、NOT 演算子が通常の性質を保存するよう定義されている。行の中にNULLが現れると、UNKNOWNが生じる可能性があることに注意してほしい。たとえば、

(a, b, c) < (x, y, z)

この式は次のように展開される。

((a < x)
OR ((a = x) AND (b < y))
OR ((a = x) AND (b = y) AND (c < z)))

標準SQL では、行比較の入力となる行は、1行でさえあれば形式を問わないことになっている。したがって、1行だけのサブクエリも、左辺右辺どちらの入力項としても妥当である。同様に、標準SQLではBETWEEN述語も任意の場所で行式を入力に取ることができる。

16.3

IS [NOT] DISTINCT FROM 演算子

SQL:2003では、冗長だが有用なシータ演算子が追加された。SQLは2種類の比較、あるいは同値類を持っている。同値性テストとグルーピングである。

同値性テストでは、NULLを比較不能として扱い、戻り値が{TRUE, FALSE, UNKNOWN}になる3値論理を採用した。他方グルーピングのテストでは、NULL は互いに等しい値として扱われ、{TRUE, FALSE}を戻り値とする通常の2値論理を採用している。これがGROUP BYが、SQL 以外の言語がそうであるように、NULLを1つのグループとして扱う理由だ。

本章で見てきたシータ演算子は、同値性モデルの上に成り立っている。そのため、NULL を1つのグループにまとめた比較を行いたい場合は、ややこしいCASE式を書かねばならなかった。この問題を解消するために導入されたのが、[NOT] DISTINCT FROM 演算子である。これは、NOT が付いたバージョンのほうが理解しやすい。<expression 1> IS NOT DISTINCT FROM<expression 2>は、以下の式と論理的に同値である。

```
(<expression 1> IS NOT NULL
AND <expression 2> IS NOT NULL
AND <expression 1> = <expression 2>)
OR (<expression 1> IS NULL AND <expression 2> IS NULL)
```

この式がTRUEになるのは、<expression 1>と<expression 2>がともにNULLの場合か、あるいはともに非NULL かつ<expression 1>=<expression 2>の場合である。NOT を省いた<expression 1> IS

DISTINCT FROM <expression 2>は、上の式にNOT 演算子を付加した、以下の二重否定と同じ意味になる。

```
NOT (<expression 1> IS NOT DISTINCT FROM <expression 2>)
```

これを二重否定で定義しているのは、最初に定義されたのが、よく使うIS NOT DISTINCT FROMのほうで、IS DISTINCT FROMをその否定として定義したためである。

これと同じことをやろうとして、次のようなCOALESCE関数を使った検索条件を見たことのある人もいるだろう。

```
COALESCE (x, <適当な非NULL値>) = COALESCE (y, <適当な非NULL値>)
```

このトリックは、x とy がともにNULL なら、<適当な非NULL 値>に同値性テストが適用され、TRUEまたはFALSEが返るという仕組みだ。このトリックで注意が必要なのは、もしxかyのどちらか一方だけがNULLだった場合、xまたはyと、<適当な非NULL値>との比較になる。したがって、この場合にTRUEになることを避けるために、<適当な非NULL値>は、xとy の取り得ない値でなければならない。

【訳注1】　シータ演算子（theta operator）とは奇妙な名前だと思うかもしれないが、これは数学由来の術語で「差分」に関わる演算子を指す。そのため、差分演算子（differential operator）ともいう。=や>は、2つの入力項が「等しいか否か」、「等しくないとすればどの程度違っているか」を測るための演算子であるから、エドガー・F・コッドはこのような呼び方をしている。す

でに気づいた人もいるかもしれないが、コッドは数学者でもあるので、リレーショナルモデルやSQLの用語の中にも、数学由来の耳慣れない言葉がしばしば登場する。しかしいずれも直観的な理解は難しくないので、心配には及ばない。

【訳注2】 「クリスマス」という文字列または「12月25日」という日付を数値に変換することはできない、という意味。

【訳注3】 行式は、Oracle、PostgreSQL、MySQLなどがサポートしている。

【訳注4】 直交性には複数の意味があるが、ここでは、1 つの機能がある要素に対して作用する場合、他の要素にも同じように作用することを「直交性が高い」と表現している。直観的な類推の容易さを示す概念と考えてほしい。直交性が高いほど、例外的事項が少ないため、学びやすく使いやすい言語だと言える。

17

評価述語

Valued Predicates

　評価述語はブール代数関連の単項述語に私が付けた名前だ。この述語はTRUEかFALSEしか返さない。

17.1

IS NULL

```
<null predicate> ::= <row value constructor> IS [NOT] NULL
```

式がNULL か非NULL かを調べる唯一の方法が、この述語である。これはすでにSQL-86から存在し、現在にいたるまで標準SQLの一部である。SQL-92はこれを拡張してスカラ式または単一の列だけでなく、行式を入力に取れるよう拡張した。

この拡張版は、他の行式のサポートと歩調を合わせて、実装でのサポートが進んでいる。もし行R を構成するすべての列の値がNULLの場合、R IS NULL はTRUE に、それ以外の場合はFALSEになる。もしRを構成するすべての列の値が非NULLならば、R IS NOT NULLはTRUEに、それ以外の場合はFALSEになる。RにNULLと非NULLが混在している場合は表17.1のルールで決定される。ここで、次数（Degree）は、行式内の列数である。

R IS NOT NULL がNOT R IS NULLと同値になるのは、R の次数が1の場合に限られることに注意してほしい。これは、他の述語と違い、行式への拡張ルールに反して直交性がない。以下に、いくつか例を挙げる。

A. (1, 2, 3) IS NULL = FALSE
B. (1, NULL, 3) IS NULL = FALSE
C. (1, NULL, 3) IS NOT NULL = FALSE
D. (NULL, NULL, NULL) IS NULL = TRUE
E. (NULL, NULL, NULL) IS NOT NULL = FALSE
A'. NOT (1, 2, 3) IS NULL = TRUE

B'. NOT (1, NULL, 3) IS NULL = TRUE

C'. NOT (1, NULL, 3) IS NOT NULL = TRUE

D'. NOT (NULL, NULL, NULL) IS NULL = FALSE

E'. NOT (NULL, NULL, NULL) IS NOT NULL = TRUE

表17.1　行がNULLと非NULLの混合であるパターンの分類

Rの次数（Degree）		R IS NULL	R IS NOT	NULL NOT （R IS NULL）	NOT（R IS NOT NULL）
Degree = 1	NULL	TRUE	FALSE	FALSE	TRUE
	Not NULL	FALSE	TRUE	TRUE	FALSE
Degree > 1	All NULLs	TRUE	FALSE	FALSE	TRUE
	Some NULLs	FALSE	FALSE	TRUE	TRUE
	No NULLs	FALSE	TRUE	TRUE	FALSE

17.1.1　NULL が発生する理由

NULLがどこで生じるかを覚えておくのは大切なことだ。というのも、NULLは必ずしも列の値が不明であったり適用不能な場合にだけ生じるわけではないからだ。集約関数の入力が空集合であったり、

外部結合、算術式にNULLが入力された場合なども結果がNULLになる。そのため、しばしばSELECT文の結果やビューの中にNULLが現れることになる。

17.2

IS [NOT]{TRUE | FALSE | UNKNOWN}述語

この述語は、真理値TRUE、FALSE、UNKNOWNを引数に取り、TRUEまたはFALSEを返す。構文は次のとおりである。

```
<Boolean test> ::=
<Boolean primary> [IS [NOT] <truth value>]
<truth value> ::= TRUE | FALSE | UNKNOWN
<Boolean primary> ::=
<predicate> | <left paren> <search condition> <right paren>
```

この述語の場合、IS NOT <logical value>はNOT (x IS < logical value>)と問題なく論理的に同値になるので、この述語は表17.2によって定義される。

表17.2　IS TRUE、IS FALSE、IS UNKNOWNの定義

IS <logical value>	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
UNKNOWN	FALSE	FALSE	TRUE

もしクリス・デイトの著作を読んだことがあれば、デイトの言うMAYBE(x)述語が、ANSI の(x) IS NOT FALSE述語ではなく(x) IS UNKNOWN述語と同値であることに気づくだろう。

デイトの述語では、述語内のすべての条件がTRUEのケースが除外されており、英語で条件的質問文を作ることは難しいことを指摘し

ている。デイトの著作から例を借りると（Date１９９0）、１９７５年1月18日より前に生まれていて、給与が５0, 000ドル以下のプログラマである可能性のある人物を、社員の中から見つけよう。この問題文は、「可能性のある（might be）」という語のせいで少しあいまいだ。この言葉は「プログラマ」の部分だけにかかるのか、それとも生年月日、給与、仕事の3つの条件にかかるのかわからない。ここではとりあえず、3つのいずれかの条件が1つでも該当する可能性のある人物はすべて選び出すとしよう。答えのクエリは、IS UNKNOWN述語を使えば簡単に書ける【訳注1】。

```
SELECT *
  FROM Personnel
 WHERE (   job_title = 'Programmer'
       AND birth_date < DATE '1975-01-18'
       AND salary_amt < 50000.00) IS UNKNOWN;
```

　これでOKだ。これをIS UNKNOWNを使わない古いSQLの構文で書くとこうなる。

```
SELECT *
  FROM Personnel
 WHERE (job_title IS NULL
       AND birth_date < CAST('1975-01-18' AS DATE)
       AND salary_amt < 50000.00)
    OR (job_title = 'Programmer'
       AND birth_date IS NULL
       AND salary_amt < 50000.00)
    OR (job_title = 'Programmer'
       AND birth_date < CAST('1975-01-18' AS DATE)
       AND salary_amt IS NULL)
    OR (job_title IS NULL
       AND birth_date IS NULL
       AND salary_amt < 50000.00)
    OR (job_title IS NULL
       AND birth_date < CAST('1975-01-18' AS DATE)
       AND salary_amt IS NULL)
    OR (job_title = 'Programmer'
       AND birth_date IS NULL
       AND salary_amt IS NULL)
    OR (job_title IS NULL
       AND birth_date IS NULL
       AND salary_amt IS NULL);
```

一見して非常に冗長なことがわかるだろう。すべてのNULLと非NULLの組み合わせをテストしなければならないのだ。条件が3つで、それがNULLか非NULLかの2つの状態を判別するので、(2^3−1)＝7通りの組み合わせをチェックする必要がある。もちろん、条件が増えるほど組み合わせ数も増える。

他方、IS UNKNOWN 述語の場合、いずれかの列がNULL かを調べるだけで済むので、上のように組み合わせ数が爆発することはない。

17.3

IS [NOT] NORMALIZED 述語

<string> IS [NOT] NORMALIZED は、Unicode 文字列が4つの正規形（D、C、KD、KC）のいずれかに該当するか否かを調べる【訳注2】。ここでの「正規形」は、関係モデルにおけるテーブルの正規形とは無関係なので注意してほしい。Unicode では、1つの文字が複数の文字から構成されることがある。たとえば、アクセントマークを基本的なラテン文字に付けたり、複数の文字の組み合わせを合字（aeでæを表すような）として表示するやり方である。ハングル（韓国語）やベトナム語のように、複数の記号を組み合わせて2 次元の合わせ文字を作る言語もある。また、ギリシア語の語末でだけ使われる小文字のシグマ（ς）やチェコ語のアクセント付きのuのように、文脈の中で文字の特別な形が決められる言語もある【訳注3】。書き言葉は、単に文字を1つずつ並べていくだけの作業以上に複雑なのだ。

Unicode標準は、こうした合成済み文字の順序を正規形として定義している。この順序が異なると、同じ結果を得ることもあれば、異なる記号の組み合わせになることもある。だが、そうした文字列を検索する際、実行時に逐一合わせ文字についてパースするよりも、それが正規形に準じていることがあらかじめわかっているほうが便利だろう。

正規形についての詳細情報は、Unicodeコンソーシアムのサイト（http://www.unicode.org/）で得ることができる。

【訳注1】　念のため、このクエリは、次のクエリと同値ではない。

```
SELECT *
```

```
    FROM Personnel
  WHERE job_title IS NULL
        OR birth_date IS NULL
        OR salary_amt IS NULL;
```

このクエリだと、3列のどれか1つがNULLであれば、残りの列の条件は満たしていなくても（つまりFALSEであっても）、その社員も結果に含めてしまう。たとえば、誕生日がNULLで職種が営業の社員も結果に含まれる。IS UNKNOWNのクエリはこの社員は除外される。

【訳注2】 D、C、KD、KCは、Unicode標準で認められている4種類の正規化形式。

【訳注3】 ギリシア語のシグマの小文字は通常「σ」だが、語末のみ「ς」が使われる。

18

CASE式

CASE Expressions

CASE式は、SQL-92で追加されたもののうち、おそらく最も有用な機能である。本章では、この機能の使い方を概観するが、CASE式を使ったトリックは本書全体にわたって登場する。CASE 式が重要な理由は、任意のデータ型に対して使うことができるからだ。プログラマはIF-THENELSE制御文をクエリ内のCASE式で置き換えることで、手続き型のコードを使わなくても済むようになる。これはまた、オプティマイザにとっても良い話だ。CASE式を使ったSQL文は、原始的な再帰関数と同値になる。このあたりのことは、数学専攻の人以外は興味はないかもしれないが、とても重要なポイントである。計算理論の本を読めばわかるが、これはある種の問題が解けることを保証する数学的に有用な性質なのである。

18.1

CASE 式

CASE 式は、論理的な式に基づいてある1 つの値を選択するための機能である。ANSIは、その考え方と構文を、いまはなきADAから継承した（ADAは1987年に米国国防総省で使用するために作られ、1997 年にその使用義務が実質的に停止された。技術的には2005 年に標準が定められた）。CASE式の条件部<case specification>の構文は次のとおりである。

```
<case specification> ::= <simple case> | <searched case>
<simple case> ::=
CASE <case operand>
<simple when clause> ...
[<else clause>]
END
<searched case> ::=
CASE
<searched when clause> ...
[<else clause>]
END
<simple when clause> ::= WHEN <when operand> THEN <result>
<searched when clause> ::= WHEN <search condition> THEN <result>
<else clause> ::= ELSE <result>
<case operand> ::= <value expression>
<when operand> ::= <value expression>
<result> ::= <result expression> | NULL
<result expression> ::= <value expression>
```

CASE式には単純CASE式と検索CASE式があるが、よく使われるのは後者である。CASE式はまず、THEN 句の中で最も強いデータ型を見つけるために、式にはデータ型が与えられる。WHEN ... THEN ... 句は左から右に実行され、最初にTRUEに評価されたWHEN句に対応するTHEN句の値が返される。

CASE 式は互いに入れ子にすることができる。また、明示的なELSE 句がない場合は、データベースは暗黙にENDの前にELSE NULL句を挿入する。もし特定のTHENでNULLを返したい場合は、コンパイラにデータ型を教えるために、CAST(NULL AS <data type>)と記述する必要がある。

```
--これは正しい
CASE WHEN 1 = 1
     THEN NULL
     ELSE CAST(NULL AS INTEGER)
 END

--これも正しい
CASE WHEN 1 = 1
     THEN CAST(NULL AS INTEGER)
     ELSE NULL
 END

--これは正しくない（結果にデータ型が指定されていない[訳注1]）
CASE WHEN 1 = 1
     THEN NULL
     ELSE NULL
 END

--動くかどうかは実装依存
CAST (CASE WHEN 1 = 1
           THEN NULL
           ELSE NULL
       END AS INTEGER)
```

【訳注1】 実際はこのような記述でも、ほとんどの実装で動作する。

私は常々、明示的なELSE句を書くことを推奨している。そうすることで、あとからNULL以外の値に変えたいときも簡単だし、コードの中でNULLが発生することが見てとれる。また、THEN 句の中でNULLを返すときは明示的にCASTを行うことも推奨したい。もしTHEN句に複数のデータ型が存在した場合、コンパイラは最も一般的

なデータ型に他のデータ型を合わせるが、いつどうやって型変換を行うかは、実装によって微妙に異なるからだ。

<simple case expression>は、すべてのWHEN 句が<case operand>に対する等値条件であるような検索CASE式として定義される。たとえば、

```
CASE iso_sex_code
WHEN 0 THEN 'Unknown'
WHEN 1 THEN 'Male'
WHEN 2 THEN 'Female'
WHEN 9 THEN 'N/A'
ELSE NULL END
```

この式は次と同値である:

```
CASE
WHEN iso_sex_code = 0 THEN 'Unknown'
WHEN iso_sex_code = 1 THEN 'Male'
WHEN iso_sex_code = 2 THEN 'Female'
WHEN iso_sex_code = 9 THEN 'N/A'
ELSE NULL END
```

この定義には1つ注意すべき点がある。次の式を見てほしい。

```
CASE foo
WHEN 1    THEN 'bar'
WHEN NULL THEN 'no bar'
END
```

これは次のように展開される。

```
CASE WHEN foo = 1    THEN 'bar'
     WHEN foo = NULL THEN 'no_bar' -- ここが問題！
     ELSE NULL
 END
```

「WHEN foo = NULL」という句は、常にUNKNOWN に評価される。したがって、このCASE 式が'no_bar'を返すことはない。

この定義から、式の中で乱数を使うと非常に困ったことになる。いまRANDOM 関数が常に0.0000～0.99 ...9の間の一様に分散した値を返すと仮定しよう。すると以下の式はほとんどすべてELSE句のほうへ行ってしまい、'one'～'five'が返されることはない。

```
pick_one := CASE CAST((5.0 * RANDOM()) + 1 AS INTEGER)
            WHEN 1 THEN 'one'
            WHEN 2 THEN 'two'
            WHEN 3 THEN 'three'
            WHEN 4 THEN 'four'
            WHEN 5 THEN 'five'
            ELSE 'This should not happen' END;
```

実装にもよるが、この式を展開する際には、CAST 式が個々のWHEN 句につき再評価されるため、RANDOM関数もそのたびに再評価されることが多い。これは無駄だ。一度しか評価しないようにするには次のように書く。

```
BEGIN
pick_a_number INTEGER;

pick_a_number := CAST((5.0 * RANDOM()) + 1 AS INTEGER);
pick_one := CASE pick_a_number
            WHEN 1 THEN 'one'
            WHEN 2 THEN 'two'
            WHEN 3 THEN 'three'
            WHEN 4 THEN 'four'
            WHEN 5 THEN 'five'
            ELSE 'This should not happen' END;
END;
```

変数pick_a_numberもWHEN 句で展開されるが、関数呼び出しがないため、繰り返し評価されることはない。

18.1.1　COALESCE 関数とNULLIF 関数

SQL-92は、CASE式に似た式を2つ定義している。これらはなくてもCASE式で代用できるが、使うとコードをコンパクトで簡単に書ける。たとえば、COALESCE関数は、入力の式が1つと2つの場合は、以下のように定義される。

1. COALESCE (<value exp #1>)は(<value exp #1>)と同値。
2. COALESCE (<value exp #1>, <value exp #2>)は以下と同値。

```
CASE WHEN <value exp #1> IS NOT NULL
     THEN <value exp #1>
     ELSE <value exp #2>
 END
```

すると、入力数がn（≧3）の場合も再帰的に定義できる。COALESCE (<value exp #1>, <value exp #2>, ..., n)は以下と同値だ。

```
CASE WHEN <value exp #1> IS NOT NULL
     THEN <value exp #1>
     ELSE COALESCE (<value exp #2>, ..., n)
 END
```

同様にNULLIF (<value exp #1>, <value exp #2>)関数も次のようにCASE式で定義できる。

```
CASE WHEN <value exp #1> = <value exp #2>
     THEN NULL
     ELSE <value exp #1>
 END
```

18.1.2　集約関数とともにCASE 式を使う

CASE式をGROUP BY 句で使うと非常に便利だ。たとえば、社員テーブルから、部署別性別に社員数を求める場合、次のようなコードになる。

```
SELECT dept_nbr,
       SUM(CASE WHEN sex_code = 1 THEN 1 ELSE 0) AS males,
       SUM(CASE WHEN sex_code = 2 THEN 1 ELSE 0) AS females
  FROM Personnel
 GROUP BY dept_nbr;
```

またはこうでもいい。

```
SELECT dept_nbr,
       COUNT(CASE WHEN sex_code = 1 THEN 1 ELSE NULL) AS males,
       COUNT(CASE WHEN sex_code = 2' THEN 1 ELSE NULL) AS females
  FROM Personnel
 GROUP BY dept_nbr;
```

一般的にどちらのほうが速いかはわからない。ただ、集約関数は実行の前にNULLを除外するため、ELSE 0とELSE NULLの間には性能上の違いが出る可能性もある。

上の例はCASE式を集約関数の中で使っていたが、CASE式の中で集約関数を使うこともできる。たとえば、社員のスキルを管理する次のようなテーブルを考えよう。

```
CREATE TABLE PersonnelSkills
(emp_id CHAR(11) NOT NULL,
 skill_code CHAR(11) NOT NULL,
 primary_skill_flg CHAR(1) NOT NULL
     CONSTRAINT primary_skill_given
       CHECK (primary_skill_flg IN ('Y', 'N')),
     PRIMARY KEY (emp_id, skill_code));
```

社員の持つスキル単位で行が存在する。複数のスキルを持つ社員については、主なスキルを示すフラグに'Y'が立つ。1 つしかスキルを持たない社員（つまりテーブルに1 行しか存在しない）のprimary_skill_flgはNULLである。ここで、以下の条件を満たすように社員を表示したい。

条件1　1つしかスキルを持たない社員は、そのスキルを1行で表示する。

条件2　複数のスキルを持つ社員は、主なスキルを1行で表示する。

これは次のようなクエリで可能だ。

```
SELECT emp_id,
       CASE WHEN COUNT(*) = 1
            THEN MAX(skill_code)
            ELSE MAX(CASE WHEN primary_skill_flg = 'Y'
                          THEN skill_code END) END AS main_skill
  FROM PersonnelSkills
 GROUP BY emp_id;
```

この解は一見すると、集約関数をネストできないというSQLのルールに違反するように見える。だが、そうではない。外側のCASE式は集約関数、つまりMAX関数を評価する。ELSE句は単純に中のMAX関数――すでにスカラ値に評価されている――を返すだけである。

18.1.3　CASE 式、CHECK 制約、条件法

複雑な論理述語をCASE式の中に入れて'T'（TRUE）または'F'（FALSE）を返すようにすることもできる。

```
CONSTRAINT implication_example
  CHECK (CASE WHEN dept_nbr = 'D1'
              THEN CASE WHEN salary < 44000.00
                        THEN 'T'
                        ELSE 'F' END
              ELSE 'T' END = 'T')
```

これは、第13章で簡単に触れた条件法の演算子である。これは通常矢印の形をした記号（→）で書くが、その定義は日本語でいうと「前提が真の場合、結論は偽ではありえない」または「もしAならばB」という意味である【訳注2】。

上の制約を自然言語で表現すると、「もし社員が部署D1に属しているなら、その社員の年収は44, 000ドルより少ない」という意味である。これは、(dept_nbr ='D1' AND salary <44000.00)という制約とは意味が異なる。標準的なブール論理では条件法を基本的な論理演算子に同値変換する「スミステルルール」という有名なルールがある。それを使うと、(a→b)は(¬avb)と同値である。

しかしSQLでは、CHECK制約は結果がTRUEかUNKNOWNの場合に成功し、一方WHERE句やON 句では結果がFALSE かUNKNOWN の場合に失敗する。つまり、UNKNOWN のケースにおいてDDL の制約とDMLの評価に相違が生じる。これが、3値論理における条件法のやっかいなところだ——DDLとDMLで動作原理が異なることを忘れてはならない！

スミステルルールによって条件法を変換した制約を持つテーブルを使って確認してみよう。

```
CREATE TABLE Foobar_DDL_1
(a CHAR(1) CHECK (a IN ('T', 'F')),
 b CHAR(1) CHECK (b IN ('T', 'F')),
     CONSTRAINT implication_example
       CHECK (NOT (A = 'T') OR (B = 'T')));

INSERT INTO Foobar_DDL_1 VALUES ('T', 'T');
INSERT INTO Foobar_DDL_1 VALUES ('T', 'F'); -- 制約違反で失敗
INSERT INTO Foobar_DDL_1 VALUES ('T', NULL);
INSERT INTO Foobar_DDL_1 VALUES ('F', 'T');
INSERT INTO Foobar_DDL_1 VALUES ('F', 'F');
INSERT INTO Foobar_DDL_1 VALUES ('F', NULL);
INSERT INTO Foobar_DDL_1 VALUES (NULL, 'T');
INSERT INTO Foobar_DDL_1 VALUES (NULL, 'F');
INSERT INTO Foobar_DDL_1 VALUES (NULL, NULL);

SELECT * FROM Foobar_DDL_1;
```

結果

```
 a      b
=============
 T      T
 T      NULL
 F      T
 F      F
 F      NULL
 NULL   T
 NULL   F
 NULL   NULL
```

前件（a）が真（'T'）で、後件（b）が偽（'F'）の組み合わせが、制約違反となることがわかる。一方、私のオリジナルバージョンの条件法を使うとどうなるだろうか。

```
CREATE TABLE Foobar_DDL
(a CHAR(1) CHECK (a IN ('T', 'F')),
 b CHAR(1) CHECK (b IN ('T', 'F')),
     CONSTRAINT implication_example_2
       CHECK(CASE WHEN A = 'T'
                  THEN CASE WHEN B = 'T'
                            THEN 1 ELSE 0 END
                  ELSE 1 END = 1));

INSERT INTO Foobar_DDL VALUES ('T', 'T');
INSERT INTO Foobar_DDL VALUES ('T', 'F');    -- 制約違反で失敗
INSERT INTO Foobar_DDL VALUES ('T', NULL);  -- 制約違反で失敗
INSERT INTO Foobar_DDL VALUES ('F', 'T');
INSERT INTO Foobar_DDL VALUES ('F', 'F');
INSERT INTO Foobar_DDL VALUES ('F', NULL);
INSERT INTO Foobar_DDL VALUES (NULL, 'T');
INSERT INTO Foobar_DDL VALUES (NULL, 'F');
INSERT INTO Foobar_DDL VALUES (NULL, NULL);

SELECT * FROM Foobar_DDL;
```

結果

```
 a      b
=============
 T      T
 F      T
 F      F
 F      NULL
 NULL   T
 NULL   F
 NULL   NULL
```

どちらも、前提がTRUEの場合、結論がFALSEになり得ないという点は同じである。だが、スミステルルールでは、('T', NULL)は許される。これは2値論理の条件法ではあり得ない！ 私のバージョンであれば、このケースもエラーとなる。

次に、DMLにおける条件法の評価を見てみよう。

```
CREATE TABLE Foobar_DML
(a CHAR(1) CHECK (a IN ('T', 'F')),
 b CHAR(1) CHECK (b IN ('T', 'F')));

INSERT INTO Foobar_DML
VALUES ('T', 'T'),
       ('T', 'F'),
       ('T', NULL),
       ('F', 'T'),
       ('F', 'F'),
       ('F', NULL),
       (NULL, 'T'),
       (NULL, 'F'),
       (NULL, NULL);
```

検索条件にスミステルルールを使うと、次のようなクエリになる。

```
SELECT *
  FROM Foobar_DML
 WHERE (NOT (A ='T') OR (B = 'T'));
```

結果

```
 a      b
=============
 T      T
 F      T
 F      F
 F      NULL
 NULL   T
```

私のオリジナルバージョンの条件法のクエリだとこうなる。

```
SELECT *
  FROM Foobar_DML
 WHERE CASE WHEN A = 'T'
            THEN CASE WHEN B = 'T'
                      THEN 1 ELSE 0 END
            ELSE 1 END = 1;
```

結果

```
a       b
============
T       T
F       T
F       F
F       NULL
NULL    T
NULL    F
NULL    NULL
```

　このように、両者の結果は一致しない。だが真に注目すべきは、スミステルルールを使うとDDLとDML の結果が異なるのに対し、私のCASE 式の条件を使えば、DDLとDML の結果が一致する。これが、私がCASE式を使った理由である。CASE式ならば、DDLでもDMLでも同じ動作を保証できるのだ。

18.2

ローゼンシュタインの特性関数

特性関数はTRUEの場合に1を、FALSEの場合に0を返す関数である。特性関数は、いままで我々がCASE式を使ってやってきたことを言い直しただけである。表記法としては小文字のdまたは、大文字のChi(x)のように表記する。これはもともと、ケン・アイバーソンのAPLプログラミング言語に存在したもので、後にドナルド・クヌースのプログラミングの本にも取り上げられた。この特性関数と言う名前は、ある要素が集合に含まれるルールを与えることで集合を定義することからきている。

デヴィッド・ローゼンシュタインは、特性関数をSybaseのT-SQL言語を使って、数値列に対する代数式で実装する方法を見つけた。当時のSybaseはCASE 式を持っていなかったので、そういう方法を採用したのだ。ここでは詳細を触れないが、ローゼンシュタインの表記法を借りて、いくつか有名な数値比較のための式を、特性関数の計算に置き換えたものを紹介しよう。a、b は任意の数値とする。

- (a = b)：(1-ABS(SIGN(a-b)))
- (a <> b)：(ABS(SIGN(a-b)))
- (a <b)：(1-SIGN(1 + SIGN(a-b)))
- (a <= b)：(SIGN(1-SIGN(a-b)))
- (a > b)：(1-SIGN(1-SIGN(a-b)))
- (a >= b)：(SIGN(1 + SIGN(a-b)))

基本的な論理演算子も特性関数の計算に置き換えることができる。NULL の問題を無視して標準的なブール論理を使うとすれば次のように変換できる。a、bは0または1とする。

- NOT (a)：(1-a)
- (a AND b)： SIGN(a * b)
- (a OR b)： SIGN(a + b)

もし読者が、ジョージ・ブールのオリジナルの表記法を覚えていれば、これはとても見慣れたものだろう。しかし、aまたはbがNULLの場合、結果はNULLになり{0,1}の世界は破れてしまう。これは、ブール自身は夢にも思わなかった事態である。

特性関数はまた、文字列の比較をPOSITION関数を使って変換することができる。

- (a = s)： POSITION (a IN s)
- (a <> s)： SIGN (1-POSITION (a IN s))

ローゼンシュタインはさらに多くのトリックを挙げているが、その多くはSybaseのT-SQL関数に依存しているため移植性がない。もう1つの問題は、コードがあまりに複雑でいったい何をやっているのか、他のプログラマに説明するのが困難なことだ。

特性関数の代わりにCASE式を使うことで、オプティマイザはより効率的に実行計画を立てられる。それゆえ特性関数を使っている古いコードを見たら、CASE式で書き直してやってほしい。

18.3

行のソート

2001 年5 月27日、フレッド・ブルックはSQL Server のニュースグループに次のような質問を投稿した。オリジナルから問題を少し変えてあるが、要点はこうだ。5つの文字列型の列を持つテーブルがある。どの列にもアルファベットのデータが格納される。いわばこのテーブルは、「水平展開」された状態にある。この列が何を意味するかは、ひとまずおいておこう。もしかすると年月かもしれないし、他のデータかもしれない。テーブル定義は次のようなものだ。これを使って、ブルックの直面した問題を見ていこう。

```
CREATE TABLE Foobar
(key_col INTEGER NOT NULL PRIMARY KEY,
 c1 VARCHAR(20) NOT NULL,
 c2 VARCHAR(20) NOT NULL,
 c3 VARCHAR(20) NOT NULL,
 c4 VARCHAR(20) NOT NULL,
 c5 VARCHAR(20) NOT NULL);
```

ブルックによると、各列の間には次のような関係が成り立つべきだという。

c1 <= c2 <= c3 <= c4 <= c5

当たり前のことだが、彼がこの条件をCHECK 制約として最初からテーブルに付与していたなら、何の問題も起きない。それはまったく正しい措置であり、本節で問題として取り上げることは何もない。だが興味深いことに、彼の投稿をニュースグループで読んだ全員が、即座にストアドプロシージャを作ることで解決を図ろうとした。5 つの値を引数に取り、ソートし、テーブルのオリジナルの行に

更新する。このアプローチを可能にする唯一の方法は、更新カーソルを書いてテーブルの全行をループすることだ。ある者は、5 つの値を一時テーブルにロードし、ソートされた順番で取り出すシンプルなプロシージャを投稿した。もちろんループが使われていた。

別のトリックとして、ボース–ネルソンソートを使う方法もある（"A Sorting Problem," R.C. Bose and R.J. Nelson, *Journal of the ACM*, Volume 9 Issue 2, 1962: 282–296）。この方法では再帰的手続きを使って、入力された整数値に対して、順序を入れ替えたペアを生成する。このペアは、1からnまでのベクトルから決まる位置番号を持ったペアだ。この入れ替えペアはまた、ソーティングネットワークとも関係している（詳細については次の書籍を参照：『The Art of Computer Programming, vol.3』（Knuth, 1998）【訳注3】。

n 個のペアに対して入れ替えを行うソーティングネットワークを生成するアルゴリズムは、これだけではないし、常に最適というわけでもない。nが16より大きくなると、特定のソーティングネットワークが最適であるという証明は存在しない。以下に、n が16 以下の場合の最適な比較回数の早見表を示す。

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
比較回数	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60

この問題に興味がある人は、次のサイトを参照してほしい。

http://pages.ripco.net/~jgamble/nw.html

この方法は機能的に少し弱いのではないかと思う読者もいるだろう。というのも、これは固定された数の項目をソートする場合にしか使えないからだ。だが、テーブルに含まれる列数は固定されているので、非正規化されたテーブルに利用する場合は、この弱点は問題にならない。

最小の入れ替え回数で、5 つの項目をきれいにソートするためのソーティングネットワークを作ることができる（私がイメージしているのは、ポーカーの役だ）。こんなイメージだ。

```
Swap(c1, c2);
Swap(c4, c5);
Swap(c3, c5);
Swap(c3, c4);
Swap(c1, c4);
Swap(c1, c3);
Swap(c2, c5);
Swap(c2, c4);
Swap(c2, c3);
```

最初は、5 枚の手札がランダムに配られる。カードはすべてテーブルに伏せたままにしておいて、任意の2枚だけめくる。もし、順番を入れ替える必要があるなら、カードの場所を入れ替える。この手順を必要な分だけ繰り返し、最後に5 枚全部を開いて、順序が整列されているか確認する。

理論的には、n個の項目をソートするのに最低限必要な入れ替え回数は、`CEILING(log2(n!))`である。そして、n が増えるにつれ、回数はO(n*log2(n))に近づく。ボースネルソンのアルゴリズムは、nが小

さい場合には非常に優秀だ。もし(n<9)であれば、完璧に近いパフォーマンスである。

　だが規模が大きくなると、ボース–ネルソンの計算量はO(n^1.585)に近づく。平たく言うと、この方法は、16 個以下の固定サイズの入力に対してはよいが、それ以上増えると遅くて使いものにならない。

　SQLを使って、n個の引数を取るボース–ネルソンのプロシージャを書くこともできる。SQLで値を入れ替える最も簡単な方法は、UPDATE文を何個も書くことだ。SET句では複数の列を書けることを覚えているだろうか。これを使えば、順序どおりではない値を入れ替えるのはとても簡単だ。

```
BEGIN ATOMIC
-- Swap(c1, c2);
UPDATE Foobar
   SET c1 = c2, c2 = c1
 WHERE c1 > c2;
-- Swap(c4, c5);
UPDATE Foobar
   SET c4 = c5, c5 = c4
 WHERE c4 > c5;
-- Swap(c3, c5);
UPDATE Foobar
   SET c3 = c5, c5 = c3
 WHERE c3 > c5;
-- Swap(c3, c4);
UPDATE Foobar
   SET c3 = c4, c4 = c3
 WHERE c3 > c4;
-- Swap(c1, c4);
UPDATE Foobar
   SET c1 = c4, c4 = c1
 WHERE c1 > c4;
-- Swap(c1, c3);
UPDATE Foobar
   SET c1 = c3, c3 = c1
 WHERE c1 > c3;
-- Swap(c2, c5);
UPDATE Foobar
   SET c2 = c5, c5 = c2
 WHERE c2 > c5;
-- Swap(c2, c4);
UPDATE Foobar
   SET c2 = c4, c4 = c2
 WHERE c2 > c4;
-- Swap(c2, c3);
UPDATE Foobar
   SET c2 = c3, c3 = c2
 WHERE c2 > c3;
END;
```

　これは完全に標準SQL準拠の移植可能なコードだし、機械的に生成することもできる。だが処理が直列でしか動かないのが欠点だ。そこで複数のUPDATE 文を統合して、並行して動けるようにしよう。ただし、このとき入れ替え操作の順番を壊さないよう気をつけなければならない。

最初の2 つのUPDATE 文を見てほしい。入れ替え対象の値が重複していない。これは互いに影響を与えない処理なので、1つの文にまとめることが可能だ。

```
-- Swap(c1, c2) AND Swap(c4, c5);
UPDATE Foobar
   SET c1 = CASE WHEN c1 <= c2 THEN c1 ELSE c2 END,
       c2 = CASE WHEN c1 <= c2 THEN c2 ELSE c1 END,
       c4 = CASE WHEN c4 <= c5 THEN c4 ELSE c5 END,
       c5 = CASE WHEN c4 <= c5 THEN c5 ELSE c4 END
 WHERE c4 > c5 OR c1 > c2;
```

UPDATE文をまとめることの利点は明らかだ。一度の実行で済むのでパフォーマンスが良い。テーブルの更新は、たとえ非キー列に対する更新であっても（実装のロック粒度にも依存するが）、テーブルをロックして他のユーザーの利用を妨げるため、極力更新は効率的に行うべきだ。

同じようなパターンが次の入れ替えにも適用できる。

```
Swap(c1, c3);
Swap(c2, c5);
```

こうしたパターンのための一般的なテンプレートを書いてみよう。次の一般化したケースを考える。

```
Swap(x, y);
Swap(x, z);
```

x、y、zの3つの可能な組み合わせを考えて、上記の2つの入れ替え操作を行うとする。

```
(x, y, z) => (x, y, z)
(x, z, y) => (x, z, y)
(y, x, z) => (x, y, z)
(y, z, x) => (x, z, y)
(z, x, y) => (x, y, z)
(z, y, x) => (x, y, z)
```

このパターンの結果は、xが3 つの値の中で最小で、yとzが互いに対して相対的に同じ位置にとどまるか、適切にソートされるかのどちらかである。

三者を適切にソートできれば、それ以上の入れ替えを省略できるし、個々のUPDATE文で操作されていた部分集合の集合を削減できる。ちょっと考えれば、これは次のような対称的なコードで書くことができる。

```
-- Swap(x, y) AND Swap(x, z);
UPDATE Foobar
SET x = CASE WHEN x BETWEEN y AND z THEN y
             WHEN z BETWEEN y AND x THEN y
             WHEN y BETWEEN z AND x THEN z
             WHEN x BETWEEN z AND y THEN z
             ELSE x END,
    y = CASE WHEN x BETWEEN y AND z THEN x
             WHEN x BETWEEN z AND y THEN x
             WHEN z BETWEEN x AND y THEN z
             WHEN z BETWEEN y AND x THEN z
             ELSE y END,
    z = CASE WHEN x BETWEEN z AND y THEN y
             WHEN z BETWEEN x AND y THEN y
             WHEN y BETWEEN z AND x THEN x
             WHEN z BETWEEN y AND x THEN x
             ELSE z END
WHERE x > z OR x > y;
```

このパターンをさらに多くの入力項に拡張したテンプレートを作りたい誘惑は強いが、それはあまりおすすめできない。簡略化のメリットに比して、保守性と可読性が大きく下がっていくからだ。

【訳注2】　条件法については第13章の訳注4も参照。

【訳注3】　ソーティングネットワークは、ソートのための数学的モデルの1つで、2つの値を入れ替えるメカニズム（コンパレータ）と、そのコンパレータに値を渡す「ワイヤ」という構成要素からなる。

19

LIKE述語とSIMILAR TO述語

LIKE and SIMILAR TO Predicates

LIKE述語は簡単なパターンマッチングのテストを行う。構文は次のとおりだ。

```
<like predicate> ::=
<match value> [NOT] LIKE <pattern>
[ESCAPE <escape character>]
<match value> ::= <character value expression>
<pattern> ::= <character value expression>
<escape character> ::= <character value expression>
```

式M NOT LIKE Pは、NOT (M LIKE P)と同値である。これは、他のSQLの構文パターンと同じだ。<pattern>文字列には、2 種類のワイルドカードが使える。%と_である。_は、任意の1文字を、%は任意の0個以上の文字列を表す。0文字の任意の文字列を表す方法はないことに注意してほしい。これは多くのテキスト検索言語と異なる点なので、時に問題を引き起こしたり、条件が非常に複雑になる原因となる。

<pattern>文字列に記述されたそれ以外の文字は、文字そのものを表す。これはつまり、SQL の文字列パターンは、大文字小文字を区別するということだ。だが、多くのベンダーは、これをデータベースレベルのパラメータで制御できるようにしている。

<escape character>は、<pattern>文字列において、これに続く文字がワイルドカードではなくリテラルであることを示す。したがって、エスケープ文字そのものを含めたい場合は、エスケープ文字列のあとにエスケープ文字列を記述する。昔のC言語プログラマは、このルールに慣れていて、エスケープ文字列として「\」が定義されていた。SQLプログラマもこのルールに従えば、混乱がないだろう。

19.1

文字列パターンのトリック

_を使ったテストは%を使ったテストよりも高速だ。理由は明らかで、パーサが文字列をパターンと比べる際、次の文字に行く前にアンダースコアを1回操作するだけで済むからだ。しかし%の場合、一種の先読みパースを行う必要がある。ワイルドカードはパターンの最初か中間に置くことができる。たとえば、'B%K'は、'BOOK'、'BLOCK'、'BK'と一致するが、'BLOCKS'とは一致しない。

パーサは各文字をスキャンし、それをワイルドカードと一致するか、正確に一致するかのいずれかに分類する。'B%K'の比較なら、'BLOCKS'の場合、先頭のBは厳密に一致し、パーサは次の文字に進む。今度は、L、O、C はワイルドカードと一致しなければならない。なぜなら、これらの文字はパターン文字列には現れないからである。Kは最後の文字を読むまで分類することはできない。最後の文字Sは不一致となる。

たとえば、長さ7の可変長文字型の列を考えよう。この列に対してLIKE述語でMacで始まる名前を探す。普通は次のように書くだろう。

```
CREATE TABLE People
(last_name VARCHAR(7));
```

People：人名テーブル　　last_name：姓

```
SELECT *
  FROM People
 WHERE last_name LIKE 'Mac%';
```

だがこれは次のように書いたほうが速い。

```
SELECT *
  FROM People
 WHERE last_name LIKE 'Mac_'
    OR last_name LIKE 'Mac__'
    OR last_name LIKE 'Mac___'
    OR last_name LIKE 'Mac____';
```

切り捨てられた空白もまた、厳密に一致しうる文字である。

パターンの最初に%を使うと、処理に非常に時間がかかる。理由は単純で、SQLは木構造のインデックスを使い、そのインデックスを左から右へスキャンする。そのためこの場合は、インデックスを使うことができず、テーブルのフルキャンが選択されてしまう。たとえば、最後が「son」で終わる名前を探すことを考えよう。クエリは次のようになる。

```
SELECT *
  FROM People
 WHERE last_name LIKE '%son';
```

ここで%の代わりに_を使うと多くの実装では大きな性能改善効果がある。たいていの実装は文字列を常に左から右へパースするからだ。

```
SELECT *
  FROM People
 WHERE last_name LIKE '_son'
    OR last_name LIKE '__son '
    OR last_name LIKE '___son '
    OR last_name LIKE '____son';
```

なお、_は一致する文字を必要とするが、%はそうではないことを忘れないでほしい。たとえば、次の2つのクエリは、微妙に異なる意味になる。

```
SELECT *
  FROM People
 WHERE last_name LIKE 'John_%';

SELECT *
  FROM People
 WHERE last_name LIKE 'John%';
```

どちらも'Johnson'と'Johns'には一致する。しかし、最初のクエリは'John'は不一致とみなす。これがSQL における「1 文字以上の文字」のパターンマッチングである。<pattern>は<match value>同様、文字列連結の演算子やSUBSTRING関数およびその他の文字列関数から構成することもできる。たとえば、名前が姓の一部であるような人を見つけるクエリを考えよう。

```
SELECT *
  FROM People
 WHERE last_name LIKE '%' || first_name || '%';
```

このクエリは'John Johnson'、'Anders Anderse'、'Bob McBoble'といった人を選択するが、非常に遅い。また、大文字、小

文字は区別されるので'Jon Anjon'といった名前は選択されない。

この人を結果に含めるにはSQL文を次のように変更する。

```
SELECT *
  FROM People
 WHERE UPPER (last_name) LIKE '%' || UPPER(first_name) || '%';
```

19.2

NULL および空文字列の結果

ご想像のとおり、条件の中にNULL があるとUNKNOWN が返される。NULL はエスケープ文字、パターン、一致する値のいずれにも現れる。

もしMとPがともに長さ0の文字列だった場合、M LIKE PはTRUE になる【訳注1】。もしどちらか一方が1文字以上であれば、通常のマッチングルールが適用される。したがって、以下のクエリは、テーブルの全行を選択する。

```
SELECT *
  FROM People
 WHERE '' LIKE '';
```

19.3

LIKE は同値性を意味するのではない

よく誤解されている重要なポイントとして、2つの文字列が等しいにもかかわらず、LIKEでは不一致とみなされることがあるという事実がある。固定長文字列に対する同値性のテストでは、最初に2 つの文字列の短いほうの末尾を空白で埋め、それから比較を行う。したがって、'Smith'と'Smith '（末尾に空白）は等しい。だが、LIKE述語は空白で埋めることはしないので、両者のテストは失敗する。

この問題を回避する便利なトリックはTRIM関数を使って不要な空白をあらかじめ取り除いておくことだ。

19.4

結合ではLIKE 述語を避ける

初心者はよく「<string> IN LIKE (<pattern list>)」のような書き方を好む。LIKE 述語をOR でつなげるのはよくないと思っているようだ。この構文は間違っているが、望む結果を得るにはパターンのテーブルと結合を使う必要がある。

```
CREATE TABLE Patterns
(template VARCHAR(10) NOT NULL PRIMARY KEY);
```

Patterns：パターンテーブル　　template：テンプレート

```
INSERT INTO Patterns
VALUES ('Celko%'),
       ('Chelko%'),
       ('Cilko%'),
       ('Selko%'),
       ('Silko%');

CREATE TABLE Authors
(last_name VARCHAR(10) NOT NULL PRIMARY KEY);
```

Authors：著者テーブル　　last_name：姓

```
INSERT INTO Authors
VALUES ('Celko'),
       ('Celkos'),
       ('Selko'),
       ('Sellko');

SELECT A1.last_name
  FROM Patterns AS P1, Authors AS A1
 WHERE A1.last_name LIKE P1.template;
```

結果

```
 last_name
===========
 Celko
 Celkos
 Selko
```

だが、結合においてLIKE述語を使うのは、パフォーマンスを悪化させる可能性が高く、極力避けるべきである。

19.5

CASE 式とLIKE の検索条件

CASE式とLIKE述語を組み合わせて使うと、面白い応用が可能だ。ごく単純な例を挙げよう。文字列の中にある特定の文字列が何回現れるかを数えたい。text_col列はCHAR(25)で、この中から「term」という文字が何回現れるかを数えてみよう。

```
CREATE TABLE Foobar(text_col CHAR(25) PRIMARY KEY);

INSERT INTO Foobar VALUES
       ('term'), ('tarmtermtarm'), ('tarmtermtarmterm'),
       ('termterm'), ('tarmtarmtermtermtermtarm'),('tarmtarm');

SELECT text_col,
       CASE WHEN text_col LIKE '%term%term%term%term%term%term%'
              THEN 6
            WHEN text_col LIKE '%term%term%term%term%term%'
              THEN 5
            WHEN text_col LIKE '%term%term%term%term%'
              THEN 4
            WHEN text_col LIKE '%term%term%term%'
              THEN 3
            WHEN text_col LIKE '%term%term%'
              THEN 2
            WHEN text_col LIKE '%term%'
              THEN 1
            ELSE 0 END AS term_tally
  FROM Foobar
 WHERE text_col LIKE '%term%';
```

結果

```
 text_col                  term_tally
=====================================
 term                      1
 tarmtermtarm              1
 tarmtermtarmterm          2
 termterm                  2
 tarmtarmtermtermtermtarm  3
```

これが正しい結果を返すのはCASE 式がWHEN 句を現れる順に評価し、いったんWHEN 句が評価されるとそれ以降には進まないという短絡評価を行うからだ。なお、正確にはこのクエリは'term'が0個の場合は、そもそも結果に現れない。0個の場合も結果に出力したいなら、WHERE 句の条件を削除すればよい。

CASE式とLIKEを組み合わせたもう1つの応用方法はLIKEの中でCASE式を使うものだ。

```
name LIKE CASE WHEN language = 'English'
                 THEN 'Red%'
               WHEN language = 'French'
                 THEN 'Rouge%'
               ELSE 'R%' END
```

これによって、条件によってマッチングパターンを変えるという芸当が可能になる。

19.6

SIMILAR TO 述語

これまで見てきてわかるように、LIKE 述語は非常に貧弱だ。特にUNIX でgrepを使った経験のある人はその印象を強く持つだろう。grepは「general regular expression parser（一般正規表現パーサ）」の略だ。正規表現は形式言語の1つのクラスである。もしコンピュータ科学を専攻していれば学んだことがあるだろうが、そうでなくても心配には及ばない。ただgrepの難点はegrep()、fgrep()、xgrep()のようにいくつかのバージョンがあることだ。ISO/IEC 9945 における特殊文字を表19.1に示す。

表19.1　特殊文字

文字	内容
|	2つの候補のうち1つ。abc|klmはabcまたはklmにマッチする
*	直前の1文字の0回以上の繰り返し。ab*は、ab、abb、abbbなどとマッチする
+	直前の1文字の1回以上の繰り返し。ab+は、abb、abbbなどとマッチする
{<下限>[, <上限>]}	要素の<下限>以上<上限>以下の並びにマッチする
()	括弧内の文字を1つの単位としてグループ化する

[……]	括弧内の任意の文字。[abc]はaまたはbまたはcと一致する

表19.2は、POSIX標準のよく使われる文字集合の略称の一覧である。

表19.2　よく使われる文字集合の省略形

文字集合の省略形	内容
[:ALPHA:]	大文字小文字関係なく任意のアルファベット文字とマッチする
[:UPPER:]	任意の大文字のアルファベット
[:LOWER:]	任意の小文字のアルファベット
[:DIGIT:]	任意の数字
[:ALNUM:]	任意の数字またはアルファベット

SQL-99は、こうした特殊文字を利用できる、次のような正規表現の構文を追加した。

```
[<string expression> SIMILAR TO <pattern> ESCAPE <character>],
```

これはPOSIX標準の`grep`を元にしている。いくつか例を示そう。

1. `'foo'`または`'bar'`のあとに任意の文字列が続くパターン。

```
Foobar SIMILAR TO '(foo|bar)%'
```

2. 'SER #'のあとに1つ以上の数字が続くパターン。

```
serial_nbr SIMILAR TO 'SER #[:DIGIT:]+'
```

正規表現の詳細については、使用している製品のマニュアルを読んでほしいが、大半のgrep 関数は、SIMILAR TO述語より一般的な検索のための特殊文字を使うことができる。かつ、grep 関数は、NULLの伝播についてのルールを持っていない。基本的なトークンは表19.3に示す。

表19.3　基本的なトークン

トークン	内容
.	任意の一文字にマッチする（SQLのアンダースコアと同じ）
^	文字列の先頭にマッチする（SQLの文字列では使われない）
$	文字列の末尾にマッチする（SQLの文字列では使われない）
\	エスケープ文字。この文字の次の文字は特殊文字ではなくリテラルが来る。SQLでは、ESCAPE句で使用する文字を指定することもできる

[^]	括弧内の文字列以外の任意の文字とマッチする

正規表現は、便利な応用がいくつもあるが、特に文字列の整合性チェックに役立つ。URL、VIN（車両識別番号）、その他の標準的なエンコーディングスキーマに対する正規表現を使ったWebサイトは多く存在する。

19.7

文字列のトリック

以下では、文字列を扱うとき、普通はなかなか思いつかないようなトリックを、いくつか紹介しよう。

19.7.1　文字列の中身を調べる

可変長文字列の中身が数字だけから構成されているかどうかを調べる風変わりな方法。1999 年10月にケン・シェリダンがコンピュサーブのAccessフォーラムに投稿したものだ。もし最初の文字が0でなければ、`VARCHAR(n)`がすべて文字列かどうかは、次の式で調べられる。

```
SELECT CAST (LOG (CAST (test_column AS INTEGER)) AS INTEGER);
```

`LOG`関数は、10を底とする常用対数を求める。つまり、これは数値の「桁数−1」を求めているのだ【訳注2】。n文字がすべて数字だった場合、この関数はn−1を返す。もし文字列の中に数字でない文字が含まれていた場合、内部の`CAST` 関数が数値への変換に失敗し、エラーを返す。ゼロまたは空白によるパディングが心配な場合は文字列の前に`'1'`を結合するとよい。

もう1つのトリックは、1文字ずつ考えるのはなく、文字列全体で考える方法である。1文字ずつスキャンすることなしに、「文字列が部分的にアルファベット」、「すべてアルファベットでない」、「すべてアルファベット」であることを調べるにはどうすればよいだろう。前二者については、オセロットソフトウェア社から驚くほど簡単な答えが寄せられた。一方、「すべてアルファベット」につ

いては、SUBSTRING関数とBETWEEN述語をCHECK制約で大量に使うしかなかった。

```
CREATE TABLE Foobar
(no_alpha VARCHAR(6) NOT NULL
          CHECK (UPPER(no_alpha) = LOWER(no_alpha)),
 some_alpha VARCHAR(6) NOT NULL
          CHECK (UPPER(some_alpha) <> LOWER(some_alpha)),
 all_alpha VARCHAR(6) NOT NULL
          CHECK (UPPER(all_alpha) <> LOWER(all_alpha)
            AND SUBSTRING(LOWER(all_alpha) FROM 1 FOR 1)
                BETWEEN 'a' AND 'z'
            AND SUBSTRING(LOWER(all_alpha) FROM 2 FOR 1) || 'a'
                BETWEEN 'a' AND 'za'
            AND SUBSTRING(LOWER(all_alpha) FROM 3 FOR 1) || 'a'
                BETWEEN 'a' AND 'za'
            AND SUBSTRING(LOWER(all_alpha) FROM 4 FOR 1) || 'a'
                BETWEEN 'a' AND 'za'
            AND SUBSTRING(LOWER(all_alpha) FROM 5 FOR 1) || 'a'
                BETWEEN 'a' AND 'za'
            AND SUBSTRING(LOWER(all_alpha) FROM 6 FOR 1) || 'a'
                BETWEEN 'a' AND 'za' ));
```

no_alpha：アルファベットを含まない　　some_alpha：1文字はアルファベットを含む　　all_alpha：アルファベットだけを含む

　アルファベットの文字は大文字と小文字で異なる値とみなされるが、非アルファベットの文字にはこういう性質はない。これによって列が非アルファベット文字か部分的にアルファベットを含んでいるか、またはアルファベットだけであるかを判断することができる。

　文字列がすべてアルファベットであるかを判断する方法は、今日ならやはり正規表現を使うべきだろう。

```
all_alpha VARCHAR(6) NOT NULL
    CHECK(all_alpha SIMILAR TO '[:ALPHA]+')
```

他の2つも正規表現で表現してみてほしい。

19.7.2　文字列の検索 VS 文字列の宣言

文字列をDDLの中で宣言するときは、細心の注意を払わなくてはならない。だがそうしておけば、DMLでそれらの列を検索するときは、シンプルに考えられる。たとえば、大半のクレジットカードの番号は4 桁の数字の4 つのグループから成る。そして各グループが何らかの妥当性ルールを持っている。たとえば、次のようなイメージだ。

```
CREATE TABLE CreditCards
(card_nbr CHAR(19) NOT NULL PRIMARY KEY
     CONSTRAINT valid_card_nbr_format
       CHECK (card_nbr SIMILAR TO '[0-9]{4}-[0-9]{4}-[0-9]{4}-[0-9]{4}'),
     CONSTRAINT valid_bank_nbr
       CHECK (SUBSTRING (card_nbr FROM 1 FOR 4) IN ('2349', '2345')));
```

CreditCards：クレジットカードテーブル　　card_nbr：カード番号

```
INSERT INTO CreditCards VALUES
     ('2349-0000-0000-0000'),
     ('2349-1234-0000-0000'),
     ('2345-0000-1234-0000'),
     ('2345-0000-0000-1234');
```

このようにDDLの制約で厳密なルールを決めておけば、クレジットカードの番号が正しく保存されていることが保証されるので、検索では単純なLIKE 述語を使える。たとえば、3 つ目のグループに1234のカードを見つけるには次のように書けばよい。

```
SELECT card_nbr
  FROM CreditCards
 WHERE card_nbr LIKE '____-____-1234-____';
```

あるいは、次のように書くことすらできる。

```
SELECT card_nbr
  FROM CreditCards
 WHERE card_nbr LIKE '__________1234_____';
```

結果

```
 card_nbr
====================
 2345-0000-1234-0000
```

SIMILAR TO 述語はパターンをパースするために内部に有限状態マシンを作る。一方、LIKE におけるアンダースコア（_）はLIKE述語が列全体をパラレルでスキャンできるように最適化することができる。これが、SIMILAR TOよりもLIKEのほうがパフォーマンスが良い理由だ。

19.7.3　文字列にインデックスを作る

多くの文字列のエンコーディング技術は、同じプレフィックスを持っているが、これは我々が文字列を左から右へ読んで一番大きなカテゴリのコードを左に割り当てる傾向があるからだ。たとえば、クレジットカード番号の最初のグループの数字はカードを発行した銀行を意味する。この列に対するインデックスの構文は次のようになる。

```
-- インデックスのDDLは標準SQLには含まれていない
CREATE INDEX acct_searching
    ON CreditCards(card_nbr);
```

読者の使っている製品がインデックスの中に関数を定義することができれば、文字列の順番を逆にしたり、並べ替えたりすることでアクセス速度を速くすることもできる【訳注3】。こうした機能は実装依存ではあるが、クエリで明示的に同じ関数を使わなければならないことも多い。

もう1 つの選択肢はテーブルに、保存するデータの順序を（検索に有利な物理的順序で）並べ替えて、表示するときに本当の並び順に戻すことだ。こうすれば、ユーザーには見えないよう高速性を担保しつつ正しい表示結果を保証できる。

【訳注1】 Oracleでは長さ0の文字列をNULLとして扱うため、NULL LIKE NULLとなり、UNKNOWNが返る。したがって1行も選択されない。

【訳注2】 ある数xに対する常用対数の定義は、10^a＝xとなるような指数aである。したがって、

```
    x : a
   10 : 1
  100 : 2
 1000 : 3
...
```

という関係になる。もしテストする文字列がすべて数値ならば、常用対数によってその桁数が求められることを利用している。

【訳注3】　こうした機能の例としては、Oracleの逆キーインデックス、PostgreSQLのハッシュインデックスなどがある。もちろん実装依存である。

20

BETWEEN述語とOVERLAPS述語

BETWEEN and OVERLAPS Predicates

BETWEEN 述語とOVERLAPS 述語は、どちらも2 点で定義された線分上に3 つ目の点が含まれるか否かを調べる簡単な方法である。BETWEENは、任意のデータ型のスカラ線分に使うことができる。OVERLAPSは2つの期間（始点と終点、または始点とINTERVALで定義される）を調べて、時間的に重なりが存在するかを調べる。

20.1

BETWEEN 述語

BETWEEN 述語は、SQL の機能の中でも特に注目すべきものだ。また、少しトリッキーなところがあって、初心者が引っかかりやすい機能でもある。

```
<between predicate>
::= <row value predicand> <between predicate part 2>
<between predicate part 2> ::=
[NOT] BETWEEN [ASYMMETRIC | SYMMETRIC]
<row value predicand> AND <row value predicand>
```

SYMMETRICもASYMMETRICも指定されない場合は、暗黙にASYMMETRICとされる。これはもともとのBETWEENの定義で、多くの実装でもこのオプションで動く。これは、ANDの右側の値が、左側の値以下であることを要求する。SYMMETRIC が指定された場合は、この制限はなくなる。

SYMMETRICは、もともとMicrosoft Accessが実装した方式で、範囲の上限値と下限値を入れ替えることが許される。

いま、x、y、zをBETWEEN述語の入力として、述語の定義を詳しく見ていこう。

1. x NOT BETWEEN SYMMETRIC y AND z：以下と同値

```
NOT (x BETWEEN SYMMETRIC y AND z)
```

これは特におかしなところはない。SQLのNOT演算子の通常の動作である。

2. X BETWEEN SYMMETRIC y AND z：以下と同値

```
((x BETWEEN ASYMMETRIC y AND z)
OR (x BETWEEN ASYMMETRIC z AND y))
```

これはSYMMETRICの定義そのものである。

3. x NOT BETWEEN ASYMMETRIC y AND z：以下と同値

```
NOT (x BETWEEN ASYMMETRIC y AND z)
```

これもNOTの通常の使い方であり、特におかしなところはない。

4.x BETWEEN ASYMMETRIC y AND z：以下と同値

```
x >= y AND x <= z
```

始点と終点はともに範囲に含まれる（したがって不等号に「=」が付いている）ことに注意しよう。

BETWEEN 述語は、比較可能なデータ型であればすべてに適用することができる。このことを忘れていて、数値型にしか適用できないと勘違いしているプログラマも多いが、これは正しくない。文字列型や時間型にも同様に適用できるのだ。入力値<row value predicand>は、式または定数であるが、やはり多くのプログラマが、ここには定数か列名しか記述できないと勘違いしている。

多くの実装が、BETWEEN述語の評価の際には、いくつか特別なステップを踏む。それは、SQL はB+をはじめとして、ノードの範囲を調べることで高速な範囲検索を行えるツリーインデックスを持っているからだ。こうしたインデックスは、BETWEEN の評価に大いに役立つ。:var1 と:var2 をホスト言語変数またはリテラルとすると、

インデックスを使えば(<column name> BETWEEN :var1 AND :var2)は、ColumnName >= :var1 AND ColumnName <= :var2よりもオプティマイザにとってはずっと評価しやすい。また人間にとっても、BETWEEN述語のほうが不等号を使った定義の式よりも読みやすい。

20.1.1　入力がNULL の場合の結果

<row value predicand>にNULL が含まれた場合のBETWEEN の結果は、定義から直接導ける。つまり、3つの入力のどれか1つでもNULLであれば、結果はUNKNOWNになる。ASYMMETRIC なBETWEEN の定義「x >= y AND x <= z」からして、これは明らかなことだろう。

20.1.2　入力が空集合の場合の結果

2 つ目の<row value predicand>が3 つ目の<row value predicand>よりも大きかった場合、ASYMMETRICなBETWEENは常にFALSEになる（ただし3つ目の<row value predicand>がNULL の場合はUNKNOWN である）。結果をTRUEにするような<value expression>の値が存在しないというのは、少し直観に反するが、しかしこれは定義を展開してみれば、すぐに導ける結論だ。

```
x BETWEEN ASYMMETRIC 12 AND 15 -- 結果はxの値による
x BETWEEN ASYMMETRIC 15 AND 12 -- 常にFALSE(ただしxがNULLの場合はUNKNOWN)
x BETWEEN ASYMMETRIC NULL AND 15 -- 常にUNKNOWN
NULL BETWEEN ASYMMETRIC 12 AND 15 --常にUNKNOWN
x BETWEEN ASYMMETRIC 12 AND NULL -- 常にUNKNOWN
x BETWEEN ASYMMETRIC x AND x -- 常にTRUE(ただしxがNULLの場合はUNKNOWN)
x BETWEEN SYMMETRIC 12 AND 15 -- 結果はxの値による
x BETWEEN SYMMETRIC 15 AND 12 -- 結果はxの値による
x BETWEEN SYMMETRIC NULL AND 15 -- 常にUNKNOWN
NULL BETWEEN SYMMETRIC 12 AND 15 -- 常にUNKNOWN
x BETWEEN SYMMETRIC 12 AND NULL -- 常にUNKNOWN
x BETWEEN SYMMETRIC x AND x -- 常にTRUE(ただしxがNULLの場合はUNKNOWN)
```

20.1.3　BETWEEN を使ったプログラミングの小技

BETWEENの範囲は両端の点を含む。この点には注意する必要がある。たとえば、BETWEENを使ってテスト結果の点数の範囲を成績のコードに変換することを考えよう。

Grades

low_score	high_score	grade
90	100	A
80	90	B
70	80	C
60	70	D
0	60	E

このデータだと、生徒が境界線の点数（90、80、70、60など）をとったときに、BETWEENではうまくいかない。たとえば、90 点の生

徒の成績はA なのかBなのか決まらない。この問題を解決する1つの方法は、low_score列に1を加えて重複をなくすことだ。だがこの場合、90.1点をとった生徒は、自分にはB ではなくAを得る権利があると主張するだろう。この生徒に配慮して、low_score 列に0.01を加えるように変えたらどうだろう。今度は、90.001をとった生徒が同じことを主張するだろう。以下無限に続く。

そう、これは連続的な値に関する問題である。もっとよい解は、BETWEEN を使うのをやめて不等号に変えることだ。

```
(score BETWEEN low_score AND high_score) AND (score > low_score)
```

あるいはもっと単純に、次のようにしてもよい（ただし、これだと0 点の生徒に成績を割り当てられないので、0点がありうる場合は、下限値を−1にすべきだ）。

```
(low_score < score) AND (score <= high_score)
```

上の2つのアプローチのパフォーマンスは、このサンプルでは大差ない。境界線の狭間に落ちる値は少なく、テーブルの規模も極めて小さいからだ。

とはいえ、テーブルのサイズが大きい場合には、インデックス次第ではBETWEEN述語のほうが速いこともある。インデックスは、ツリー内の範囲を規定するノードの値を保持する（詳しくはコンピュータサイエンスの書籍でB ツリーの記述を調べてほしい）。オプティマイザは、BETWEEN述語内の値の範囲を、インデックスノード内の値の範囲と、1回の操作で比較することができる。もしBETWEEN

述語が不等号を使って2 つの比較で表現されていた場合、データベースはそれぞれ別の動作として行うため、BETWEENよりも遅くなるだろう。

　余談になるが、この例題を見てファジー論理に興味を持つ人もいるかもしれないので、入門書を1つ挙げておこう【訳注1】。

　『Fuzzy Thinking: The New Science of Fuzzy Logic』Bart Kosko 著、Flamingo、1994

　ファジー論理のモデルでは、ある要素は集合に含まれるか含まれないかが厳密には規定されず、集合へ属する度合いを想定する。趣味サークルのコアメンバーか、たまに顔を出す程度の付き合いか、といった集合をイメージすればよいだろう。ファジー論理を実装した実験的なデータベースも考えられている。

20.2

OVERLAPS 述語

OVERLAPS述語は、まだ大半の実装では利用できない機能である【訳注2】。その理由は、これを使うためには、標準SQL の時間データの機能を、現在の実装が持っているよりも多くサポートしなければならないからである。そのため多くのプログラマは、この述語をINTERVALデータ型と、製品が実装済みの日付と時刻の機能を使って代替している。

20.2.1　時間の期間とOVERLAPS 述語

時間型と時間関数は、SQL製品における最もイレギュラーな機能である。ANSI/ISO標準が規定される前、実装はてんでバラバラの方言を持っていた。しかし次第に、時間の持続期間を測るINTERVAL（日、時間、分といった単位で指定する）の概念が固まってきた。日付、時間、分から時間を足し引きするときは、このINTERVALを使う。

期間は、開始時刻と終了時刻の2点を与えることで定義される。OVERLAPS述語は2つの期間を比較する機能だ。これらの期間は、2つの列の値によって定義される。最初の列（開始時刻）は、常に<datetime>型であり、2 つ目の列（終了時刻）は<datetime>型または期間（INTERVAL）を指定できる。もし開始時刻と終了時刻が同じ場合、（期間の幅を持たない）一瞬の出来事であることを意味する。

OVERLAPS述語のBNF記法は以下のとおりである。

```
<overlaps predicate>
::= <overlaps predicate part 1> <overlaps predicate part 2>
<overlaps predicate part 1> ::= <row value predicand 1>
<overlaps predicate part 2> ::= OVERLAPS <row value predicand 2>
<row value predicand 1> ::= <row value predicand>
<row value predicand 2> ::= <row value predicand>
```

`<overlaps predicate>`の結果は、形式的には以下の式の結果として定義される。

```
(S1 > S2 AND NOT (S1 >= T2 AND T1 >= T2))
OR (S2 > S1 AND NOT (S2 >= T1 AND T2 >= T1))
OR (S1 = S2 AND (T1 <> T2 OR T1 = T2))
```

ここで、`S1`と`S2`は2 つの期間`T1`と`T2`の開始時刻と終了時刻である。`OVERLAPS` 述語のルールは直観的に理解できるべきだが、残念ながらそうはなっていない。SQL 標準の定義に望まれた原則は、次のとおりであった。

原則1

時間の期間は、始点は含むが終点は含まない。この理由は、1日の時刻に24:00は存在しないというISOの規約と合わせるためだ。00:00は翌日に属するとみなされる。半開期間は便利な閉包性を持っており、2 つの半開期間を結合した結果はやはり半開期間になる【訳注3】。

原則2

もし2つの期間が瞬間的ではなく、かつ、両者が共有する期間を持つ場合、オーバーラップ（重複）しているとみなす。

原則3

2つの期間のうち、1つがINTERVALで、もう1つが瞬間的（<datetime>データ型）であった場合、もし後者が前者の期間（ただし「原則1」により、期間の終点は除く）の中に含まれていれば、オーバーラップしているとみなす。

原則4

2つの期間がともに瞬間的な場合、2つがオーバーラップするのは、ともに等しい時刻の場合に限る。

原則5

開始時刻がNULL で、終了時刻が既知の時刻だった場合、終了時刻を開始時刻とみなす。もし開始時刻がNULLで、終了時刻がINTERVAL 値だった場合、開始時刻と終了時刻の両方をNULLとみなす【訳注4】。

これらの原則について、直観的にみてどうなるか考えてほしい。いま、期間の粒度を年月日レベルだと仮定し、具体例を見ていこう。1日は00:00時から始まる（原則1）ことから、

```
(today, today) OVERLAPS (today, today) = TRUE
(today, tomorrow) OVERLAPS (today, today) = TRUE
(today, tomorrow) OVERLAPS (tomorrow, tomorrow) = FALSE  --終点は期間に含まれない
(yesterday, today) OVERLAPS (today, tomorrow) = FALSE    --終点は期間に含まれない
```

これはまだ直観的にわかりやすいとは言いがたい。そこで、図を描いて見てみよう。何か特別なイベントのため、ホテルに宿泊客が滞在する期間を考えてほしい。そのテーブルはこういう形になるだろう。

```
CREATE TABLE Guests
(guest_name CHARACTER(30) NOT NULL PRIMARY KEY,
 arrival_date DATE NOT NULL,
 departure_date DATE NOT NULL);
```

Guests：宿泊客テーブル　guest_name：客名　arrival_date：到着日　departure_date：出発日

Guests

```
 guest_name         arrival_date    departure_date
=================================================
 'Dorothy Gale'     '2015-02-01'    '2015-11-01'
 'Indiana Jones'    '2015-02-01'    '2015-02-01'
 'Don Quixote'      '2015-01-01'    '2015-10-01'
 'James T. Kirk'    '2015-02-01'    '2015-02-28'
 'Santa Claus'      '2015-12-01'    '2015-12-25'
```

```
CREATE TABLE Celebrations
(celeb_name CHARACTER(30) PRIMARY KEY,
 celeb_start_date DATE NOT NULL,
 celeb_end_date DATE NOT NULL);
```

Celebrations：イベントテーブル　celeb_name：イベント名　celeb_start_date：イベント開始日　celeb_end_date：イベント終了日

Celebrations

```
 celeb_name             celeb_start_date   celeb_end_date
=========================================================
 'Apple Month'          '2015-02-01'       '2015-02-28'
 'Christmas Season'     '2015-12-01'       '2015-12-25'
 'Garlic Festival'      '2015-01-15'       '2015-02-15'
 'National Pear Week'   '2015-01-01'       '2015-01-07'
 'New Year's Day'       '2015-01-01'       '2015-01-01'
 'St. Fred's Day'       '2015-02-24'       '2015-02-24'
 'Year of the Prune'    '2015-01-01'       '2015-12-31'
```

特定のイベントの開始日と終了日の間に特定の日付が収まるかどうかをテストするだけなら、BETWEEN述語でも十分である。注意す

べきは、BETWEENは両端の点を含むが、OVERLAPSはそうではないことだ。到着日がイベントの開催期間と重なるかどうかを調べるには、単純に次のように書けばよい。

```
SELECT guest_name, celeb_name
  FROM Guests, Celebrations
 WHERE arrival_date BETWEEN celeb_start_date AND celeb_end_date
   AND arrival_date <> celeb_end_date;
```

これは、特定のイベントの開催中にホテルへ到着した客を見つけるという意味のクエリだ。「arrival_date <> celeb_end_date」という条件は、ANSIの規約に従えば必要だが、状況によってはないほうがビジネスルールに沿うこともあるだろう。ここからは、クエリの読みやすさのため、両端は含めるという前提で話を進める。

```
SELECT guest_name, celeb_name
  FROM Guests, Celebrations
 WHERE arrival_date BETWEEN celeb_start_date AND celeb_end_date;
```

結果

```
 guest_name        celeb_name
=======================================
 'Dorothy Gale'    'Apple Month'
 'Dorothy Gale'    'Garlic Festival'
 'Dorothy Gale'    'Year of the Prune'
 'Indiana Jones'   'Apple Month'
 'Indiana Jones'   'Garlic Festival'
 'Indiana Jones'   'Year of the Prune'
 'Don Quixote'     'National Pear Week'
 'Don Quixote'     'New Year's Day'
 'Don Quixote'     'Year of the Prune'
 'James T. Kirk'   'Apple Month'
 'James T. Kirk'   'Garlic Festival'
 'James T. Kirk'   'Year of the Prune'
 'Santa Claus'     'Christmas Season'
 'Santa Claus'     'Year of the Prune'
```

これは客の視点から、滞在中どんなイベントを体験できるかを知るには便利だが、ホテル側としては、イベントの期間中にホテルに滞在している客は誰かも知りたいだろう。これはつまり、イベント期間と客の滞在期間の2 つがオーバーラップしているかを調べることになる。OVERLAPS 述語が使える製品ならば、次のように簡単に書ける。

```
SELECT guest_name, celeb_name
  FROM Guests, Celebrations
 WHERE (arrival_date, departure_date)
         OVERLAPS (celeb_start_date, celeb_end_date);
```

だが、OVERLAPS述語が使えない製品ではどうすればよいだろう。よくある間違いは、BETWEEN 述語を次のように使ってしまうことだ。

```
SELECT guest_name, celeb_name
  FROM Guests, Celebrations
 WHERE arrival_date BETWEEN celeb_start_date AND celeb_end_date
    OR departure_date BETWEEN celeb_start_date AND celeb_end_date;
```

これは間違いである。なぜなら、このクエリは、客の滞在中に始まって終了するイベントのケースを選択できないからだ。この間違いにはまらないようにするには、4つの考えうるオーバーラップのパターンについて、図20.1のようなタイムラインのダイアグラムを描いてみることだ。

そうすると、正しくは次のように条件を追加する必要があることがわかる。

```
SELECT guest_name, celeb_name
  FROM Guests, Celebrations
 WHERE arrival_date BETWEEN celeb_start_date AND celeb_end_date
    OR departure_date BETWEEN celeb_start_date AND celeb_end_date
    OR celeb_start_date BETWEEN arrival_date AND departure_date
    OR celeb_end_date BETWEEN arrival_date AND departure_date;
```

注意深いプログラマなら、最後の条件は不要で、削除してもかまわないことに気づくだろう。だがともかく、これは正しいクエリである。ただ、最適解とは言いがたい。鍵は反対に考えること、つまり、期間がオーバーラップ「しない」ケースに着目することだ。客の滞在期間とイベント期間が重ならないケースは2つしかない。イベントが始まる前に客がチェックアウトするケースと、客が到着する前にイベントが終わるケースである。タイムラインのダイアグラムで表すと、図20.2のようになる。

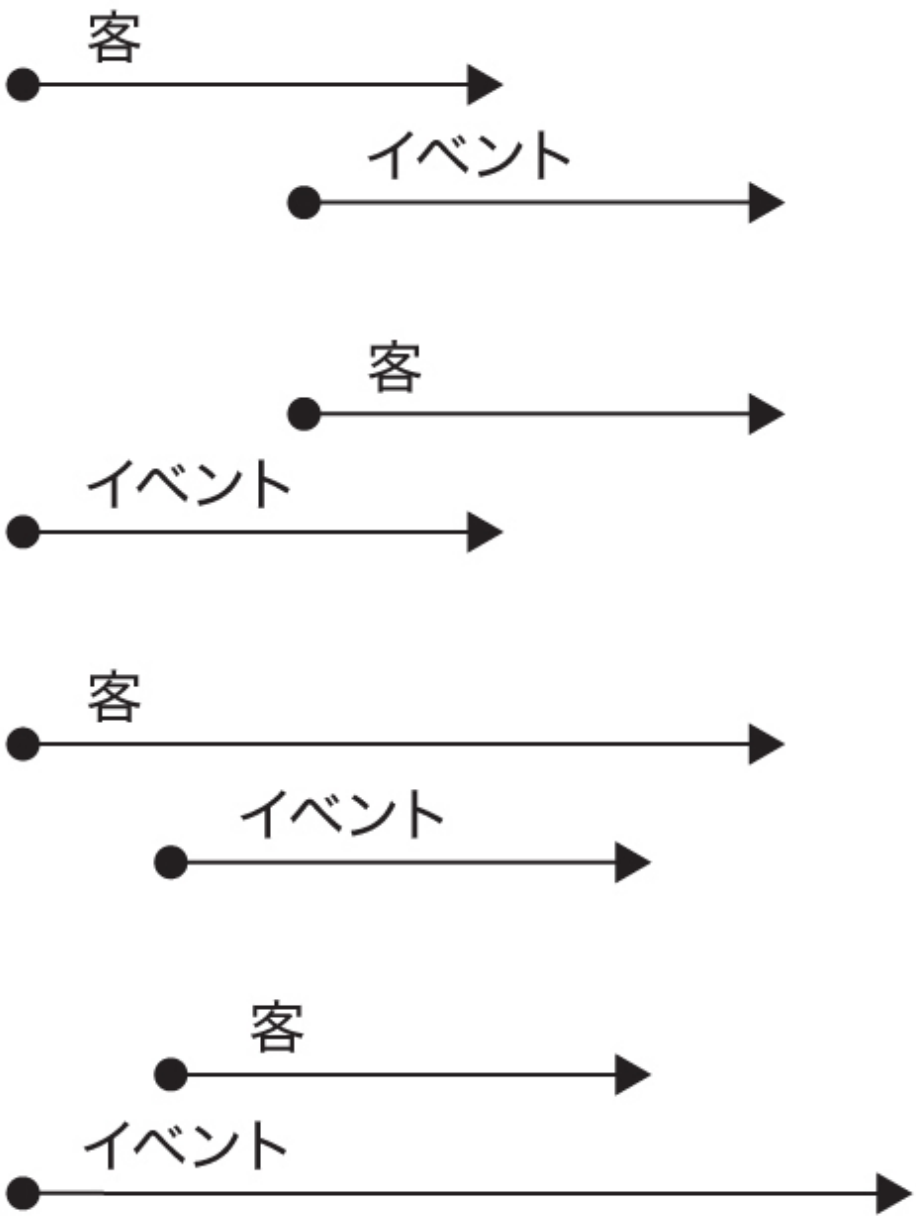


図20.1　すべての考えうるオーバーラップのタイムライン

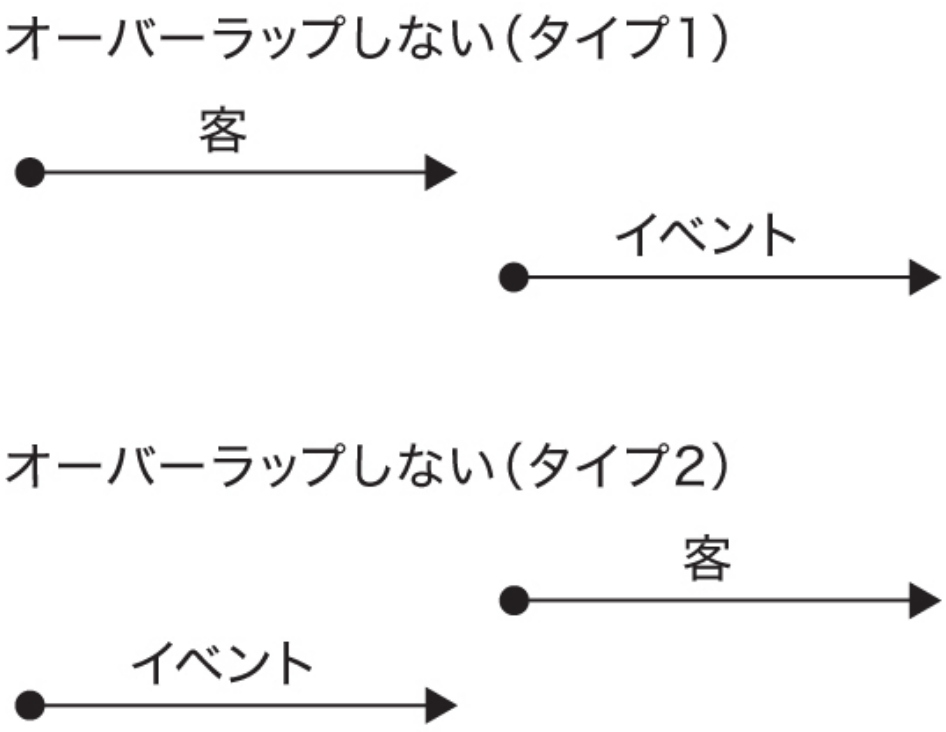


図20.2　滞在期間とイベント期間が オーバーラップしないタイムライン

この2つのケースを除外すればよいわけだから、SQL文で記述すると次のようになる。WHERE 句でNOTを使った理由は、これを付けたり削除することで、クエリの意味を簡単に反転させられるためだ。

```
SELECT guest_name, celeb_name
  FROM Guests, Celebrations
 WHERE NOT ( (departure_date < celeb_start_date)
            OR (arrival_date > celeb_end_date));
```

結果

```
 guest_name        celeb_name
=======================================
 'Dorothy Gale'    'Apple Month'
 'Dorothy Gale'    'Garlic Festival'
 'Dorothy Gale'    'St. Fred's Day'
 'Dorothy Gale'    'Year of the Prune'
 'Indiana Jones'   'Apple Month'
 'Indiana Jones'   'Garlic Festival'
 'Indiana Jones'   'Year of the Prune'
 'Don Quixote'     'Apple Month'
 'Don Quixote'     'Garlic Festival'
 'Don Quixote'     'National Pear Week'
 'Don Quixote'     'New Year's Day'
 'Don Quixote'     'St. Fred's Day'
 'Don Quixote'     'Year of the Prune'
 'James T. Kirk'   'Apple Month'
 'James T. Kirk'   'Garlic Festival'
 'James T. Kirk'   'St. Fred's Day'
 'James T. Kirk'   'Year of the Prune'
 'Santa Claus'     'Christmas Season'
 'Santa Claus'     'Year of the Prune'
```

これをビューとして保存しておくと、他の用途にも便利だ。たとえば、宿泊客が何個のイベントを見られるかを数えるときは、次のようなクエリにすればよい。

```
CREATE VIEW GuestCelebrations (guest_name, celeb_name)
AS
SELECT guest_name, celeb_name
  FROM Guests, Celebrations
 WHERE NOT ( (departure_date < celeb_start_date)
             OR (arrival_date > celeb_end_date));

SELECT guest_name, COUNT(*) AS celebcount
  FROM GuestCelebrations
 GROUP BY guest_name;
```

結果

```
 guest_name        celebcount
==============================
 'Dorothy Gale'    4
 'Indiana Jones'   3
 'Don Quixote'     6
 'James T. Kirk'   4
 'Santa Claus'     2
```

反対に、各イベントの期間中に何人の客が訪れるかを見つけるには、次のように書く。食事の量や席数を見積もるのに便利だろう。

```
SELECT celeb_name, COUNT(*) AS guest_cnt
  FROM GuestCelebrations
 GROUP BY celeb_name;
```

結果

```
 celeb_name            guest_cnt
================================
 'Apple Month'         4
 'Christmas Season'    1
 'Garlic Festival'     4
 'National Pear Week'  1
 'New Year's Day'      1
 'St. Fred's Day'      3
 'Year of the Prune'   5
```

この最後のクエリは、まだ完成形ではない。ホテルの経営者が本当に知りたいのは、イベントの期間中に稼働した部屋日数である。イベントの開催日数は、`(celeb_end_date - celeb_start_date + 1)`という式で簡単に求められる。また、客の宿泊日数も同じように、`(Guests.departure_date - Guests.arrival_date + 1)`で求められる。ここで、宿泊期間とイベント期間がオーバーラップするパターンのタイムラインをもう一度描いてみよう（図20.3）。

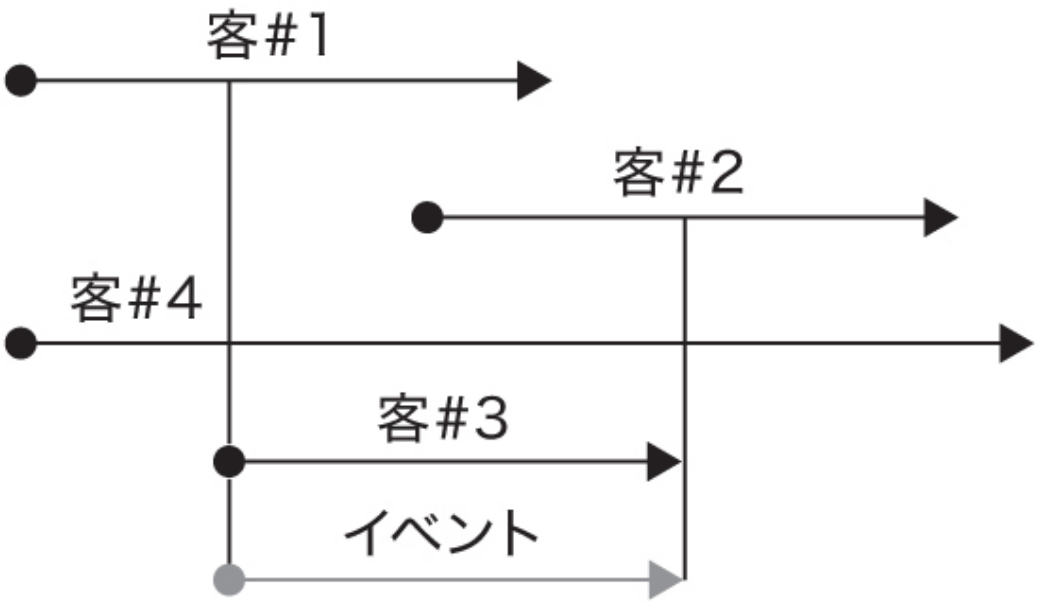


図20.3　オーバーラップするタイムライン

我々が知りたいのは、イベント期間の中に含まれる客の宿泊期間の部分だ。客#1と客#2は、イベント期間の一部しか滞在していない。一方、客#3はイベント期間中ずっと滞在し、客#4は、イベント期間よりも長く滞在している。このオーバーラップする期間は、次のような2 つの点によって定義される。

```
--始点
(CASE WHEN arrival_date > celeb_start_date
      THEN arrival_date
      ELSE celeb_start_date END)

--終点
(CASE WHEN departure_date < celeb_end_date
      THEN departure_date
      ELSE celeb_end_date END)
```

あるいは、集約関数を使って、先ほどのビューからさらにビューを作ってもよい。

```
CREATE VIEW Working (guest_name, celeb_name, entry_date, exit_date)
AS
SELECT GE.guest_name, GE.celeb_name,
       celeb_start_date, celeb_end_date
  FROM GuestCelebrations AS GE, Celebrations AS E1
 WHERE E1.celeb_name = GE.celeb_name
UNION
SELECT GE.guest_name, GE.celeb_name,
       arrival_date, departure_date
  FROM GuestCelebrations AS GE, Guests AS G1
 WHERE G1.guest_name = GE.guest_name;
```

結果

```
 guest_name        celeb_name           entry_date      exit_date
=================================================================
 'Dorothy Gale'    'Apple Month'        '2015-02-01'    '2015-02-28'
 'Dorothy Gale'    'Apple Month'        '2015-02-01'    '2015-11-01'
 'Dorothy Gale'    'Garlic Festival'    '2015-02-01'    '2015-11-01'
 'Dorothy Gale'    'Garlic Festival'    '2015-01-15'    '2015-02-15'
 'Dorothy Gale'    'St. Fred's Day'     '2015-02-01'    '2015-11-01'
 'Dorothy Gale'    'St. Fred's Day'     '2015-02-24'    '2015-02-24'
 'Dorothy Gale'    'Year of the Prune'  '2015-02-01'    '2015-11-01'
 'Dorothy Gale'    'Year of the Prune'  '2015-01-01'    '2015-12-31'
 'Indiana Jones'   'Apple Month'        '2015-02-01'    '2015-02-01'
 'Indiana Jones'   'Apple Month'        '2015-02-01'    '2015-02-28'
 'Indiana Jones'   'Garlic Festival'    '2015-02-01'    '2015-02-01'
 'Indiana Jones'   'Garlic Festival'    '2015-01-15'    '2015-02-15'
 'Indiana Jones'   'Year of the Prune'  '2015-02-01'    '2015-02-01'
 'Indiana Jones'   'Year of the Prune'  '2015-01-01'    '2015-12-31'
 'Don Quixote'     'Apple Month'        '2015-02-01'    '2015-02-28'
 'Don Quixote'     'Apple Month'        '2015-01-01'    '2015-10-01'
 'Don Quixote'     'Garlic Festival'    '2015-01-01'    '2015-10-01'
 'Don Quixote'     'Garlic Festival'    '2015-01-15'    '2015-02-15'
 'Don Quixote'     'National Pear Week' '2015-01-01'    '2015-01-07'
 'Don Quixote'     'National Pear Week' '2015-01-01'    '2015-10-01'
 'Don Quixote'     'New Year's Day'     '2015-01-01'    '2015-01-01'
 'Don Quixote'     'New Year's Day'     '2015-01-01'    '2015-10-01'
 'Don Quixote'     'St. Fred's Day'     '2015-02-24'    '2015-02-24'
 'Don Quixote'     'St. Fred's Day'     '2015-01-01'    '2015-10-01'
 'Don Quixote'     'Year of the Prune'  '2015-01-01'    '2015-12-31'
 'Don Quixote'     'Year of the Prune'  '2015-01-01'    '2015-10-01'
 'James T. Kirk'   'Apple Month'        '2015-02-01'    '2015-02-28'
 'James T. Kirk'   'Garlic Festival'    '2015-02-01'    '2015-02-28'
 'James T. Kirk'   'Garlic Festival'    '2015-01-15'    '2015-02-15'
 'James T. Kirk'   'St.Fred's Day'      '2015-02-01'    '2015-02-28'
 'James T. Kirk'   'St. Fred's Day'     '2015-02-24'    '2015-02-24'
 'James T. Kirk'   'Year of the Prune'  '2015-02-01'    '2015-02-28'
 'James T. Kirk'   'Year of the Prune'  '2015-01-01'    '2015-12-31'
 'Santa Claus'     'Christmas Season'   '2015-12-01'    '2015-12-25'
 'Santa Claus'     'Year of the Prune'  '2015-12-01'    '2015-12-25'
 'Santa Claus'     'Year of the Prune'  '2015-01-01'    '2015-12-31'
```

ここからさらに、次のビューによって、2つの期間の始点と終点のうち、最大の始点と最小の終点を1つの期間としてまとめる。

```
CREATE VIEW Attendees (guest_name, celeb_name, entry_date,exit_date) AS
SELECT guest_name, celeb_name, MAX(entry_date),
       MIN(exit_date)
  FROM Working
 GROUP BY guest_name, celeb_name;
```

結果

```
 guest_name        celeb_name            entry_date      exit_date
=========================================================================
 'Dorothy Gale'    'Apple Month'         '2015-02-01'    '2015-02-28'
 'Dorothy Gale'    'Garlic Festival'     '2015-02-01'    '2015-02-15'
 'Dorothy Gale'    'St. Fred's Day'      '2015-02-24'    '2015-02-24'
 'Dorothy Gale'    'Year of the Prune'   '2015-02-01'    '2015-11-01'
 'Indiana Jones'   'Apple Month'         '2015-02-01'    '2015-02-01'
 'Indiana Jones'   'Garlic Festival'     '2015-02-01'    '2015-02-01'
 'Indiana Jones'   'Year of the Prune'   '2015-02-01'    '2015-02-01'
 'Don Quixote'     'Apple Month'         '2015-02-01'    '2015-02-28'
 'Don Quixote'     'Garlic Festival'     '2015-01-15'    '2015-02-15'
 'Don Quixote'     'National Pear Week'  '2015-01-01'    '2015-01-07'
 'Don Quixote'     'New Year's Day'      '2015-01-01'    '2015-01-01'
 'Don Quixote'     'St. Fred's Day'      '2015-02-24'    '2015-02-24'
 'Don Quixote'     'Year of the Prune'   '2015-01-01'    '2015-10-01'
 'James T. Kirk'   'Apple Month'         '2015-02-01'    '2015-02-28'
 'James T. Kirk'   'Garlic Festival'     '2015-02-01'    '2015-02-15'
 'James T. Kirk'   'St. Fred's Day'      '2015-02-24'    '2015-02-24'
 'James T. Kirk'   'Year of the Prune'   '2015-02-01'    '2015-02-28'
 'Santa Claus'     'Christmas Season'    '2015-12-01'    '2015-12-25'
 'Santa Claus'     'Year of the Prune'   '2015-12-01'    '2015-12-25'
```

このAttendeesビューは、宿泊客とイベントで一意となり、宿泊とイベントのオーバーラップする期間だけを保持する。このビューは、イベントごとのトータルの部屋日数を計算するのにも使える。2つの日数の減算は、両者の期間の日数を整数型で返す。

```
SELECT celeb_name,
       SUM(exit_date - entry_date + 1) AS room_days
  FROM Attendees
 GROUP BY celeb_name;
```

結果

```
 celeb_name             room_days
==================================
 'Apple Month'          85
 'Christmas Season'     25
 'Garlic Festival'      63
 'National Pear Week'   7
 'New Year's Day'       1
 'St. Fred's Day'       3
 'Year of the Prune'    602
```

もし「1 月に稼働した部屋数×日数」を知りたければ、次のようなクエリが使える。これは、BETWEENもOVERLAPSも使わない。

```
SELECT SUM(CASE WHEN departure_date > CAST('2015-01-31' AS DATE)
                THEN CAST('2015-01-31' AS DATE)
                ELSE departure_date END
         - CASE WHEN arrival_date < CAST('2015-01-01' AS DATE)
                THEN CAST('2015-01-01' AS DATE)
                ELSE arrival_date END + 1)
           AS room_days
  FROM Guests
 WHERE departure_date > CAST('2015-01-01' AS DATE)
   AND arrival_date <=  CAST('2015-01-31' AS DATE);
```

【訳注1】　このサンプルでは、テストの点数は離散値なので、成績を厳密に分類できる。だが、もしテストの点数が有理数や実数などのなだらかに変化する値だった場合は、成績もよりあいまいに「Aにかなり近いB」とか「ほとんどDなんだけどギリギリC」のような意味を持ちうる。

【訳注2】　2013年現在、OVERLAPS述語をサポートしているのは、PostgreSQLのみである。Oracleでは、使用することはできるが、ドキュメントに記載がない。

【訳注3】　閉包性（closure property）は、ある演算に対する入力と出力の属するドメインが一致する性質のこと。たとえば、整数は加算、減算、乗算に対しては閉包性を持つ。整数に対するいずれの演算も、結果は必ず整数になるからである。だが、除算に対しては整数は閉包性を持たない（1/2の結果を見れば明らか）。ここでは、次のような2つの半開期間を考えて、この2つを「足した」結果がなお半開期間であるということを述べている。 a、b、cを特定の時刻とし、期間Dnを点xの動くことのできる時刻の集合とする。

期間D1　　：a <= x <b

期間D2　　：b <= x <c

このとき、

D1 + D2　　：a <= x <c

となり、「D1 + D2」も半開期間となる。点「b」でうまい具合にD1とD2が「連結」されるのがポイントである。

【訳注4】　終点がINTERVALというのは、(DATE '2013-02-16', INTERVAL '100' days) OVERLAPS (DATE '2013-10 -30', DATE '2013-10-30')のようなケースである。このように、OVERLAPS述語の2つ目の引数には、時刻だけでなく期間を指定することもできる。

21

[NOT] IN述語

The [NOT] IN() Predicate

IN 述語は、SQL で認められている「省略形」の1 つである。この述語は、普通のAND、OR、NOTといった論理演算子を使って展開できるが、プログラマにとってはINを使うほうが意味のわかりやすいコーディングができる。経験の浅いSQLプログラマは、「省略していない」書き方にこだわる傾向があるが、これは2つの理由による。

理由1　そのほうが手続き型言語に似ている

理由2　オプティマイザにとっても、省略しないロジックのほうがわかりやすいと思っている

これは両方とも完璧に間違っている。INの構文は非常に自然で、Pascalから借りてきたものだ。INは述語の左側の値を、右側の値リストと比較していく。標準SQL では、右側のリストに式やサブクエリを取ることができる。構文は以下のとおりである。

```
<in predicate> ::=
<row value predicand> = [NOT] IN <in predicate value>
<in predicate value> ::= <table subquery> | <left paren>
<in value list> <right paren>
<in value list> ::= <row value expression> [{<comma>
<row value expression>}...]
```

<row value constructor> NOT IN <in predicate value>は、NOT (<row value const ructor> IN <in predicate value>)を意味する。これはSQLにおけるNOTの普通の使い方であり、他の述語の場合と同じルールだ。

SQL:2006では、もし<in value list>が1行の<row value expression>から構成されている場合、その<row value expression>はスカラサブクエリであってはならないとされた。この構文ルールは、<in predicate value>が<table subquery>にも<scalar subquery>にも解釈できるあいまいさをなくすために導入された。

これはつまり、(<in predicate value>)は、<table value constructor>に(VALUES <in predicate value>)をとる場合と同値である。

<row value constructor> IN <in predicate value>は、<row value constructor>=ANY <in predicate value>と定義的に同じである。多くのオプティマイザはこれを認識し、どちらでも同じコードを実行する。どちらも<inpredicate value>が空ならば（たとえば、サブクエリが1行も返さない場合）、結果は、

```
(<row value constructor> = (NULL, ..., NULL))
```

と同じになる。これは常にUNKNOWNに評価される。同様に、もし<in predicate value>が明示的にNULLのリストである場合も、

結果はUNKNOWNである。だが、空のテーブルと、すべての行がNULLのテーブルは、すべての場合で同じではないことは覚えておいてほしい（たとえば、第22章のNOT EXISTSについての振る舞いを参照）。

サブクエリを伴うIN 述語はEXISTS 述語に変換可能な場合がある。しかし、この2 つの述語の間にはいくつか違うところがあり、それがやっかいな問題を引き起こす。EXISTS述語に変換するのはパフォーマンス改善のためには良い方法だが、オリジナルのIN述語に比べると読みにくいコードになる。EXISTSは述語の条件を満たす（または満たさない）値を見つけるのにインデックスを使えるためパフォーマンスが良いが、IN 述語は、サブクエリの結果を一時テーブルに保持する動作をすることが多い。だが、この動作は製品によっても違う。

21.1

IN 述語の最適化

定数リストの値の相対頻度に関する統計を持っているデータベースエンジンは少ない。そのため、基本的にはリストに現れる順にスキャンする。人間はリストをアルファベット順や数値の大小で並べることを好むが、パフォーマンスのことを考えるなら、最も頻度の高い値から滅多に現れない値の順に並べるのが望ましい。また、リストの定数に重複値を書くことは無意味である。なぜなら、その重複した値の最初のほうでマッチしたならば述語はTRUEを返し、2 つ目の重複値は見ないからである。同様に、もしその値に対して述語がFALSEを返すなら、意味なく長いリストをスキャンするだけでCPUの無駄遣いである。

多くのSQLエンジンは、サブクエリを伴うIN述語を実行する際、まずはサブクエリを実行し、一時的なワークテーブルに結果を入れる。次にそのテーブルを順次スキャンする。これは高コストな場合が多い。たとえば、こういう例を考えよう。大きなスポーツチームの本拠地の都市に住む社員を見つけたい（彼らに試合のチケットをとらせよう）。クエリは次のようになる。

```
SELECT emp_name
  FROM Personnel AS P
 WHERE P.city_name
          IN (SELECT S.city_name
                FROM SportTeams AS S);
```

都市の名前に重複はないと仮定しよう。また、社員が住んでいる都市の数はn 個、スポーツチームが存在する都市はm 個だとする（mはnよりずっと大きい）。もし一致する都市がサブクエリの作るリス

トの最初のほうに出てくるなら、最後のほうで出てくる場合に比べてクエリの実行速度は非常に速い。だが、サブクエリの結果がどのような順序で返されるかを制御する手段はない【訳注1】。

定数リストを使うならば、リストを一番ありそうな値から順に並べて書くことができる。たとえば次のようなイメージだ。

```
SELECT emp_name
  FROM Personnel
 WHERE city_name
         IN ('New York', 'Chicago', 'Atlanta', ..., 'Austin');
```

標準SQLは行式の比較を許しているので、もし標準SQLに忠実な実装を使っていれば、都市と州が別々の列であったとしても、次のように書くことができる。

```
SELECT *
  FROM Personnel
 WHERE (city_name, state_code)
           IN (SELECT city_name, state_code
                 FROM SportTeams);
```

今日では、多くの製品はサブクエリの結果テーブルから重複値を除外する。そのため、明示的にサブクエリの中で`SELECT DISTINCT`を使う必要はない。レガシーコードでは、こういう書き方をしているのを見かけることがある。

また、メジャーなSQL 製品は、リストがある程度のサイズになったところで最適化を開始する。これは人間の視点から見ると奇妙な結果をもたらすことがある。たとえば、閾値がkだとして、それよりも小さいn個のリストに対しては、nがkより大きい場合より検索に長時間かかることがあるのだ。

このときオプティマイザが使う最適化のトリックには次のようなものがある。

1. 一時テーブルにインデックスを付ける。
2. 一時テーブルに各値の頻度の隠し列を持ち、それでソートする。この頻度列の初期値は、述語の左側のテーブルの統計からセットする。
3. リストを配列上にヒープとして展開する。ヒープとは木構造の一種で、親ノードの値が子ノードの値よりも必ず大きいか同じ（または小さいか同じ）であるような木である。特に、二分木で作るヒープは二分ヒープと呼ばれ、二分探索に向いた構造だ。

IN述語を高速化するもう1つのトリックは、結合演算で置き換えることだ。たとえば、レストランの電話番号のテーブルがあるとしよう。ここからミシュランで三ツ星をとったレストランだけを選択するクエリを考える。

```
SELECT R.restaurant_name, R.phone_nbr
  FROM Restaurants AS R
 WHERE E.restaurant_name
           IN (SELECT Q.restaurant_name
                 FROM QualityGuide AS Q
                WHERE Q.michelin_stars = 3);
```

もし、michelin_starsの列にインデックスがあれば、SQLエンジンはおそらく三ツ星レストランの一時テーブルを作り、それを外側のクエリに渡すだろう。受け取った外側のクエリは、一時テーブルをあたかも定数のリストであるかのように扱う。

だが、普通はmichelin_stars列にはインデックスはないだろう。なぜなら、ミシュランの星は0 から3 までしか種類がないし、偏りも大きい（掲載レストランの8 割は星0 個だ）。もし、ミシュランの星にインデックスがなければ、エンジンは単純にQualityGuideテーブルをシーケンシャルに検索する【訳注2】。

このクエリを結合で置き換えると次のようになる。

```
SELECT R.restaurant_name, R.phone_nbr
  FROM Restaurants AS R, QualityGuide AS Q
 WHERE michelin_stars = 3
   AND R.restaurant_name = Q.restaurant_name;
```

パフォーマンスの観点から言えば、このクエリのほうが速いに決まっている。なぜなら、レストラン名はどちらのテーブルのキーでもあり、一意性を保証するためのインデックスも張られている。ただし、このクエリは重複行を返す可能性があるのでSELECT DISTINCTとする必要がある【訳注3】。

次に、もう少し財布に優しいレストランを探すとしよう。レストランのすべてのメニューを網羅したテーブルがあるとする。ここから10ドル以下のメニューを持つレストランを探そう。クエリの形は似たようなものになる。

```
SELECT R.restaurant_name, R.phone_nbr
  FROM Restaurants AS R
 WHERE R.restaurant_name
         IN (SELECT M.restaurant_name
               FROM MenuGuide AS M
              WHERE M.meal_price <= 10.00);
```

このクエリは、やはり結合を使って次のように書き換えることができる。

```
SELECT R.restaurant_name, R.phone_nbr
  FROM Restaurants AS R, MenuGuide AS M
 WHERE M.meal_price <= 10.00
   AND R.restaurant_name = M.restaurant_name;
```

このクエリは、ファストフード店のメニューをすべて返すだろう。だが、必要なのは1 行のレストラン名と電話番号なのでSELECT DISTINCTを付ける必要がある。これは重複行を排除する時間がかかるが、これがどの程度かは結果に含まれる行数と環境に依存するので、読者が各自実験してみるしかない。

NOT IN述語は、たいていはNOT EXISTS述語で置き換えられる。もう一度レストランの例を使って考えよう。我々の友人のジョンが、彼独自の名店リストを持っているとする。ここから、ガイドブックでは見つけられない店だけを見つけようと思う。自然に考えれば次のようなクエリになる。

```
SELECT J.*
  FROM JohnsBook AS J
 WHERE J.restaurant_name
         NOT IN (SELECT Q.restaurant_name
                   FROM QualityGuide AS Q);
```

だが、NOT EXISTS述語を使えば、同じ結果を得られて、しかももっと速くなる。

```
SELECT J.*
  FROM JohnsBook AS J
 WHERE NOT EXISTS
        (SELECT *
           FROM QualityGuide AS Q
          WHERE Q.restaurant_name = J.restaurant_name);
```

このバージョンが速い理由は、存在チェックのために`QualityGuide`テーブルのインデックスを使うことができるからだ。`NOT IN`バージョンではサブクエリの一テーブルの値をすべて比較しなければいけなかった。多くの実装では、`IN`述語の中で使われるサブクエリが`WHERE`句を持っていた場合、一時テーブルが作られる。だが一時テーブルはインデックスを持たない。また、重複行も存在するうえ、行の順序もランダムだ。そのため、SQL エンジンは一時テーブルをフルスキャンせざるを得ない。

21.2

ORをIN 述語で置き換える

SQLの初級者が見過ごしがちな簡単なトリックが、複数のORでつなげられた条件を1つのIN 述語で置き換えるものだ。たとえば次のようなケースだ。

```
SELECT *
  FROM QualityControlReport
 WHERE test_1 = 'passed'
    OR test_2 = 'passed'
    OR test_3 = 'passed'
    OR test_4 = 'passed';
```

これは次のように書き換えることができる。

```
SELECT *
  FROM QualityControlReport
 WHERE 'passed' IN (test_1, test_2, test_3, test_4);
```

このコードが意外に見える理由は、IN 述語の中に書けるのはサブクエリかシンプルな定数のリストだけだと思い込んでいるからだ。実際には、IN述語のリストには式を書けるのだ。オプティマイザは、ORのコードに対してはWHERE句の述語を別々に扱うが、IN述語の場合はひとまとまりにして扱う。これによって評価の順序が変わることがある。ORとINとどちらが速いか一般には言えないが、IN 述語のコードはインデックスを使えなくなる可能性がある。読者の使うDBMSでどうなるか、チェックしてほしい。

NULLとIN 述語

NULLはサブクエリを伴うNOT IN述語においていくつか特別な問題を引き起こす。次の2つのテーブルを考えよう。

```
CREATE TABLE Table1 (x INTEGER);
INSERT INTO Table1 VALUES (1), (2), (3), (4);

CREATE TABLE Table2 (x INTEGER);
INSERT INTO Table2 VALUES (1), (NULL), (2);
```

ここで次のクエリを実行する。

```
SELECT *
  FROM Table1
 WHERE x NOT IN (SELECT x FROM Table2);
```

結果はどうなるだろうか？ 答えを聞く前に、各自想像してみてほしい。実際、このクエリの評価ステップは非常に複雑だ。以下順に追ってみよう。

1. サブクエリを実行する。

```
SELECT *
  FROM Table1
 WHERE x NOT IN (1, NULL, 2);
```

2. NOT INを定義により変換する。

```
SELECT *
  FROM Table1
 WHERE NOT (x IN (1, NULL, 2));
```

3. IN述語を展開する。

```
SELECT *
  FROM Table1
 WHERE NOT ((x = 1) OR (x = NULL) OR (x = 2));
```

4. ド・モルガンの法則を適用する。

```
SELECT *
  FROM Table1
 WHERE ((x <> 1) AND (x <> NULL) AND (x <> 2));
```

5. 式を定数に評価する。

```
SELECT *
  FROM Table1
 WHERE ((x <> 1) AND UNKNOWN AND (x <> 2));
```

6. ANDの演算を定数に評価する。

```
SELECT *
  FROM Table1
 WHERE UNKNOWN または FALSE;
```

7. 結果は常に空になる。

そう、このクエリの結果は常に空、1 行も選択されないのだ。「3 と4 の2 行が返る」と予想した人もいたのではないだろうか。残念ながらその期待は裏切られる。

今度は別のテーブルの例を考えよう。次は、外側のテーブルにNULLを入れる。

```
CREATE TABLE Table3 (x INTEGER);
INSERT INTO Table3 VALUES (1), (2), (NULL), (4);

CREATE TABLE Table4 (x INTEGER);
INSERT INTO Table4 VALUES (1), (3), (2);
```

同じクエリをステップごとに評価してみよう。

1. サブクエリを評価する。

```
SELECT *
  FROM Table3
 WHERE x NOT IN (1, 3, 2);
```

2. NOT INを定義により変換する。

```
SELECT *
  FROM Table3
 WHERE NOT (x IN (1, 3, 2));
```

3. IN述語を展開する。

```
SELECT *
  FROM Table3
 WHERE NOT ((x = 1) OR (x = 3) OR (x = 2));
```

4. ド・モルガンの法則を適用する。

```
SELECT *
  FROM Table3
 WHERE ((x <> 1) AND (x <> 3) AND (x <> 2));
```

5. 結果セットを計算する。UNIONを使って表現する。

```
SELECT *
  FROM Table3
 WHERE ((1 <> 1) AND (1 <> 3) AND (1 <> 2)) -- FALSE
UNION ALL
SELECT *
  FROM Table3
 WHERE ((2 <> 1) AND (2 <> 3) AND (2 <> 2)) -- FALSE
UNION ALL
SELECT * FROM Table3
 WHERE ((CAST(NULL AS INTEGER) <> 1)
   AND (CAST(NULL AS INTEGER) <> 3)
   AND (CAST(NULL AS INTEGER) <> 2)) -- UNKNOWN
UNION ALL
SELECT *
  FROM Table3
 WHERE ((4 <> 1) AND (4 <> 3) AND (4 <> 2)); -- TRUE
```

6. 結果は1行になる＝(4)

まとめよう。サブクエリの結果にNULLが含まれる場合、NOT IN述語の結果は常に空になる。これは恐ろしいことに、SQLの正常な動作なのだ。この問題については第13章も参照してほしい。

21.4

IN 述語と参照整合性制約

IN 述語のポピュラーな使い方の1 つにテーブルのCHECK 制約の中で使うという方法がある。列に保持することのできる値をリストで表現するのだ。

```
CREATE TABLE Addresses
(addressee_name CHAR(25) NOT NULL PRIMARY KEY,
 street_loc CHAR(25) NOT NULL,
 city_name CHAR(20) NOT NULL,
 state_code CHAR(2) NOT NULL
     CONSTRAINT valid_state_code
       CHECK (state_code IN ('AL', 'AK', ...)));
```

Addresses：住所テーブル　addressee_name：住所　street_loc：番地
city_name：市町村　state_code：州コード

この方法は、値のリストが短い場合には問題ないが、リストが長くなると少々困る。この「短い」とか「長い」というのはあくまで相対的なものだ。特に昨今は、ストレージの容量も増え、安く、速くなっているので絶対的な評価はまた別の話である。リストの検索を速くするには、最も頻繁に現れる値をリストの最初のほうに配置するのがよいというのは前述のとおりである。DMLはオプティマイザの最適化の対象になるが、DDLはおそらくその恩恵を受けられない。また、DMLとDDL ではNULL の扱いが違うことも注意が必要だ。かいつまんで言うと、DDL ではUNKNOWNはTRUEとして扱われるが、DMLではFALSEとして扱われる（この点については第18章も参照）。

標準SQLでは制約において他のテーブルを参照することができる。したがって先ほどと同じ制約を次のように書くこともできる。

```
CREATE TABLE Addresses
(addressee_name CHAR(25) NOT NULL PRIMARY KEY,
 street_loc CHAR(25) NOT NULL,
 city_name CHAR(20) NOT NULL,
 state_code CHAR(2) NOT NULL,
     CONSTRAINT valid_state_code
       CHECK (state_code
               IN (SELECT state_code
                     FROM ZipCodes AS Z
                    WHERE Z.state_code = Addresses.state_code)));
```

この書き方の利点は、ZipCodesテーブルを変更すれば自動的にAddressesテーブルの制約にも変更が反映されることだ。これは動的にデータを追加変更したい場合に便利だ（たとえば、将来プエルトリコが米国に参加したあかつきには、州コードは'QB'が割り当てられる【訳注4】）。だがこの方法は、外側のテーブルからデータを削除するときには悪い影響を及ぼすこともある。たとえば、カリフォルニアが脱退すると州コードに'CA'を持つ行はすべて妥当でなくなる。また、テキサスが5 つの州に分割されて新しい郵便番号と州コードが割り振られた場合のことを考えてほしい。

経験から言って、CHECK 制約の中でIN 述語を使うのは、リストが短く静的で、外部のテーブルと連動しない場合に限るのがよい。リストが長く動的だったり、複数のテーブルで使われるようなコードは、ドメインに対するCHECK制約の中で、IN述語を使って定義するのがよい。

また、リストが長く動的で、複数のスキーマオブジェクト（ビューやストアドプロシージャなど）が値を参照する場合はREFERENCES 句を使うほうがよい。そうすれば、参照先のテーブル

がインデックスを持つことができ、検索や結合のパフォーマンスも大きく改善するだろう。

21.5

IN 述語とスカラサブクエリ

先述のように、IN述語のリストは任意のスカラ式を記述できる。この「式」にはスカラサブクエリも含まれるのだが、多くの人は、そのようなことは不可能だと思っているようだ。たとえば、倉庫とトラックの配送センターを保持するテーブルを考えよう。製品を倉庫に持っているかどうかはUPCコードによって識別できる。

```
SELECT P.upc
  FROM Picklist AS P
 WHERE P.upc
         IN ((SELECT upc FROM Warehouse AS W WHERE W.upc = P.upc),
             (SELECT upc FROM Stores AS S WHERE S.upc = P.upc),
             (SELECT upc FROM Garbage AS G WHERE G.upc = P.upc));
```

空の結果セットは、リストではNULLに変換される。このコードと同じことをしようとするには、通常は外部結合をいくつも使うか、EXISTS述語をORでつなげる。この奇妙なコードは、本書執筆時点では効率的に動かないかもしれない。だが、読者がこの本を読んでいるとき、もう一度チェックしてほしい。SQLのトレンドとして、クエリのパラレル実行がますます強化されており、個々のスカラ式を同時実行できるようになっているかもしれない。

このトリックのより便利な使い方はストアドプロシージャで長いパラメータリストを扱うときだ。与えられたパラメータがテーブルに保持しているパラメータと一致するかを調べるクエリは、最も単純なバージョンであれば、定数のリストを使って次のように書ける。

```
SELECT *
  FROM Parameter_List
 WHERE Parameter_List.i
         IN (SELECT X.i
               FROM (VALUES (1), (2), (3)) AS X(i));
```

これは、行構築子を使って次のように一般化できる。

```
SELECT Parameter_List.*
  FROM Parameter_List
 WHERE (Parameter_List.i, Parameter_List.j)
          IN (SELECT X.i, X.j
                FROM (VALUES (1, 'a'),
                             (2, 'b'),
                             (3, 'c')) AS X(i, j));
```

だが、真価が発揮されるのは入力となるパラメータのリストをとり、それをVALUES 句を使ったテーブルに変換するときだ。たとえば、GTIN（国際取引商品番号）の変数リストで商品を識別するとして、このリストに基づく結果を返すプロシージャを作ることができる。もしProducts テーブルにGTINの製品にマッチするものがなかった場合は、NULLが返される。

```
CREATE PROCEDURE Foobar
(IN in_gtin_1 CHAR(15), ... IN in_gtin_n CHAR(15))
BEGIN
SELECT << something here >>
  FROM Products AS P
 WHERE gtin
         IN (SELECT Picklist.item
               FROM (VALUES (in_gtin_1),
                            ..., (in_gtin_n)) AS Picklist(item)
              WHERE Picklist.item IS NOT NULL);
END;
```

VALUES のリストにはCOALESCE (in_gtin_n, '1234567890123456')や、適切なスカラ値を返す式を使ってもよ

い。

【訳注1】　サブクエリ内でORDER BY 句を使用できる製品もあるが、実装依存である。

【訳注2】　実際には、最近の製品は、IN述語をEXISTS述語と同じようにrestaurant_name列のインデックスを使った結合アルゴリズムで実行するものも多い。この場合の結合の方法を「半結合（semi-join）」と呼ぶ。1件でもヒットしたら残りの重複値は調べることなく検索を打ち切ることが可能（＝全件検索はしない）なことから、この名前が付いている。

【訳注3】　まさに重複行が生成されることで、パフォーマンスが劣化するケースもあるので注意が必要だ。特に、このクエリの結果をさらに別のテーブルと結合する場合は、行数が増えたことによるコスト増加を考慮しなければならない。

【訳注4】　2013年現在、プエルトリコは内政自治権を認められた未編入領域と定められている。

22

EXISTS述語

EXISTS() Predicate

EXISTS 述語の意味はとても自然だ。この述語は、集合（つまりテーブル）が空か否かのテストを行う。もしサブクエリに1 つでも行が存在すればTRUEになり、それ以外の場合はFALSEになる。この述語はUNKNOWNを返すことがない。

EXISTS述語の構文は次のとおりだ。

```
<exists predicate> ::= [NOT] EXISTS <table subquery>
```

注意すべきは、<table subquery>は常に括弧の中に書く必要があることだ。これは、パースの際に文法的な問題を避けるための約束である。

SQL-89 では、サブクエリ内のSELECT 句は、1 つの列かアスタリスク（*）を指定するルールになっていた。もし「SELECT *」が使われると、データベースエンジンは、（理論上は）1列を選んでそれを使う。このような取り決めがあったのは、SQL-89 がサブクエリは 1 つの列しか返せないと定めていたためだ。だが、今日ではこのような制限はなくなっている。サブクエリ内のSELECT句には、以下の3つの選択肢がある。

- 列　　：EXISTS(SELECT <column> ...)
- 定数　　：EXISTS(SELECT <constant> ...)
- アスタリスク ：EXISTS(SELECT * ...)

私の知る限り、この3 つの書き方の間に、結果やパフォーマンスに違いはない。現在では、EXISTS(SELECT *...)が残りの2つよりもよく使われている。この書き方をすると、問題が特定の列を選択することではなく、テーブルレベルにあることがより明確になるからだ。

EXISTS 述語は、インデックスと併用することで高いパフォーマンスを発揮する。これは、EXISTSの検索が基底テーブルに対して行われ、一時テーブルを作る必要がないからだ。たとえば、誰か有名人と同じ誕生日に生まれた社員を検索することを考える。まず、次のようなサンプルテーブルを作る。

```
CREATE TABLE Personnel
(emp_name VARCHAR(32),
 birth_date DATE,
 PRIMARY KEY (emp_name));
```

Personnel：社員テーブル　　emp_name：社員名　　birth_date：誕生日

```
INSERT INTO Personnel VALUES('Ken',   '1989-01-01'),
                            ('Smith', '1970-02-05'),
                            ('Dave',  '1965-03-10');
```

```
CREATE TABLE Celebrities
(celeb_name VARCHAR(32),
 birth_date DATE,
 birth_city_name VARCHAR(32),
 PRIMARY KEY (celeb_name));
```

Celebrities：有名人テーブル　　celeb_name：有名人名　　birth_date：誕生日
birth_city_name：出身都市

```
INSERT INTO Celebrities VALUES
       ('Lenon',   '1989-01-01', 'Tokyo'),
       ('Paul',    '1950-09-20', 'Los Angels'),
       ('Glamor',   NULL,        'New York');
```

クエリは次のような形になる。

```
SELECT P.emp_name AS famous_person_birth_date_guy
  FROM Personnel AS P
 WHERE EXISTS
        (SELECT *
           FROM Celebrities AS C
          WHERE P.birth_date = C.birth_date);
```

結果

```
 famous_person_birth_date_guy
==============================
 Ken
```

もしCelebritiesテーブルがbirth_date 列にインデックスを持っていれば、オプティマイザはそのインデックスを使って、カレント行の社員の誕生日（P.birth_date）に対する値を検索できる。もしその値がインデックスの中にあれば、TRUEを返す。このため、Celebritiesテーブルそのものを検索する必要がない【訳注1】。もしインデックスの中に値が見つからなければ、比較はFALSE

となる。この場合でもやはりCelebritiesテーブルに対するスキャンは必要ない。インデックスオンリースキャンは非常に高速である。なぜなら、インデックスはテーブルに比べると非常にサイズが小さく、かつ高速な検索が可能な構造を持っているからだ。

しかし、もしCelebrities テーブルがbirth_date 列にインデックスを持っていなければ、クエリはテーブルそのものをスキャンして、1行ずつbirth_date列を比較していくことになる。この場合でも、賢いオプティマイザならば、高速化するいくつかのトリックを持っている。たとえばSQL Serverは、インデックスに含まれない列値をインデックスのツリーに保持することのできるINCLUDEオプションを持っている。このオプションを使うことで、基底テーブルへのアクセスを省略することが可能になる【訳注2】。

22.1

EXISTSとNULL

NULLは値ではないが、SQLに存在はする。NULLの概念と振る舞いは、経験の浅いSQLプログラマにとっては大きな悩みの種だ。NULLは褐色の紙袋みたいなものだと思ってほしい。中に何かが入っていることはわかる。袋を持ち上げると、重さを感じるからだ。だが、何が入っているのか、正確には知ることができない。もし持ってみて空の袋だとわかれば、中を見る必要はない。例を挙げよう。どの有名人とも誕生日の一致しない社員を全員リストアップする。これは、上のクエリの否定形として得られる。

```
SELECT P.emp_name AS not_famous_birth_date_person
  FROM Personnel AS P
 WHERE NOT EXISTS
        (SELECT *
           FROM Celebrities AS C
          WHERE P.birth_date = C.birth_date);
```

結果

```
 not_famous_birth_date_person
==============================
 Smith
 Dave
```

Celebritiesテーブルの中には、年齢を隠すため生年月日を公表していない映画スターもいる。そういう行は('Glamor', NULL)のような形を取る。SQL 初心者は、Glamor は全員と誕生日が一致する有名人だと思うかもしれない。なぜなら、いつかはタブロイド紙が彼女の出生証明書を入手すると考えれば、現在のところ彼女と誕生日が一致する可能性を全員が持っているからだ。だが実際には、彼

女は誰とも誕生日が一致しない人間とみなされる。サブクエリが評価されるステップも追ってみれば、この動きを納得できるだろう。

```
...
WHERE NOT EXISTS
(SELECT *
   FROM Celebrities
  WHERE P.birth_date = NULL);
```

NULLに対する比較述語は常にUNKNOWNを返すので、

```
...
WHERE NOT EXISTS
(SELECT *
   FROM Celebrities
  WHERE UNKNOWN);
```

EXISTSは、サブクエリがUNKNOWNの場合FALSEを返す。NOT EXISTSはこれを反転させるので、

```
...
WHERE TRUE;
```

したがって、Ms. Glamor はNOT EXISTS において、全員をTRUEに評価してしまう。端的に言って、彼女はCelebrities テーブルに存在しないも同然なのだ。また、このサンプルは、NOT EXISTSとNOT IN の結果の違いを示す格好の例でもある。試しに、これをNOT IN を使って書き換えてみよう。

```
SELECT P.emp_name AS famous_birth_date_person
  FROM Personnel AS P
 WHERE P.birth_date NOT IN
       (SELECT C.birth_date
          FROM Celebrities AS C);
```

今度は結果はどうなるだろうか？ 前章の内容を覚えていればわかるだろう。そう、結果は空、このクエリは1行も返さないのである。Glamorの誕生日がNULLのせいで、結果が常にUNKNOWN になってしまうからだ。

次に、このクエリをもう少し複雑にしてみよう。検索対象とする有名人は、ニューヨーク出身の人物に限るという条件を付けることにする。NOT INを使ったクエリは次のようになる。

```
SELECT P.emp_name --有名なニューヨーカーと誕生日が一致しない社員
  FROM Personnel AS P
 WHERE P.birth_date NOT IN
         (SELECT C.birth_date
            FROM Celebrities AS C
           WHERE C.birth_city_name = 'New York');
```

NOT EXISTSバージョンはこうだ。

```
SELECT P.emp_name --有名なニューヨーカーと誕生日が一致しない社員
  FROM Personnel AS P
 WHERE NOT EXISTS
       (SELECT *
          FROM Celebrities AS C
         WHERE C.birth_city_name = 'New York'
           AND C.birth_date = P.birth_date);
```

さて、Glamorが唯一のニューヨーカーで、我々は相変わらず彼女の誕生日を知らないとしよう。NOT EXISTSバージョンでは、サブクエリはすべての社員に対して空となり、NOT EXISTSはTRUEを返す。したがって、全社員が返されることになる。

だがNOT INの場合、サブクエリの結果は1つのNULLであるため、述語は次のように展開される。

```
P.birth_date NOT IN (NULL)
```

これは常にUNKNOWNになり、社員は誰も選択されない。

これと同じで、一般に、量化比較述語をEXISTS述語に変換することはできない。ここでも問題はNULLである。`x <> ALL <subquery>`は、`x NOT IN <subquery>`の短縮形であり、`x = ANY <subquery>`は`x IN <subquery>`の短縮形である。それゆえ、EXISTS 述語と常に置換可能とは限らない。

一般に、EXISTS述語はIN述語よりも高速である。問題は、外側のクエリとサブクエリのどちらを先に実行するかにある。最適解はそれぞれのテーブルのサイズと値の分散によって変わるので、普通は実行してみるまでわからない。

22.2

EXISTSと内部結合

[NOT] EXISTS述語は、ほとんど必ず相関サブクエリと併用される。サブクエリは結合に展開できることが多く、そのほうがパフォーマンスが良いことも多い。先ほどのサンプルクエリを結合に変えてみよう。

```
SELECT P.emp_name AS famous_birth_date_person
  FROM Personnel AS P, Celebrities AS C
 WHERE P.birth_date = C.birth_date;
```

結合の便利なところは、結果に両方のテーブルの列を表示できることだ。したがって、次のように結果の情報量を増やすことができる。

```
SELECT P.emp_name, C.celeb_name
  FROM Personnel AS P, Celebrities AS C
 WHERE P.birth_date = C.birth_date;
```

EXISTS述語を使って、このようにSELECT句にCelebritiesテーブルの列を含めるのはできなくはないが、余計な結合を増やすことになり、端的にリソースの無駄遣いである。

```
SELECT P.emp_name, 'は', C.celeb_name, 'と誕生日が同じ'
  FROM Personnel AS P, Celebrities AS C
 WHERE EXISTS
       (SELECT *
          FROM Celebrities AS C2
         WHERE P.birth_date = C2.birth_date
           AND C.celeb_name = C2.celeb_name);
```

NOT EXISTSと外部結合

サブクエリを伴ったNOT EXISTS述語は、外部結合を使って展開することもできる。これもたいていの場合はオリジナルのクエリよりも速い。次のようなサンプルクエリを考えよう。

```
SELECT P.emp_name AS Non_famous_New_Yorker_birth_date
  FROM Personnel AS P
 WHERE NOT EXISTS
        (SELECT *
           FROM Celebrities AS C
          WHERE C.birth_city_name = 'New York'
            AND C.birth_date = P.birth_date);
```

これは次に置き換えることができる。

```
SELECT P.emp_name AS famous_New_Yorker_birth_date
  FROM Personnel AS P
         LEFT OUTER JOIN
           Celebrities AS C
    ON C.birth_city_name = 'New York'
   AND C.birth_date = P.birth_date
 WHERE C.celeb_name IS NULL;
```

この書き換えは、Celebritiesテーブルにおいて名前のわからない有名人はいないことを前提にしている。もしC.celeb_name 列がNULLを持っていた場合、もともと存在するNULLと外部結合で生成されたNULLを区別することができない。したがって、このクエリのWHERE句におけるNULL テストは、NOT NULL 制約の存在する列に対して適用する必要がある。このサンプルで言えば、そもそも名前がわからないようでは有名人とは呼べないので、C.celeb_name列は主キー（少なくともその一部）だと考えてよいだろう。

22.4

EXISTSと量化子

述語論理では、量化子という概念が存在する。量化子には2つあって、1つが全称量化子と呼ばれ「すべてのxについてP(x)」という意味を表す（Pは任意の述語）。もう1つが存在量化子で、「P(x)であるようなxが少なくとも1つ存在する」という意味を表す。前者を∀（Aを上下逆さにしたもの）、後者を∃（Eを左右逆にしたもの）という記号で表す【訳注3】。

100年以上前、論理学において大問題になったのが、存在仮定の問題だった。「すべての人間は死ぬ」が「死なない人間はいない」を包含することは、誰の目にも明らかだった。しかしこれはまた「死ぬ人間が存在する」も包含するだろうか――つまり少なくとも誰か1人は死ぬ人が存在するのだろうか？

現在ではこの議論は決着を見ており、「すべての人間は死ぬ」は「死なない人間は存在しない」と同義だが、人間の存在そのものは含意しない、とされている。SQLもこの規約に従っている。試しに、「少なくとも1人は嘘つきのセールスマンがいる」という文を考えよう。これをEXISTS 述語で表現すると、次のようになる。

```
...
EXISTS(SELECT *
         FROM Personnel AS P, Liars AS L
        WHERE P.job = 'Salesman'
          AND P.emp_name = L.emp_name);
```

セールスマンに対してもう少しシニカルな見方をするなら、「すべてのセールスマンは嘘つきだ」という定式化もできなくはない。

これをSQLで行うには、先の変換ルールをEXISTS述語に適用して、次のような二重否定の条件を記述する。

```
...
NOT EXISTS(SELECT *
             FROM Personnel AS P
            WHERE P.job = 'Salesman'
              AND P.emp_name
                    NOT IN
                      (SELECT L.emp_name
                         FROM Liars AS L));
```

このWHERE 句を日本語に直訳すれば、「嘘つきでないセールスマンはいない」という意味だ。このケースでは、IN述語は結合に置き換えることができるし、そのほうがパフォーマンスも速く、多少読みやすくなる。

22.5

EXISTSと参照整合性制約

標準SQLは、参照整合性制約を、CHECK 制約の中でEXISTS述語を使って表現することができるよう設計した。例を挙げよう。

```
CREATE TABLE Addresses
(addressee_name CHAR(25) NOT NULL PRIMARY KEY,
 street_addr CHAR(25) NOT NULL,
 city_name CHAR(20) NOT NULL,
 state_code CHAR(2) NOT NULL
     REFERENCES ZipCodes(state_code),
 ...);
```

このDDLは次のように書き換えることができる。

```
CREATE TABLE Addresses
(addressee_name CHAR(25) NOT NULL PRIMARY KEY,
 street_addr CHAR(25) NOT NULL,
 city_name CHAR(20) NOT NULL,
 state_code CHAR(2) NOT NULL,
 CONSTRAINT valid_state_code
     CHECK (EXISTS(SELECT *
                   FROM ZipCodes AS Z1
                  WHERE Z1.state_code = Addresses.state_code)),
 ...);
```

データベースは、どちらのやり方に対しても同じメカニズムを適用する。だが、DBA にとって後者を選択するメリットはない。なぜなら、CHECK制約で参照整合性制約を実装すると、カスケードの削除や更新といったアクションを定義できないからだ。

22.6

EXISTSと3 値論理

以下のサンプルは、FirstSQL 社のリー・フェスパーマンによるものだ（http://www.firstsql.com/）。これは、クリス・デイトの「業者–部品」テーブルに3行のデータを入れて使う。

```
CREATE TABLE SuppliersParts
(sup_nbr CHAR(2) NOT NULL PRIMARY KEY,
 part_nbr CHAR(2) NOT NULL,
 onhand_qty INTEGER CHECK (onhand_qty > 0));
```

SuppliersParts：業者－部品テーブル　sup_nbr：業者番号　part_nbr：部品番号
onhand_qty：在庫数

SuppliersParts

```
 sup_nbr    part_nbr    onhand_qty
==================================
 'S1'       'P'         NULL
 'S2'       'P'         200
 'S3'       'P'         1000
```

行('S1', 'P', NULL)は、業者S1は、部品Pを提供はしているが、どの程度の量を扱っているかはわからないという意味である。我々が作りたいクエリは、「部品Pを扱っているが、在庫量が1000個ではない業者を見つけること」である。正解は、在庫200個のS2である。テーブル内のすべての業者が部品Pを扱っているが、在庫数が1000 のS3は対象外である。S1は取り扱い量がわからないので、確実に条件に合致するのは、業者S2だけである。そこで、S2だけを選択するSQLを考える。単純に考えれば次のようなクエリになる。

```
SELECT *
  FROM SuppliersParts
 WHERE part_nbr = 'P'
   AND onhand_qty <> 1000;
```

このクエリは'S2'だけを返す。S3は在庫量が1000個なので明白に対象外だし、S1は在庫量が不明なので、「onhand_qty <> 1000」の結果がUNKNOWNになる。

いま、これをあえてNOT IN述語を使って同値変換すると、次のようになる。

```
SELECT SPX.sup_nbr
  FROM SuppliersParts AS SPX
 WHERE SPX.part_nbr = 'P'
   AND 1000
         NOT IN (SELECT SPY.onhand_qty
                   FROM SuppliersParts AS SPY
                  WHERE SPY.sup_nbr = SPX.sup_nbr
                    AND SPY.part_nbr = 'P');
```

このクエリはやはり'S2'だけを返す。ここまでは問題ない。問題は次だ。このクエリをNOT EXISTS述語を使って書き換えよう。

```
SELECT SPX.sup_nbr
  FROM SuppliersParts AS SPX
 WHERE SPX.part_nbr = 'P'
   AND NOT EXISTS
        (SELECT *
           FROM SuppliersParts AS SPY
          WHERE SPY.sup_nbr = SPX.sup_nbr
            AND SPY.part_nbr = 'P'
            AND SPY.onhand_qty = 1000);
```

このクエリは一見するとNOT IN 述語のクエリと同値に見えるが、('S1', 'S2')を返す。この結果を正しくないと考える人もい

るだろう。'S1'の'P'の在庫量が1000 個なのか、それより少ないのか多いのか、断定できないからである。

このような結果になる理由は、EXISTS 述語がTRUE かFALSEしか返さないからだ。たとえサブクエリがUNKNOWNを返すケース（たとえば、NULL =1000）でもだ。解決策は、サブクエリ内で量を扱う述語を、NULLに対して「疑わしきは被告人の利益」にならないことを明示するよう、修正することだ。

修正方法はいくつか考えられる。まず、標準SQLの評価述語を使って論理値を評価する方法が考えられる。

方法1 **(SPY.onhand_qty = 1000) IS NOT FALSE**

だがこれは率直に言って、読みにくいし、保守性も悪い。次に、条件となる述語を1 つ追加する方法がある。

方法2 **(SPY.onhand_qty = 1000 OR SPY.onhand_qty IS NULL)**

この場合もオプティマイザはおそらくonhand_qty 列に対するインデックスを使ってくれるだろう。あるいは次のような方法もある。

方法3 **(COALESCE(SPY.onhand_qty, 1000)= 1000)**

これは移植性が高く保守しやすい。唯一のデメリットは、onhand_qty が式の中に含まれてしまったため、インデックスが使えない実装があることだ。

本当の問題は、このクエリがNOT EXISTSと暗黙的なIS NOT FALSE条件によって、二重否定の形をしていることだ。問題の根本

原因は、EXISTS 述語がSQL の中でも少ない2 値述語であることと、(NOT (NOT UNKNOWN)) = UNKNOWNと評価されることにある。

【訳注1】 注意すべきは、EXISTSは外側のテーブルPersonnelのbirth_date列にインデックスがあったとしても、これを使用するわけではないことである。EXISTSでインデックスが使用できるのは、あくまでサブクエリ内のテーブルに対してだけだからだ。このサンプルでは、Personnelテーブルに対するWHERE句の条件がEXISTS以外にはないので、Personnelテーブルはフルスキャンが行われる。

また、著者は「Celebritiesテーブルそのものを検索する必要がない」と述べているが、これはこのクエリでbirth_date列以外にCelebritiesテーブルの列が使われていないからである。もしCelebritiesテーブルの、インデックスに存在しない列が（たとえば、サブクエリ内のWHERE句で）使われていれば、Celebrities テーブルに対するスキャンが発生することになる。

【訳注2】 INCLUDEオプションは、非クラスタ化インデックスのリーフレベルに、非キー列を追加する。詳細は以下のMicrosoft社のドキュメントを参照。

- CREATE INDEX (Transact-SQL) - SQL Server 2012 ［MSDNライブラリ］

http://msdn.microsoft.com/ja-jp/library/ms188783.aspx

【訳注3】　このような記号を使う理由は、全称量化子は英語で「for All x, P(x)」、存在量化子は「there Exists x that P(x)」という意味になるからである。

23

量化子とサブクエリ述語

Quantified Subquery Predicates

量化子は、文をTRUEにするために必要な対象の量を述べる論理演算子である。これは論理的な量であって、数学的な測度や数ではない。量化子の役割は、可能なすべての対象を文に関係付けることだ。我々は、日常生活においても、「恐竜の口臭を防げるうがい薬は1つしかない」とか「医者はみんなベンツに乗っている」とか「誰もおばあちゃんのフルーツケーキなんて食べないよ」、「牛の先物取引に投資して儲ける人もいる」といった発言をする。これらはどれも量化された表現だ。

最初の文は、一意量化子を使っている。もし恐竜の臭い息から我々を守ることのできる製品が2つ以上存在すれば、この文はFALSEになる。2番目と3番目の文は全称量化子の例だ。すべての医者について言及しているため、1人でも例外を見つければFALSEになる。同様に、おばあちゃんのフルーツケーキを食べた人間を1人でも見つければ、文はFALSEになる。最後の文は、存在量化子を使っている。これは、牛の先物で成功した人が1 人以上存在することを主張している。このような人物の1人がヒラリー・クリントンであるから、この文はTRUEである【訳注1】。

SQLの量化子は、比較述語を拡張することで結果を量化するという機能の面では、述語論理のそれと同じである。したがって、AND、OR、NOTをサブクエリの結果に適用することで表現できる。ただし、SQLの量化子は、述語論理が持っている量化子そのままではない。3値論理に適合するようカスタムされているため、戻り値がTRUEとFALSEだけにはならないのだ。

23.1

スカラサブクエリの比較

標準SQLで比較述語の対象となるのは、スカラ値と行式である。もっとも、大半のクエリではスカラ値しか使わない。もしサブクエリが1行1列の結果テーブルを返す場合、その結果はスカラ値として扱われるため、スカラ値が使える場所ではどこでも使うことができる。

たとえば、すべての生徒よりも1歳以上年上の教師をすべて見つけるクエリを考えよう。これは次のように書くことができる。

```
SELECT T1.teacher_name
  FROM Teachers AS T1
 WHERE T1.birth_date
        > (SELECT MAX(S1.birth_date) + INTERVAL '365' DAY
             FROM Students AS S1);
```

このケースでは、スカラサブクエリは一度しか実行されず、サブクエリは`Teachers`テーブルをスキャンする前に定数値に変換される。

相関サブクエリの動作はもっと複雑である。なぜなら、外側のクエリの各値につき毎回サブクエリを実行しなければならないからだ。たとえば、我々に100個未満の製品しか送ってこなかった供給業者を見つけることを考えよう。これは、次のようなクエリで可能だ。`SUM(onhand_qty)`は業者番号（`sup_nbr`）ごとに実行しなければならないことに注意してほしい。

```
SELECT sup_nbr, sup_name
  FROM Suppliers
 WHERE 100 > (SELECT SUM(onhand_qty)
                FROM Shipments
               WHERE Shipments.sup_nbr = Suppliers.sup_nbr);
```

もしスカラサブクエリがNULLを返す場合、NULLに対する比較のルールが適用される。しかし、サブクエリが空集合を返した場合はどうなるだろう——つまり供給業者が何も我々に発送していなかった場合は？ 標準SQLでは、空集合は適当なデータ型のNULLに変換される。

標準SQLでは、スカラ値または行サブクエリを、比較述語の左右どちら側でも（比較可能な結果を返す限りは）使うことができる。だが、行比較のルールには注意が必要だ。たとえば、次のクエリは、担当している製品が倉庫よりも店舗に多く置かれている製品マネージャーを見つける。

```
SELECT manager_name, product_nbr
  FROM Stores AS S1
 WHERE (SELECT SUM(onhand_qty)
          FROM Warehouses AS W1
         WHERE S1.product_nbr = W1.product_nbr)
                < (SELECT SUM(shelf_qty)
                     FROM RetailStores AS R1
                    WHERE S1.product_nbr = R1.product_nbr);
```

このようにサブクエリを比較述語の入力に取る際の大きな問題が、複数行が結果に含まれてしまう可能性があることだ。結果の一意性を保証する方法はいくつかある。集約されていないテーブルに対しては、集約関数を使って単一の値にするというのが1つ。あるい

は、外側のテーブルのキーと結合することでも一意性を担保できる。

23.2

量化子と失われたデータ

サブクエリ式を伴う量化述語は、サブクエリの返す複数の値のうちの1つを比較するために使う。その一般的な形は、次のようなものだ。

```
<value expression> <comp op> <quantifier> <subquery>
```

量化子（quantifier）はANYまたはSOMEを使う。SOMEはANYと同じであり、どちらを使うかは単に可読性の問題である。述語<value expression> <comp op>[ANY|SOME] <table exp-ression>は、<table expression>の各行（1 からn まで順番が振られているとしよう）に対して、<value expression> <comp op>の比較をORでつなげてn回繰り返すのと同値だ。

```
((<value expression> <comp op> s1)
OR (<value expression> <comp op> s2)
...
OR (<value expression> <comp op> sn))
```

ORの性質から、もし1つでもTRUEになる比較があれば（たとえUNKNOWNになる比較が存在しても）、述語全体がTRUEになる。

同様に、<value expression> <comp op> ALL <table expression>は、<table expression>の各行に対して<value expression> <comp op>の比較をAND でつなげてn 回繰り返す。

```
((<value expression> <comp op> s1)
AND (<value expression> <comp op> s2)
...
AND (<value expression> <comp op> sn))
```

AND の性質から、もし1 つでもFALSE になる比較があれば（たとえUNKNOWN になる比較が存在しても）、述語全体がFALSEになる。

ここまでは特におかしなところはない。ここで、空のテーブルmptyTableと、行は存在するが、全列がNULLのテーブルNullTableを用意しよう。

```
CREATE TABLE EmptyTable (a INTEGER, b INTEGER);

CREATE TABLE NullTable (a INTEGER, b INTEGER);
INSERT INTO NullTable VALUES(NULL, NULL);
INSERT INTO NullTable VALUES(NULL, NULL);
INSERT INTO NullTable VALUES(NULL, NULL);
```

SQLのルールに従えば、<value expression> <comp op> ALL NullTableは常にUNKNOWN を返す。同様に、<value expression> <comp op> ANY NullTable も常にUNKNOWN を返す。したがって、次のクエリはどちらもUNKNOWNを返す。

```
SELECT (1 = ALL (SELECT a FROM NullTable)) AS truth_val;
```

結果

```
 truth_val
=============
 UNKNOWN
```

```
SELECT (1 = ANY (SELECT a FROM NullTable)) AS truth_val;
```

結果

```
 truth_val
=============
 UNKNOWN
```

展開した比較述語がすべてUNKNOWN を返すわけだから、これは当然である。このことから、ORとANDをつなげた場合の演算は、通常のルールに従うことがわかる。ところが、<value exp ression> <comp op> ALL EmptyTable は常にTRUE を、<value expression> <comp op>ANY EmptyTableは常にFALSEを返すのである。

```
SELECT (1 = ALL (SELECT a FROM EmptyTable)) AS truth_val;
```

結果

```
 truth_val
=============
 TRUE
```

```
SELECT (1 = ANY (SELECT a FROM EmptyTable)) AS truth_val;
```

結果

```
 truth_val
=============
 FALSE
```

ANY述語がこのような動作をする理由はわかる。1つも一致しないのだから、結果はFALSEで当然だ。しかし、ALL 述語がTRUEを返す

のは理解に苦しむ。これは前章でEXISTS述語を説明したときにも少し触れた、存在仮定の原理に関する問題だ。存在仮定は、現代論理学の黎明期に「すべてのxはyである」という文が「少なくとも1つのxが存在する」を定義上含むかどうかで、激しい論争になったテーマだ。だが、現在の論理学の規則では、これは否定されている。たとえば、私がバーに入っていって「オレはどんなピンクの象でもぶちのめせるぞ」とわめき散らしたとしよう。そして、この文の真理値がTRUE であるとしよう。このとき、私の発言から「バーにピンクの象が（少なくとも1匹）いる」という事実を含意してはいない。バーにピンクの象がいない場合は、単に問題の極小化されたパターン（minimum case）であるにすぎない【訳注2】。

もしこの例が不自然だと思うなら、ALLとANYをEXISTS述語で書き換えてみれば、現代論理学の規則が形式的な数学上の性質を保存するものであることがわかる。

1. ∀xP(x) = ¬∃¬xP(x)
2. ∃xP(x) = ¬∀¬xP(x)

`Table1.x <comp op> ALL (SELECT y FROM Table2 WHERE <search condition>)`は、次に変換できる。

```
... NOT EXISTS
(SELECT *
   FROM Table1, Table2
  WHERE Table1.x <comp op> Table2.y
    AND NOT <search condition>) ...
```

一方、Table1.x <comp op> ANY (SELECT y FROM Table2 WHERE <search condition>)は、次に変換できる。

```
... EXISTS
(SELECT *
   FROM Table1, Table2
  WHERE Table1.x <comp op> Table2.y
    AND <search condition>) ...
```

2 つの量化述語のうち、よく使うのはALL 述語のほうだ。ANY 述語は、EXISTS述語かIN 述語で書いたほうがより自然で簡単である。実際、標準SQLではIN 述語は「= ANY」の、NOT IN 述語は「<> ANY」の省略形として定義されている。これは普通に英語や日本語などの自然言語で考えても納得できるだろう。

ANY述語よりALL述語のほうが便利だが、その理由は、IN述語を使って書くことができないからだ。この述語のトリックは、サブクエリがある特定の値の条件を満たすことを保証するクエリを書く場合に便利だ。たとえば、書いたすべての本が49.95ドル以上で売られている著者をすべて見つけるクエリは次のように書く。

```
SELECT *
  FROM Authors AS A1
 WHERE 49.95
        <= ALL (SELECT book_price
                  FROM Books AS B1
                 WHERE A1.author_name = B1.author_name);
```

このクエリを解釈するコツは「条件yを満たすようなすべてのxを見せろ」という自然言語の文を頭の中で「条件yがすべてのxの値である」と読み変えることだ。

23.3

ALL 述語と極値関数

次の2つの述語が同値ではないというのは最初は直観に反するだろう【訳注3】。

```
x >= (SELECT MAX(y) FROM Table1)
x >= ALL (SELECT y FROM Table1)
```

だが、計算の前にNULLをすべて除外するという極値関数のルールを覚えていれば、2 つの違いがわかる。ALL述語はNULLを除外しないので結果の中にも含むのだ。もしも、列の中にNULL がないことがわかっている、または明示的にNULLを除外している場合は、この2つのクエリは問題なく同値になる。たとえば、製品と店長のテーブルを一度考えよう。最も多くの製品を扱う店長を見つけたい。おそらく最初の考えは、極値関数の共通表式を作り、2段階の集約を行うというものだろう。

```
CREATE TABLE Stores
(manager_name CHAR(16),
 product_id   CHAR(4),
     PRIMARY KEY(manager_name, product_id));

INSERT INTO Stores VALUES
     ('Adam',  '0001'),
     ('Adam',  '0002'),
     ('Adam',  '0003'),
     ('Barray','0002'),
     ('Curt',  '0001'),
     ('Curt',  '0002'),
     ('Dave',  '0001'),
     ('Dave',  '0002'),
     ('Dave',  '0003');

WITH TotalProducts (manager_name, product_tally)
AS
(SELECT manager_name, COUNT(*)
   FROM Stores
  GROUP BY manager_name)
SELECT manager_name
  FROM TotalProducts
 WHERE product_tally
         = (SELECT MAX(product_tally)
              FROM TotalProducts);
```

結果

```
 manager_name    count
=======================
 Adam            3
 Dave            3
```

しかし、アレックス・ドーフマンは1つのサブクエリで解く方法を見つけた。

```
SELECT manager_name, COUNT(*)
  FROM Stores
 GROUP BY manager_name
HAVING COUNT(*) + 1
         > ALL (SELECT DISTINCT COUNT(*)
                  FROM Stores
                 GROUP BY manager_name);
```

サブクエリの中で`SELECT DISTINCT`を使っているのは、2 人のマネージャーが同数の製品を扱っている場合に重複行を返さないためだ。

23.4

UNIQUE 述語

UNIQUE 述語は、サブクエリに重複行が存在しないことをテストするためのものだ。ユニークという予約語はテーブルや列の制約でも使われるが、この述語はサブクエリ式に対する制約を定義するために使う。構文は次のとおりだ。

```
<unique predicate> ::= UNIQUE <table subquery>
```

もし、サブクエリ内の2 つの行が一致すればこの述語はFALSEを返す。ただし、NULLに対しては「疑わしきは罰せず」の原理が適用される。この述語は、EXISTS述語を使って行数を数える次のようなクエリで書き換えることもできる。

```
EXISTS (SELECT <column list>
          FROM <subquery>
         WHERE (<column list>) IS NOT NULL
         GROUP BY <column list>
        HAVING COUNT(*) > 1);
```

サブクエリが空の場合は常にTRUEである。そもそも2行存在しないのだから重複行も存在するはずがない。これはそのままずばりである。

NULLに対するこの述語の動作は、例を使ってみたほうがわかりやすい。いま2 つの行しか持たないテーブルを考えよう。その2行は次のようなものだ。

```
('a', 'b')
('a', NULL)
```

2 行とも1列目は非NULLで同値だ。したがって、ここまでは重複する。2 列目は2 行目のほうがNULL である。したがってどのような値とも比較することができず2 列目はスキップする。その結果、最終的にTRUEと評価される。2行目のNULLは、もしかしたらいつの日か'b'になるかもしれないが、いまはまだそうではない。これがNULLに対する「疑わしきは罰せず」の原理である。

さて今度は、サブクエリが次のような2つの行を持っていた場合を考えよう。

```
('a', NULL)
('a', NULL)
```

この場合もやはり、UNIQUE述語はTRUEになる。なぜならNULL同士は互いに等しいとも等しくないとも言えないからだ。こうした面倒な考慮を避けるためにも、DDLにおいてNULLを最初から除外しておくべきだろう。

列に対する一意制約は、多くの製品では内部的に「CREATE UNIQUE INDEX <index name>ON <table>(<column list>)」文を実行することで実装されている。標準SQLは、インデックスその他あらゆる物理的なデータアクセス方法を定義していないが、ベンダーコンソーシアムはこの基本的な構文について合意している。

一意性をチェックするもう1つの方法は、ハッシュである。もしハッシュの衝突が起こった場合、値が同一でないかどうか調べることになる。データベースに対する最小完全ハッシュについては、多く

の研究が行われている【訳注4】。この技術は大規模データベースに対してはインデックスよりもずっと速い。なぜなら、インデックスが複数回の探索を必要とするのに対しハッシュは原則的に1回の計算しか必要としないからだ。

【訳注1】　ヒラリー・クリントンは1978～79年頃、牛の先物取引で1000ドルの初期投資額を1年未満で10万ドルに増やしたと言われる。夫のビルが大統領在職中の1994年に、非合法に多額の収益を得たのではないかという嫌疑をかけられて、異例の記者会見を開いたことがある。

【訳注2】　この問題は難しいが、次のような「帰納的」な考え方をすると感覚的にわかりやすいかもしれない。
いま、「オレはどんなピンクの象でもぶちのめせるぞ」という文をPとする。文PはTRUEであると前提する。もしバーに3匹のピンクの象が存在したとすれば、文Pは前提よりTRUE である。もしバーに2匹のピンクの象が存在したとすれば？ やはり前提より文PはTRUEである。2匹のピンクの象をぶちのめすのは、3匹のピンクの象をぶちのめすよりも簡単な仕事だろう。もしバーに1匹のピンクの象が存在した場合は？ この仕事はもっと簡単であり、文PはTRUEだ。それでは、バーに0匹のピンクの象が存在したとすれば？ 0匹のピンクの象をぶちのめすのは、1匹のピンクの象をぶちのめすより、さらに簡単な仕事であり、やはり文PはTRUE だ。0匹の場合が「問題の極小化されたパターン」とは、そのよ

うな意味だと思われる。

なお、「ピンクの象が見える」という英語表現は、アルコールや麻薬による幻覚症状の婉曲表現である。

【訳注3】 MAX関数を使った文を日本語で表現すれば「彼はクラスで一番背が高い」となり、ALL述語を使えば「彼はクラスの誰よりも背が高い」となる。これは同値のように見えるだろうが、それは2 値論理の世界の話であり、3 値論理においては同値にならない。この問題については、拙著『達人に学ぶSQL 徹底指南書』の第3 章で詳しく解説しているので、興味のある方は参照してほしい。

【訳注4】 完全ハッシュとは、出力値の衝突が発生しない（つまり単射）のハッシュ、最小完全ハッシュとは、n 個の入力に対して出力もn個の連続的な値になる（つまり歯抜けのない）完全ハッシュのこと。

24

単純なSELECT 文

The Simple SELECT Statement

SQLには良いところと悪いところがある。良いところは、プログラマはSELECT文を覚えたらほとんどすべてのことができてしまうことだ。悪いところは、SELECT 文は非常に多くの入れ子の句を持つことができて、さながらビクトリア朝時代の小説のような複雑さを持つことだ。

SELECT文はデータベースに問い合わせるために使う。1つ以上のテーブルを結合し、ある程度の計算を行い最終結果を別のテーブルに入れるか、ホスト言語に渡す。本書では、入門書で見るような、1テーブルしか使わない単純なSELECT 文をあまり扱ってこなかった。読者はすでに経験を積んだSQLプログラマであり、そうしたクエリは十分勉強してきたものと仮定している。だがここで、もう一度基礎に戻って、SQL で使われている概念モデルをおさらいしても無駄にはならないだろう。シンプルなクエリは、より複雑なクエリの基礎となっているのだから。

24.1

SELECT 文の実行順序

SELECT文の句が実行される順序は実質的に決まっているのだが、初級者向けの本ではあまり触れられていないようだ。だがこの順序を理解すると、SQLでできることと、できないことの違い、およびその理由がわかる。

24.2

1レベルのSELECT 文

考えうる最も単純なSELECT文は、ただ「SELECT * FROM Sometable;」と書いたものだ。これはテーブル全体を返す。構文的に最低限必要とされるのはSELECT 句だけだが、実際にはほぼ必ずWHERE句も使う。

単純なSELECT文を詳しく見ていこう。構文は次のとおりだ。

```
[WITH <common table expression list>]
SELECT [ALL | DISTINCT] <scalar expression list>
FROM <table expression>
[WHERE <search condition>]
[GROUP BY <grouping column list>]
[HAVING <group condition>];
```

実質的な実行順序は次のとおりだ。SQLのコンパイラが必ずこの順序に忠実に従うわけではないが、最終的な結果は、この順序で実行した場合と同じでなければならない。

1. クエリ本体が使うことができるように[WITH <common table expression list>]を実行する。もっとも、実際にはSQLエンジンは、多くの場合、この共通表式のテキストをインラインコードとして展開している。テーブル式は書かれた順序で実体化されていくので、カレントのリストメンバーにおいて前の式を参照することができる。いまはこの機能には立ち入らず、基本に忠実にいくとしよう。
2. FROM <table expression>句を実行し、一時的な結果テーブルを作る。この一時テーブルはSELECT文のスコープ内でのみ存

在する。FROM句にはあらゆる種類のテーブル式を記述できるが、重要なポイントは最終的には結果として1つのワークテーブルを返すことだ。ここに記述できる式の詳細はあとで見ていくことにする。特に結合演算子、導出テーブル、共通表式、相関サブクエリなどが重要である。

概念的にはテーブル名や式の単純なリストから暗黙のクロス結合が行われるが、実際には中間テーブルを作るためにクロス結合を行う製品は存在しない。このやり方だと中間テーブルがあまりに大きくなってしまうからだ。たとえばFROM 句に1, 000行のテーブルが2つ記述されていた場合、それをクロス結合すると結果は1, 000, 000 行のテーブルになる。クロス結合はあくまでFROM句の動作を説明するための方便にすぎない。

3. もしWHERE 句があれば、そこに記述された検索条件は、 FROM 句の結果テーブルに対する各行に適用される。そして、TRUEと評価された行だけが残る。FALSE やUNKNOWN の行はワークテーブルから削除される。

   このテーブルは、導出元テーブルのテーブル名を各列に維持しているという点で、他のテーブルとは異なる。たとえば、もしテーブルAとテーブルBがどちらもxという名前の列を持っていた場合、FROM句の結果にはA.xとB.xが存在することになる。このため、列名を指定するときは混乱を避けるために、テーブル名まで指定して使う習慣を持つとよい。あるいは列名をAS 演算子で名前を変えてもよい。そうすれば、明示的に付与した名前を使うことで混乱を避けられる。

WHERE 句はいわば、テーブルから行を削除していく場所だ。単純な検索条件やサブクエリを入れ子にした複雑な条件も書ける。サブクエリの構文は括弧の中にSELECT文を入れる形を取る。対応する括弧を書き忘れるのは、初心者がよくやる間違いなので注意しよう。このサブクエリは、SQLがそのオリジナルの名前「Structured English Query Language」の名前の元になった機能だ。つまり、SELECT 文を入れ子にできる機能こそ「structured」と呼ばれた由来なのだ。サブクエリについては他の章で扱うので、ここでは単純なWHERE句の条件を中心に見ていこう。最も単純なWHERE句は2 種類の検索条件からなる。WHERE 句はワークテーブルの中の1 つのテーブルから行を除外する検索変数（search argument）を持つ（これをSARGと呼ぶ）。SARGは通常、単純な比較述語、関数呼び出し、あるいは同じテーブルのほかの列への参照などからなる。2 つ目の種類の検索条件は結合述語である。この述語は2つ以上のテーブルに対する参照を含む。

4. もしGROUP BY 句があれば次にこれが実行される。GROUP BY 句はFROM 句とWHERE 句の作ったワークテーブルに対して作用し、それに含まれる行をグループに分割していく。そのとき、分割のキーに使われるのが<grouping column list>の列である。グルーピングは同値性のテストとは異なる。なぜなら、NULL同士は互いに等しいものとして扱われるからだ。したがって、NULLはそれで1つのグループを作る。グルーピングが終わったら次に各グループは一行に集約され、新しい結果テーブル

が作られる。ここで古いワークテーブルは捨てられる。この新しいテーブルに存在しうる要素は以下の4つである。

- グルーピングのキー
- 集約関数
- 関数呼び出しまたは定数
- 上記3つから作ることのできる式

グループ化されたテーブルの各行は、そのグループに対する情報を表している。すなわちこれは、オリジナルのテーブルよりも1レベル高い情報である。現在の標準SQLでは、集約関数を複数のレベルで使用することができる。また同じことをUNIONとテーブル式を使って同じ結果を得る方法もある。これについてはあとで見る。

5. もしHAVING 句があれば、その時点のワークテーブルのグループ化された各行に対して適用される。ここでTRUEに評価された行だけが残る。FALSEやUNKNOWNの行は削除される。GROUP BY句がない場合、HAVING句はテーブル全体を1つのグループとして扱う。HAVING句を使うにはGROUP BY句が必要だというのは迷信である。

普通、HAVING 句で使われた集約関数はSELECT 句にも現れるが、必ずそうしなければいけないわけでもない。それどころか、SELECT句はグルーピングのキーをまったく含まなくても、構文的には問題ない（それで役に立つ結果かというのはまた別問題だ）。

6. 最後にSELECT 句が結果テーブルに適用される。SELECT 句の<expression list>に現れない列は最終的な結果から削除される。ここに含まれる式は定数でも列名でもよいし、定数、列、関数、あるいはスカラサブクエリを使った計算でもかまわない。

　もしSELECT句がDISTINCTオプションを持っていれば、最終的な結果テーブルから余分な重複行が削除される。この「余分な」の意味は、1つは残されるということである。もしSELECT句がALLオプションを持っていたり、あるいはALLもDISTINCTもなかった場合は、すべての重複行が最終的な結果テーブルに保存される。正直な話、いかに適法な構文とはいえ、ALLオプションを使う人などいないだろう。これでようやく、結果が返される。

　SELECT句が最後に実行されるパートであるということは、つまり、SELECT句に存在するスカラサブクエリ、関数呼び出し、および式は、他のすべての句が実行されたあとに実行されるということである。SELECT 句の式に対するAS 演算子は、列に別名を与える。この新しい名前はすぐに存在を始めるが、しかしWHERE句もGROUP BY句もHAVING句も実行されたあとの話なので、この別名をその他の句で使うことはできない【訳注1】。このように、SQLはシーケンシャルファイルや手続き型言語のように左から右へ実行されるわけではない。人間が読む順序とコンピュータが実行する順序が違うのだ。

　手続き型言語では次のような2つの文は結果が異なる。

```
READ (a, b, c) FROM File_X;
READ (c, a, b) FROM File_X;
```

一方、次のような2つのSELECT文は同じ結果を返す。ただ列の順序が違うだけなのだ。

```
SELECT a, b, c FROM Table_X;
SELECT c, a, b FROM Table_X;
```

なお、SELECT句で付けた別名を、同じレベルのSELECT句で使うことはできない【訳注2】。次のような、紛らわしい書き方は構文違反である。

```
SELECT f(c2) AS c1, c1 + c2 AS c3 FROM Foobar;
```

さて、それでは実際のサンプルデータを使って、SELECT文の動作を追ってみよう。学生と成績の2つのテーブルを結合するクエリを考える。

```
SELECT S.sex_code, COUNT(*), AVG(S.stud_age),
       (MAX(S.stud_age) - MIN(S.stud_age)) AS age_range
  FROM Students AS S, Gradebook AS G
 WHERE grade = 'A'
   AND S.stud_nbr = G.stud_nbr
 GROUP BY S.sex_code
HAVING COUNT(*) > 3;
```

2つのテーブルは次のようなデータを持っている。

```
CREATE TABLE Students
(stud_nbr INTEGER NOT NULL PRIMARY KEY,
 stud_name CHAR(50) NOT NULL,
 sex_code INTEGER NOT NULL
     CHECK (sex_code IN (0, 1, 2)), --ISOコード
 stud_age INTEGER NOT NULL);
```

Students：学生　　stud_nbr：学生番号　　stud_name：学生名　　sex_code：性別コード　　stud_age：年齢

Students

```
 stud_nbr    stud_name    sex_code    stud_age
=============================================
 1           Smith        1           16
 2           Smyth        2           17
 3           Smoot        2           16
 4           Adams        2           17
 5           Jones        1           16
 6           Celko        1           17
 7           Vennor       2           16
 8           Murray       1           18
```

```
CREATE TABLE Gradebook
(stud_nbr INTEGER NOT NULL PRIMARY KEY
     REFERENCES Students(stud_nbr)
       ON DELETE CASCADE
       ON UPDATE CASCADE,
 grade CHAR(1) NOT NULL
     CHECK (grade IN ('A', 'B', 'C', 'D', 'F')));
```

Gradebook：成績表　　stud_nbr：学生番号　　grade：成績

Gradebook

```
 stud_nbr    grade
==================
 1           A
 2           B
 3           C
 4           D
 5           A
 6           A
 7           A
 8           A
```

まず、FROM句でクロス結合（CROSS JOIN）を行うと次のようなワークテーブルが作られる。

クロス結合で作ったワークテーブル

Student				Gradebook	
stud_nbr	stud_name	sex_code	stud_age	stud_nbr	grade
1	Smith	1	16	1	A
1	Smith	1	16	2	B
1	Smith	1	16	3	C
1	Smith	1	16	4	D
1	Smith	1	16	5	A
1	Smith	1	16	6	A
1	Smith	1	16	7	A
1	Smith	1	16	8	A
2	Smyth	2	17	1	A

2	Smyth	2	17	2	B
2	Smyth	2	17	3	C
2	Smyth	2	17	4	D
2	Smyth	2	17	5	A
2	Smyth	2	17	6	A
2	Smyth	2	17	7	A
2	Smyth	2	17	8	A
3	Smoot	2	16	1	A
3	Smoot	2	16	2	B
3	Smoot	2	16	3	C
3	Smoot	2	16	4	D
3	Smoot	2	16	5	A
3	Smoot	2	16	6	A
3	Smoot	2	16	7	A
3	Smoot	2	16	8	A
4	Adams	2	17	1	A
4	Adams	2	17	2	B
4	Adams	2	17	3	C
4	Adams	2	17	5	A

4	Adams	2	17	6	A
4	Adams	2	17	7	A
4	Adams	2	17	8	A
5	Jones	1	16	1	A
5	Jones	1	16	2	B
5	Jones	1	16	3	C
5	Jones	1	16	4	D
5	Jones	1	16	5	A
5	Jones	1	16	6	A
5	Jones	1	16	7	A
5	Jones	1	16	8	A
6	Celko	1	17	1	A
6	Celko	1	17	2	B
6	Celko	1	17	3	C
6	Celko	1	17	4	D
6	Celko	1	17	5	A
6	Celko	1	17	6	A
6	Celko	1	17	7	A
6	Celko	1	17	8	A

7	Vennor	2	16	1	A
7	Vennor	2	16	2	B
7	Vennor	2	16	3	C
7	Vennor	2	16	4	D
7	Vennor	2	16	5	A
7	Vennor	2	16	6	A
7	Vennor	2	16	7	A
7	Vennor	2	16	8	A
8	Murray	1	18	1	A
8	Murray	1	18	2	B
8	Murray	1	18	3	C
8	Murray	1	18	4	D
8	Murray	1	18	5	A
8	Murray	1	18	6	A
8	Murray	1	18	7	A
8	Murray	1	18	8	A

次にWHERE句を適用する。WHERE句には2つの検索条件がある。最初の述語「grade ='A'」はGradebookテーブルしか必要としない。したがってこれはSARG（p.409参照）だ（実際にはオプティマ

イザはクロス結合を行う前にこの条件に合わない行を早々と削除してしまう）。2 番目の検索条件は「S.stud_nbr=G.stud_nbr」である。これは両方のテーブルを使う必要がある。したがってこれは結合条件だ。それではこの2 つの条件に合致しない行を削除しよう。結果テーブルは次のようになる。

WHERE句の適用後　(WHERE grade = 'A')

Student				Gradebook	
stud_nbr	stud_name	sex_code	stud_age	stud_nbr	grade
1	Smith	1	16	1	A
5	Jones	1	16	5	A
6	Celko	1	17	6	A
7	Vennor	2	16	7	A
8	Murray	1	18	8	A

次にGROUP BY句を適用する。sexコードでワークテーブルをグルーピングしよう。

GROUP BY句の適用後　(GROUP BY S.sex_code)

Student				Gradebook		
stud_nbr	stud_name	sex_code	stud_age	stud_nbr	grade	
1	Smith	1	16	1	A	sex_code=1
5	Jones	1	16	5	A	
6	Celko	1	17	6	A	
8	Murray	1	18	8	A	
7	Vennor	2	16	7	A	sex_code=2

次に、SELECT句で集約関数が各グループに対して計算され、次のようになる。

集約関数適用後 (S.sex_code, COUNT(*), AVG(S.stud_age), (MAX(S.stud_age) – MIN(S.stud_age)) AS age_range)

sex_code	COUNT(*)	AVG(stud_age)	(MAX(stud_age) – MIN(stud_age)) AS age_range
1	4	16.75	(18 – 16) = 2
2	1	16.00	(16 – 16) = 0

そしてHAVING句が適用され、最後にSELECT句が適用される。最終結果はこうなる。

HAVING句の適用後 **HAVING COUNT(*) > 3;**

sex_code	COUNT(*)	AVG(stud_age)	age_range
1	4	16.75	2

繰り返しになるが、実際にこのようなステップを踏んで中間テーブルを次々に作る製品は存在しない。あまりに実行コストが高いからだ（特にクロス結合の箇所が）。ここに示したのは、SELECT 文の動作を示すモデルにすぎない。FROM 句にはもっと複雑な方法でワークテーブルを作る結合その他の演算子を書くことができるが、その場合でも、上に示した順序で実行すれば正しい結果が得られる。あとはただ、入れ子が深くなっていくだけだ。つまり、WHERE句のサブクエリは単純なクエリと同じやり方でパースされ、結果セットに変換されるということだ。

単純なSELECT文は、最適化も簡単だ。単純なSELECT文は、きちんと設計され、正規化されたスキーマでこそ最もよく機能する。

【訳注1】　PostgreSQLやMySQLなど、SELECT句で付けた別名を GROUP BY 句で参照できるものもある。だがこれは標準違反の独自拡張なので、普遍性はない。

【訳注2】　逆に言うと、導出テーブルを使えば上位レベルのSELECT句で下位のSELECT句で付けた別名を使うことができる。

25

高度なSELECT

Advanced SELECT Statements

　前章では基本的なSELECT文の使い方を見た。本章ではサブクエリを入れ子にしたり、ワークテーブルをFROM句で作る構文を見ていこう。

25.1

相関サブクエリ

ソフトウェア開発の古典の1つに、故エドガー・ダイクストラの短い論文がある。

"Go To Statement Considered Harmful"（*Communications of the ACM,* Vol. 11, No. 3, March 1968, pp. 147–148）

この論文で彼は、プログラミング言語からGOTO文を追放すべきだと主張した。いまでいう構造化プログラミングの提唱だ。

彼が発見したのは、BEGIN-ENDブロック、WHILEループ、IF-THEN-ELSE文だけを使うプログラムのほうが読みやすく保守しやすいという事実だった。GOTO文であちこちにジャンプしまくるプログラムの可読性が低い理由は、コード内の任意の箇所のGOTOラベルで実行パスが決まってしまうからだ。初めてのブロック化された構造化プログラミング言語はAlgolだった。上に挙げたような制御構造をすべて備えていたが、まだGOTO 文を持っていた――それほどにGOTO は、プログラミングの根本をなす一部分だと考えられていたのだ！ダイクストラ以前には、手続き型のコードにおける制御構造と変数スコープの制限が持つ意味を理解している人間はいなかった。スコープの基本的な考え方は、あるコードのブロックはそのブロック内で宣言された変数だけを参照できるというものだ。ローカル変数が見つけられなかった場合には、内側から外側に向かって変数の探索範囲を広げていく。

SQL においても、スコープの概念が存在する。相関サブクエリは、まさに外側のクエリで使われているテーブルの列を参照するよ

うなサブクエリだ。その一般的な構造は次のような形をしている。

```
SELECT T1.x, T1.y, T1.z
  FROM TableOne AS T1
 WHERE T1.x = (SELECT T2.x
                 FROM TableTwo AS T2
                WHERE T2.y = (SELECT foobar
                                FROM TableThree AS T3
                               WHERE T3.x = T1.x
                                 AND T3.y = T2.y
                                 AND T3.z = 42));
```

一番内側のクエリを見てほしい。述語(T3.z = 42)は、このクエリに閉じている。一方、述語(T3.y = T2.y)は、外側のテーブルT2を参照しているが、このコードが正しく動作するのは、このサブクエリが、T2をFROM 句に持つ1 つ外側のクエリのスコープに含まれているからだ。同様に、述語(T3.x = T1.x)が正しく動作するのは、一番外側の（テーブルT1をFROM句に持つ）クエリのスコープに含まれているからだ。もし、一番内側のWHERE 句でテーブル名を修飾しなかった場合、述語(T3.x = x)はローカルのxを参照するようになる。これは(T3.x = T3.x)という意味になり、評価結果は常にTRUEかUNKNOWNにしかならない。これは明らかに馬鹿げた比較だ。

しかし、述語が(T3.z = floob)のような場合、データベースエンジンはテーブルT1、T2、T3を参照し最初に見つけたfloob列を使う。これが、列名の修飾が大事な理由である。

同じテーブルを異なる相関名で参照することも可能だ。たとえば、生徒（Students）テーブルから、同性の一番年上の生徒よりも若い生徒をすべて見つけるクエリを考えよう。生徒テーブルには次のようなデータを入れる。

Students

stud_nbr	stud_name	sex_code	stud_age
1	Smith	1	16
2	Smyth	2	17
3	Smoot	2	16
4	Adams	2	17
5	Jones	1	16
6	Celko	1	17
7	Vennor	2	16
8	Murray	1	18

```
SELECT S1.stud_nbr, S1.stud_name, S1.sex_code, S1.stud_age
  FROM Students AS S1
 WHERE stud_age
         < (SELECT MAX(stud_age)
              FROM Students AS S2
             WHERE S1.sex_code = S2.sex_code);
```

このクエリは`Adams`と`Smyth`と`Murray`以外の生徒を返す。では、動作を詳細に見ていこう。

1. SQL のモデル上は、ローカルテーブル`S1`と`S2`を作ることになっている。この2 つは、たまたま`Students`テーブルと同じテータと構造を持っている。いわば、`Students`テーブルのコピーが、それぞれの相関名に割り当てられるイメージだ。もちろん、実際のSQLコンパイラがこのような動作をするよう実装さ

れているわけではない。これはあくまでモデル上の話だと考えてほしい。

単純なSELECT文のときと同じステップを踏むことで外側のクエリを展開していこう。

2. WHERE句を見ると、サブクエリがあることがわかる。このサブクエリは相関サブクエリなので、外側のクエリからデータを取得しながら実行していく。実行モデルとしては、外側の1 行についてサブクエリが一度実行される。これを外側の行数だけ繰り返すのだが、理論的にはパラレル実行可能だ。

   たとえばいま、(1, 'Smith')という男子生徒について動作を見てみよう。この生徒の行に対してクエリは次のように動く。

```
SELECT 1, 'Smith', 1, 16
  FROM Students AS S1
 WHERE 16 < (SELECT MAX(stud_age)
               FROM Students AS S2
              WHERE 1 = S2.sex_code);
```

余談だが検索条件(1 = S2.sex_code)と(S2.sex_code = 1)は論理的には同値である。どちらの書き方を選ぶかはプログラマの文化に依存している。読者は左から右に読むのと右から左に読むのと、どちらの文化圏に属しているだろうか？

3. 男子生徒に対するサブクエリを計算すると、最高の年齢は18 歳であるとわかる。同様に、Studentsテーブルのすべての行に対して展開すると、実質的には以下のようなパラレルクエリの集合を実行するのと同じことになる。

```
SELECT 1, 'Smith', 1, 16 FROM Students AS S1 WHERE 16 < 18;
SELECT 2, 'Smyth', 2, 17 FROM Students AS S1 WHERE 17 < 17;
SELECT 3, 'Smoot', 2, 16 FROM Students AS S1 WHERE 16 < 17;
SELECT 4, 'Adams', 2, 17 FROM Students AS S1 WHERE 17 < 17;
SELECT 5, 'Jones', 1, 16 FROM Students AS S1 WHERE 16 < 18;
SELECT 6, 'Celko', 1, 17 FROM Students AS S1 WHERE 17 < 18;
SELECT 7, 'Vennor', 2, 16 FROM Students AS S1 WHERE 16 < 17;
SELECT 8, 'Murray', 1, 18 FROM Students AS S1 WHERE 18 < 18;
```

このモデルでは、すべてのサブクエリが一度に評価される。ハードウェアがますます安くなっている昨今、パラレル化のコストも下がり続けていることを考えれば、すべてのサブクエリを一度に評価可能になる日も来るかもしれない。だが、現在のところ、そこまで進んだ実装は存在しない。実際のSQLコンパイラが通常採用するアプローチは、両方のテーブルをスキャンする二重ループを作ることだ。どちらのテーブルが外側になるかは、オプティマイザが決める。

最終的な結果は次のようになる。

stud_nbr	stud_name	sex_code	stud_age
1	Smith	1	16
3	Smoot	2	16
5	Jones	1	16
6	Celko	1	17
7	Vennor	2	16

もう一度繰り返すが、この説明は実際の製品の動作と同じではない。だが結果はこのプロセスで実行したときと同じでなければなら

ない。

相関サブクエリを入れ子にできる深さに理論上の制限はない。実際には、せいぜい5レベル以下の入れ子にとどめておくのが経験的には望ましいと思う。この5という数は心理学において人間が頭の中で一度に扱うことのできる上限である。古典的な研究としてはジョージ・ミラーの以下の論文がある。

> "The Magical Number Five Plus or Minus Two: Some Limits on Our Capacity for Processing Information"（The Psychological Review, 1956）

上の例ではスカラサブクエリを使ったが、相関サブクエリはテーブルの集合を返すこともできる。たとえば、セルコ教授の学生をすべて見つけるクエリは次のように書ける。

```
SELECT S1.stud_nbr, S1.stud_name, S1.sex_code, S1.stud_age
  FROM Students AS S1
 WHERE S1.stud_nbr
         IN (SELECT T.stud_nbr
               FROM Teachers AS T1
              WHERE S1.stud_nbr = T1.stud_nbr
                AND T1.teacher_name = 'Celko');
```

サブクエリにおける問題の1つは、多くのSQLプログラマが導出テーブル名のスコープのルールをよく理解していないことだ。もし結合演算の入力に導出テーブルが与えられた場合、その中に含まれるすべてのテーブル名は、外側の式からは一切隠蔽される。たとえば、次のSELECT文は構文的に正しくないため失敗する。

```
SELECT x, y, z -- 間違い！
  FROM (Foo INNER JOIN Bar
         ON Foo.y >= Bar.x) AS Foobar (x, y)
        INNER JOIN Flub
    ON Foo.y <= Flub.z;
```

なぜなら、テーブル名Fooは2つ目のINNER JOINでは利用できないからだ。ただし、次のように結合条件でFoobarを使えば正しく動作する。Foobarは露出しているからだ。

```
SELECT x, y, z
  FROM (Foo INNER JOIN Bar
          ON Foo.y >= Bar.x) AS Foobar (x, y)
        INNER JOIN Flub
    ON Foobar.y <= Flub.z;
```

これまでの例では、完全に修飾された列名を使ってきた。これは良いプログラミング習慣だが、強制というわけではない。たとえば、テーブルFooとBarがどちらもwという名前の列を持っているとしよう。その場合、次の文はどちらも名前があいまいであることによるエラーが発生する。

```
SELECT x, y, z
  FROM Foo INNER JOIN Bar
    ON y >= x
            INNER JOIN Flub
              ON y <= w;

SELECT x, y, z
  FROM Foo, Bar, Flub
 WHERE y BETWEEN x AND w;
```

だが次の文は正しく動作する。なぜなら括弧の内側が最初に評価され、次に外側の内部結合が実行されるからだ。

```
SELECT x, y, z
  FROM Foo INNER JOIN
       (Bar INNER JOIN Flub
              ON y <= w)
    ON y >= x;
```

なお、もしテーブルBarがwという名前の列を持っていなければ、パーサは外側のクエリを見に行き、Foo.wを見つけるだろう。

25.2

標準的な構文の内部結合

SQL-92 は、FROM句で結合のための専用の演算子を使う構文を追加した。結合演算子は極めて汎用的かつ柔軟であり、古い表記法ではできなかったことも1 つの文で簡単にできるようになった。基本的な構文は以下のとおりだ。

```
<joined table> ::=
<cross join> | <qualified join> | (<joined table>)
<cross join> ::= <table reference> CROSS JOIN
<table reference>
<qualified join> ::=
<table reference> [NATURAL] [<join type>] JOIN
<table reference> [<join specification>]
<join specification> ::= <join condition> | <named columns join>
<join condition> ::= ON <search condition>
<named columns join> ::= USING (<join column list>)
<join type> ::= INNER | <outer join type> [OUTER] | UNION
<outer join type> ::= LEFT | RIGHT | FULL
<join column list> ::= <column name list>
<table reference> ::=
<table name> [[AS] <correlation name>[(<derived column list>)]]
| <derived table>
[AS] <correlation name> [(<derived column list>)]
| <joined table>
<derived table> ::= <table subquery>
<column name list> ::=
<column name> [{<comma> <column name>} ...]
```

内部結合はクロス結合の結果から、ON 句で与えられた結合条件を満たさない行を削除する。ちょうどオリジナルのFROM句とWHERE句の構文で行ったのと同じような調子だ。ON句にはテーブル名と列名のスコープ内で、非常に精密な条件式を記述できる。もし、<qualified join>が<join type>なしに使われた場合、暗黙に内部結合を意味する。だが、常にすべての結合演算子を明示的に書くのがよいだろう。

もっとも、現実の世界で内部結合を使う場合、複雑な条件を記述するよりは別々のテーブルに存在する同じ名前の列を単純に等号で結んで使うことのほうが多いだろう。等値結合は非常によく使われるので、標準SQLでは2 種類の簡略的な記法を用意している。1つがUSING (c1, c2, ..., cn)のように列名をリストで並べる書き方だ。これは次のような行式と同じである。

```
ON ((T1.c1, T1.c2, ..., T1.cn) = (T2.c1, T2.c2, ..., T2.cn))
```

2 つ目がNATURAL オプションである。これはUSING 句と一緒に使って2 つのテーブルの列名が同じであることを意味する。NATURAL オプションを使う場合は、結合条件は記述しない。それもすでに含まれているからだ。NATURALオプションを使う結合を自然結合と呼ぶ。

ただし、自然結合を実際のコードで使うことは推奨できない。列名を変更したり、新しい列を追加したりすると実行時にエラーになったり結果が変わってしまう。これは「SELECT *」を使ってはいけない理由と同じだが、自然結合のほうがより危険だ。ダニエル・モーガンが指摘したように「comments」のようなあいまいで一般的な名前の列名を持つテーブルを自然結合すると、本当はまったく無関係な列同士を、意味もなく大掛かりな結合をすることになりかねない。

同様の注意は、USING 句にも必要だ。もし読者の使っているスキーマの中にproduct_id やproduct_nbr、あるいはupcといった列

名が複数のテーブルで使われているようであれば、それぞれの要素が一意な名前を持つようスキーマ全体を変更するべきだろう。

なお、ときどき誤解されているが、ON 句に書けるのは結合条件だけで、WHERE 句に書けるのは検索条件だけだと信じている人がいる。これはまったく事実ではないし、検察条件をどこに書くかというのは重要ではない。

とはいえ、結合条件と検索条件の記述場所を分けるのは、人間にとって読みやすいので、そのような習慣を持つべきだ。そのほうが、たとえばWHERE 句は削除してテーブルを結合するだけのクエリをビューとして持つ、といったやり方も可能だ。だがもちろんここには、利便性とパフォーマンスのトレードオフがあることは忘れてはならない。

25.3

外部結合

外部結合はかつて、実装依存のベンダー構文で表記されていた。今日では標準のOUTER JOIN 構文がどの製品でも使える。外部結合は結合対象となる2つのテーブルの片方または両方のすべての行を、たとえ相手の行とマッチしなかった場合でも保存する結合である。マッチしなかった列はNULLが埋められる。

サンプルを使って見てみよう。注文テーブルと業者テーブルを結合して各業者とどれだけの取引があったかを示すレポートを作りたい。伝統的な内部結合を使って書くと、次のようになる。

```
SELECT S.sup_id, S.sup_name, O.order_nbr, O.order_amt
  FROM Suppliers AS S, Orders AS O
 WHERE S.sup_id = O.sup_id;
```

結合を明示する構文ならば次のとおりだ。

```
SELECT S.sup_id, S.sup_name, O.order_nbr, O.order_amt
  FROM Suppliers AS S -- 保存すべきテーブル
         INNER JOIN Orders AS O
    ON S.sup_id = O.sup_id;
```

業者の中には、返品された商品のクレジットを合計に含むものもあるので、そのような業者との取引量の合計は結局0ドルになる。また、いくつかの業者とは、そもそも取引をしたことがないので、やはり0ドルとなる。だが最初のケースでは、この0ドルというのはレポートに表示されるが、2 つ目のケースの場合は、このような業者は内部結合を使うとそもそも結果に現れない。これはジョッキの中に0

リットルのビールが入っているか、そもそもジョッキを持っていないかの違いである。

外部結合を使えば業者テーブルを保存できるので、レポートにはすべての業者が現れる。注文テーブルには1行の注文もない業者であっても、結果には現れるのだ。その業者のO.order_nbrとO.order_amtはNULLになる。

```
SELECT S.sup_id, S.sup_name, O.order_nbr, O.order_amt
  FROM Suppliers AS S
         OUTER LEFT JOIN
           Orders AS O
    ON S.sup_id = O.sup_id;
```

25.3.1　外部結合の歴史

SQL-99 以前、外部結合の標準的な構文は存在しなかった。そのため外部結合と同じことをやろうとするには、UNIONを使った面倒なクエリを書かなければならなかった。

```
SELECT sup_id, sup_name, order_amt -- 普通の内部結合
  FROM Suppliers, Orders
 WHERE Suppliers.sup_id = Orders.sup_id
UNION ALL
SELECT sup_id, sup_name, CAST(NULL AS INTEGER) -- 保存する行
  FROM Suppliers
 WHERE NOT EXISTS
        (SELECT *
           FROM Orders
          WHERE Suppliers.sup_id = Orders.sup_id);
```

UNION を正しく動作させるには正しいデータ型のNULL を使う必要があり、そのためにCAST 関数を使う必要がある。製品によっては賢いことに、ただNULLを与えただけで自動的に正しいデータ型を判別してくれるが、CASTを使うほうが移植性があって安全だ。

NULLを使わない方法としては、値が欠落していることを示す何らかの定数を挿入するという方法もある。これは文字列型の列ならば簡単にできる。たとえば、'{{NONE}}'のようなメッセージならば、読み手も簡単に理解できる。数値型の場合は非常に難しい。たとえば、上の例だと返品や支払いを意味する列は正、負、0のいずれの値も取りうるので、値の欠落を意味する特別な記号に利用できる数値が見つからない。先ほども言ったように、そもそも取引をしたことのない業者と、取引はしたがキャンセルしたことで支払いが0ドルだった業者とは違うのである。

本書の旧版では、外部結合の実装依存の拡張について説明していたが、今日ではそうした実装依存の構文は消え去り、標準構文で置き換えられた。ベンダーの拡張は、構文、意味、またはその両方において互いに違っていた。ありがたいことに、もうこうした違いを気にとめる必要はないので、本書ではこの点については省略する。

なお、左外部結合という名前の由来は、保存対象のテーブルを演算子の左側に書くことからきている。同じく右外部結合というのは保存対象のテーブルを右側に書くからだ。完全外部結合という結合もあるがこれは両方のテーブルを保存する（これについてはあとで詳しく見る）。

では、外部結合の動作を詳しく見てみよう。以下のような外部結合の結合式があるとする。

```
SELECT Table1.a,Table1.b,Table2.a,Table2.c
  FROM Table1 LEFT OUTER JOIN Table2
    ON Table1.a = Table2.a -- 結合条件
 WHERE Table2.c = 't';     -- SARG
```

この場合、Table1が保存対象のテーブルとなりTable2は保存されない。いまから示す実行ステップは、標準SQLのモデルとは少し異なるが、結果は同値である。

1. まず2つのテーブルをクロス結合する。そして結果セットの行を1行ずつスキャンする。
2. もし条件がTRUEになる行があればその行を保持する。
3. もし条件がFALSEまたはUNKNOWN になれば、その行については保存対象のテーブルの列を保持し、保存対象ではないテーブルのすべての列をNULLに変換する。最後に重複行を削除する。

1 つずつステップを追っていって見てみよう。2 つのテーブルには、次のようなデータが入っているとする。

Table1		Table2	
a	b	a	c
1	w	1	r
2	x	2	s
3	y	3	t
4	z		

次の表で「◀」は「最初の条件をクリア」を表し、「● 」は「2番目の条件をクリア」を表している。

Table1とTable2をクロス結合

Table1		Table2	
a	b	a	c
1	w	1	r ◀
1	w	2	s
1	w	3	t ●
2	x	1	r
2	x	2	s ◀
2	x	3	t ●
3	y	1	r
3	y	2	s
3	y	3	t ◀ ● TRUEになる行集合
4	z	1	r
4	z	2	s
4	z	3	t ●

外部結合の結果は次のようになる。

Table1とTable2を左外部結合

Table1	Table2

a	b	a	c
1	w	1	r
2	x	2	s
3	y	3	t
4	z	NULL	NULL

保存対象のテーブルの行はすべて結果の中に、少なくとも1 行は登場するのが、外部結合の基本ルールだ。このサンプルで言えば、`Table1`の行はすべて結果に現れている。あとは、この結果に「`Table2.c ='t'`」というSARGの条件を適用すれば、最終結果は1行に絞り込まれる。

次に、クリス・デイトがよく使う有名なサンプル「業者と部品」のサンプルを使って考えてみよう。

業者と部品

Suppliers	SupParts		
sup_id	sup_id	part_nbr	part_qty
S1	S1	P1	100
S2	S1	P2	250
S3	S2	P1	100
	S2	P2	250

外部結合の結合条件をON句にだけ書くようにすれば次のようなクエリになる。

```
SELECT Suppliers.sup_id, SupParts.part_nbr, SupParts.part_qty
  FROM Suppliers
         LEFT OUTER JOIN
           SupParts
    ON Suppliers.sup_id = SupParts.sup_id
 WHERE part_qty < 200;
```

結果

```
sup_id  part_nbr  part_qty
==========================
S1      P1        100
S2      P1        100
```

だが、ON句に検索条件も書くと次のような結果になる。

```
SELECT Suppliers.sup_id, SupParts.part_nbr, SupParts.part_qty
  FROM Suppliers
         LEFT OUTER JOIN
           SupParts
    ON Suppliers.sup_id = SupParts.sup_id
   AND part_qty < 200;
```

結果

```
sup_id  part_nbr  part_qty
==========================
S1      P1        100
S2      P1        100
S3      NULL      NULL
```

なぜ結果が変わるか？ それはモデル上、WHERE 句はON 句のあとで実行されるため、最初のクエリだと、S3について「NULL <200」という評価が行われ、UNKNOWNになるからだ。

かつてあった外部結合に関するもう1 つの問題は、実装依存の構文では、同じテーブルを、保存対象のテーブルと保存対象でないテーブルとして同時に使うことができないことだった。だが標準SQLではこれは簡単だ。たとえば、すべての生徒について、「数学101」と「数学102」を受講したかどうかを1行で表示するには、次のように書けばよい。

```
SELECT C1.stud_nbr, C1.math_course, C2.math_course
  FROM (SELECT stud_nbr, math_course
          FROM Courses
         WHERE math_course = 'Math 101') AS C1
           LEFT OUTER JOIN
            (SELECT stud_nbr, math_course
               FROM Courses
              WHERE math_course = 'Math 102') AS C2
                 ON C1.stud_nbr = C2.stud_nbr;
```

外部結合の3 つ目の問題は、外部結合を複数回繰り返すときは、実行順序によって結果が変わることだ。つまり、`((T1 OUTER JOIN T2) OUTER JOIN T3)`と`(T1 OUTER JOIN(T2 OUTER JOIN T3))`は同じ結果とは限らない。この問題はあとでもう少し踏み込んで論じる。

25.3.2　NULL と外部結合

外部結合によって生成されるNULLは、元のテーブルでNOT NULL制約が宣言されている列においても現れることがある。たとえすべてのテーブルのすべての列にNOT NULL制約を付けたとしても、NULLの問題からは逃げられない。外部結合とOLAP 演算子の結果には、NULLが現れることがあるからだ。

だが、NULLを含むテーブルを外部結合に使うことはできる。次の2つのテーブルは共通の列にNULLを持っている。この2つのテーブルに対して異なる結合を適用するとどういう結果になるか、見てみよう。

Table1とTable2を左外部結合

T1		T2	
a	x	b	x
1	r	7	r
2	v	8	s
3	NULL	9	NULL

xに対する内部結合は当然ながら両者が一致する値の行だけを残す。NULLは何者とも一致しないので（たとえ他のNULLとも一致しない）、結果にはx列の値が'r'の1行だけがヒットする。

```
SELECT *
  FROM T1 INNER JOIN T2
    ON T1.x = T2.x;
```

結果

```
 a    T1.x    b    T2.x
========================
 1    r       7    r
```

次にT1を保存対象のテーブルとする左外部結合を実施しよう。すると結果は次のようになる。

```
SELECT *
  FROM T1 LEFT OUTER JOIN T2
    ON T1.x = T2.x;
```

結果

```
 a    T1.x    b     T2.x
=========================
 1    r       7     r
 2    v       NULL  NULL
 3    NULL    NULL  NULL
```

この場合も特に驚くところはない。この結果には先ほどの内部結合の行も含まれている。T1 の残り2つの行は同値結合の結果としてはマッチしないので、T2の列はNULLが埋められる。右外部結合の場合も同様である。問題が起きるのは完全外部結合の場合だ。結果を見てみよう。

```
SELECT *
  FROM T1 FULL OUTER JOIN T2
    ON (T1.x = T2.x);
```

結果

```
 a     T1.x    b     T2.x
============================
 1     r       7     r
 2     v       NULL  NULL
 3     NULL    NULL  NULL
 NULL  NULL    8     s
 NULL  NULL    9     NULL
```

なぜこのような結果が生じるのかは、詳しく見る価値がある。最初にT1とT2に対してON 句の条件を使った内部結合が行われる。そして、結果にマッチした行が追加される。次にT1の結合できなった行についてT2の列にNULLが埋められてから結果に追加される。最後にT2の結合できなかった行についてやはりNULLが埋められてから結果に追加される。完全外部結合のやっかいなところは、結合の結果からはオリジナルの2つのテーブルを再構成することはできないことだ。試しに完全外部結合の結果（これをR1と呼ぶ）に対してSELECT文を実行してみよう。

```
SELECT T1.a, T1.x FROM R1;
```

結果

```
 a     x
============
 1     r
 2     v
 3     NULL
 NULL  NULL
 NULL  NULL
```

この結果はオリジナルのT1とは異なる。なぜ違いが生じるかというと、外部結合によって生成されたNULLとオリジナルのNULLを区別できないからだ。だからと言って、このNULLの重複行を削除する言うわけにもいかない。もしかしたらT1に最初から存在していた行かもしれないからだ。今日では、標準SQLにはGROUPING(<column name>)という関数が存在する。これはCUBE、ROLLUP、GROUPING SET オプションと併用し、オリジナルのNULLに対しては1を、生成

されたNULL に対しては0 を返すことで、2 つのNULLを区別するための関数だ。もしかすると、読者の使う製品ではこの関数を使うことで外部結合の生成したNULLも区別できるかもしれない。

25.3.3　自然外部結合と条件付き外部結合

標準SQL では左外部結合にNATURALオプションを付けることができる。ただしこれはまだ、ほとんどの製品では実装されていない。自然結合は結果の中に共通の列のコピーを1 つしか持たない【訳注1】。一方、条件付き外部結合を使えばオリジナルのテーブルの列をテーブル名を修飾することで両方残すことができる。ちなみに自然左外部結合は、COALESCE関数を使うことでも同じ機能を実装できる。

やり方は同じ名前の列のペアをこの関数を使って1 つの列に統合してビューの中に入れるのだ。たとえば、次のようなクエリだ。

```
CREATE VIEW NLOJ12 (x, a, b)
AS SELECT T1.x, T1.a, T2.b
     FROM T1 LEFT OUTER JOIN T2
       ON T1.x = T2.x;
```

NLOJ12

```
 x     a     b
==================
 r     1     7
 v     2     NULL
 NULL  3     NULL
```

自然結合と違って、条件付き外部結合は、結合の条件として単純な一列に対する同値性を使う必要はない。複数の結合条件が書ける

し、等号以外の比較演算子も使える。たとえば、次のようなイメージだ。

```
T1 LEFT OUTER JOIN T2 ON (T1.x < T2.x)

 a    T1.x    b     T2.x
============================
 1    r       8     s
 2    v       NULL  NULL
 3    NULL    NULL  NULL
```

これを次の結果と比較してみてほしい。

```
T1 LEFT OUTER JOIN T2 ON (T1.x > T2.x)

 a    T1.x    b     T2.x
============================
 1    r       NULL  NULL
 2    v       7     r
 2    v       8     s
 3    NULL    NULL  NULL
```

25.3.4　自己外部結合

外部結合には、同じテーブル同士を結合してはいけないというルールはない。それどころか、外部結合を自己結合と組み合わせるのは、出力帳票の形に合わせてテーブルを水平展開する、うまいトリックである。たとえば、次のような学生の履修単位を管理するテーブルを考えよう。

```
CREATE TABLE Credits
(student_nbr INTEGER NOT NULL,
 course_name CHAR(8) NOT NULL,
     PRIMARY KEY (student_nbr, course_name));
```

Credits：履修単位テーブル　　student_nbr：学生番号　　course_name：講座名

このテーブルは学生IDと彼らが受講した講座の名前を記録する。ただし、'CS-101'の講座を取るまえには'CS-102'の単位を取ることはできないというルールがある。同様に'CS-102'の講座を取る前は'CS-103'は取ることができない。では、このテーブルにサンプルデータを投入しよう。学生番号1は101と102の2つの講座を修了している。学生番号2はまだ101だけだ。問題なのは学生番号3である。この学生は101を飛ばしていきなり102を履修している。これはルール上許されないことなので、この学生は102の単位を得ることはできず101からやり直してもらうしかない。

Credits

student_nbr	course_name
1	CS-101
1	CS-102
2	CS-101
3	CS-102

さて、学生たちの履修状況を確認するためのヒストグラム（棒グラフ）を作ろう。これを見ればどの学生がどれぐらい学習プログラムを進めているかわかる。調べる対象は101と102の2つの講座だけだと仮定しよう。期待するクエリの結果は次のような形になるだろう。ここでNULL は講座を未履修であることを意味する。

```
(1, 'CS-101', 'CS-102')
(2, 'CS-101', NULL)
```

この結果を得るのに自己結合を使う必要があるのは明らかだ。なぜなら、右2 つの列は同じテーブルの同じ列を使っているからだ。自己外部結合を使うにはテーブルに相関名を与える必要がある。これは一般の自己結合のときと同じルールだ。このクエリの結果から、学生番号3は除外されることに注意してほしい。学生番号3 は最初に履修した講座が102 なので、WHERE 句の最初の条件で除外される。

```
SELECT C1.student_nbr, C1.course_name, C2.course_name
  FROM Credits AS C1
         LEFT OUTER JOIN
           Credits AS C2
    ON C1.student_nbr = C2.student_nbr
   AND C1.course_name = 'CS-101'
   AND C2.course_name = 'CS-102'
 WHERE COALESCE(C1.course_name, 'CS-101') = 'CS-101'
   AND COALESCE(C2.course_name, 'CS-102') = 'CS-102';
```

25.3.5　2 つ以上の外部結合

リレーショナル原理主義者の中には、すべての演算子は逆演算を持つべきだと考える人もいる。当然の成り行きとして、この手の人々は外部結合が好きではない（先に見たように、外部結合は可逆でない）。また、生成されたNULLはテーブルに元から存在しているNULLとは根本的に異なるものであるから、これを表す特別なトークンが必要だと考える人もいる。残念ながらSQLは汎用的な1種類のNULLしか持たないので、彼らの期待に添うものではない。理論的な話はともかく、しばしばベンダーも製品に奇妙な実装を行うことがある。

1つの大きな問題は外部結合の演算子が内部結合の演算子と同じ性質を持たないことだ。たとえば、完全外部結合は2つ以上使う場合実行順序によって結果が違う。数学者ならこれを「結合法則がない」と言うだろう。次に3つ以上のテーブルを使うときに生じるいくつかの問題を見ていく。非常に単純な2列だけのテーブルを使う。

```
CREATE TABLE T1 (a INTEGER NOT NULL, b INTEGER NOT NULL);
INSERT INTO T1 VALUES (1, 2);

CREATE TABLE T2 (a INTEGER NOT NULL, c INTEGER NOT NULL);
INSERT INTO T2 VALUES (1, 3);

CREATE TABLE T3 (b INTEGER NOT NULL, c INTEGER NOT NULL);
INSERT INTO T3 VALUES (2, 100);
```

この3 つのテーブルに対して左外部結合を2回実施する。問題は結合するテーブルの順序が変わると結果も変わってしまうことだ。

```
SELECT T1.a, T1.b, T3.c
  FROM ((T1 NATURAL LEFT OUTER JOIN T2)
              NATURAL LEFT OUTER JOIN T3);
```

結果

```
 a    b    c
================
 1    2    NULL
```

```
SELECT T1.a, T1.b, T3.c
  FROM ((T1 NATURAL LEFT OUTER JOIN T3)
              NATURAL LEFT OUTER JOIN T2);
```

結果

```
 a    b    c
==============
 1    2    100
```

```
SELECT T1.a, T1.b, T3.c
  FROM ((T2 NATURAL LEFT OUTER JOIN T3)
              NATURAL LEFT OUTER JOIN T1);
```

結果

```
 a     b     c
==================
 NULL  NULL  NULL
```

さらに悪いことに、SELECT 句のリストに含める列を変えることでも結果が変わってしまう。T3.cの代わりにT2.cを表示させると次のようになる。

```
SELECT T1.a, T1.b, T2.c
  FROM ((T2 NATURAL LEFT OUTER JOIN T3)
              NATURAL LEFT OUTER JOIN T1);
```

結果

```
 a     b     c
===============
 NULL  NULL  3
```

25.3.6　外部結合と集約関数

本章の最初のほうで注文テーブルと業者テーブルを使ったのを覚えているだろう。この2 つのテーブルから各業者とどれだけの取引量があったかを調べた。そのクエリを再掲する。

```
SELECT Suppliers.sup_id, sup_name, SUM(order_amt)
  FROM Suppliers
         LEFT OUTER JOIN
           Orders
    ON Suppliers.sup_id = Orders.sup_id
 GROUP BY sup_id, sup_name;
```

結果に含まれる業者の合計額には、返品によって0ドルになるケースがある。取引が存在しなかった業者に関しては、外部結合によってorder_amt 列はNULLになる。外部結合が生成するNULLに対しても集約関数のNULLに対するルールが適用されるため、このような業者も合計の注文数は0になる。

もし、合計額が本当に0のケースと外部結合の結果、NULLが生じただけのケースを区別する必要があるのなら、問題の列にMIN 関数またはMAX 関数を使えばよい。MAX(COALESCE(CAST (order_amt AS VARCHAR(6)), '{{No Orders}}'))のようにすることで、注文がなかった場合はメッセージを表示することもできる（数値から文字列への型変換も忘れないように）。極値関数は入力がNULLの場合はNULLを返す。極値関数はHAVING 句の中でも使えるので、外部結合が生成したNULLの行を削除することもできる。だがそうすると、そもそも外部結合を使った意味がないが。

25.3.7　完全外部結合

完全外部結合は左外部結合と右外部結合の組み合わせである。両方のテーブルの行を完全に保存する。この演算は2 つのテーブルを入力として受け取り、1 つの結果テーブルを出力する。これも形式的な

定義よりは例で説明したほうがわかりやすい。先ほどから使っているT1、T2テーブルを使う。

T1とT2テーブル

T1		T2	
a	x	b	x
1	r	7	r
2	v	8	s
3	NULL	9	NULL

```
T1 FULL OUTER JOIN T2 ON (T1.x = T2.x)

 a       T1.x    b       T2.x
==================================================
 1       r       7       r      -- T1とT2の内部結合
 2       v       NULL    NULL   -- T1から保存された
 3       NULL    NULL    NULL   -- T1から保存された
 NULL    NULL    8       s      -- T2から保存された
 NULL    NULL    9       NULL   -- T2から保存された
```

25.4

UNION JOIN 演算子

標準SQLにはUNION JOINという演算子がある。これは完全外部結合の結果から内部結合の結果を除外した結果を返す。この奇妙な演算を実装している製品は、2013年時点で存在しないし、これを望んでいる人もいないように思える。珍しい例外は、SAS（www.sas.com）の統計アプリケーションである。SASのPROC SQLオプションにはこの演算が含まれている。

```
T1 UNION JOIN T2 ON (T1.x = T2.x)

 a       T1.x    b       T2.x
==================================================
 2       v       NULL    NULL  -- T1 から保存された
 3       NULL    NULL    NULL  -- T1 から保存された
 NULL    NULL    8       s     -- T2 から保存された
 NULL    NULL    9       NULL  -- T2 から保存された
```

この演算の応用例があるとすれば、たとえばこういうケースだ。男性と女性の患者の医療レコードを1 つのテーブルに統合するクエリを考える。前立腺（prostate）を持っているのは男性だけだし、妊娠可能性（pregnancy）があるのは女性だけである。したがって、男性の妊娠可能性についてはNULL、同様に女性の前立腺の項目も自動的にNULLを埋めたい。

```
SELECT *
FROM (SELECT 'male', prostate FROM Males)
     UNION JOIN
     (SELECT 'female', pregnancy FROM Females);
```

結果

```
male  prostate  female  pregnancy
================================
male  no        NULL    NULL
male  no        NULL    NULL
male  yes       NULL    NULL
male  yes       NULL    NULL
NULL  NULL      female  no
NULL  NULL      female  no
NULL  NULL      female  yes
NULL  NULL      female  yes
```

フレデリック・ブロードは、この演算と似た操作を行う面白いトリックを考えついた。それは、学生のデータを保持するテーブルをもとに、国内の学生だけのデータを持つテーブルか、留学生だけのデータを持つテーブルを、パラメータの値に基づいて結合する方法である。これは、左外部結合の「起点」となるテーブルを必要とする点で、厳密には、UNION結合ではない（UNION 結合では2つのテーブルは同等である）。

```
CREATE TABLE Students
(student_nbr INTEGER NOT NULL PRIMARY KEY,
 student_type CHAR(1) NOT NULL DEFAULT 'D'
     CHECK (student_type IN ('D', 'F')));
```

Students：学生テーブル　　student_nbr：学生番号　　student_type：学生種別

```
CREATE TABLE DomesticStudents
(student_nbr INTEGER NOT NULL PRIMARY KEY
     REFERENCES Students(student_nbr));
```

DomesticStudents：国内学生テーブル　　student_nbr：学生番号

```
CREATE TABLE ForeignStudents
(student_nbr INTEGER NOT NULL PRIMARY KEY
     REFERENCES Students(student_nbr));
```

ForeignStudents：留学生テーブル　　student_nbr：学生番号

```
SELECT Students.*, DomesticStudents.*, ForeignStudents.*
  FROM Students
         LEFT OUTER JOIN
           DomesticStudents
    ON CASE Students.student_type
       WHEN 'D'
       THEN 1
       ELSE NULL END = 1
         LEFT OUTER JOIN
           ForeignStudents
            ON CASE Students.student_type
               WHEN 'F'
               THEN 1
               ELSE NULL END = 1;
               WHERE ( Students.student_nbr = DomesticStudents.student_nbr
               OR Students.student_nbr = ForeignStudents.student_nbr);
```

25.5

スカラSELECT 式

単一の値を持つ1 行を返すSELECT 式は、スカラ式が使える場所ならばどこでも使用できる。もしスカラサブクエリの結果が空の場合は、NULLに変換される。この性質を利用して——常にできるわけではないが——外部結合をスカラサブクエリで書き換えることができる【訳注2】。たとえば、次のクエリは、各業者がゼロまたは1つの注文しか持っていない場合は、うまく動作する。

```
SELECT sup_id, sup_name,
       (SELECT order_amt
          FROM Orders
         WHERE Suppliers.sup_id = Orders.sup_id) AS order_amt
  FROM Suppliers;
```

もし業者が複数の注文を受けることがあるならば、注文件数を求めるクエリは次のようになる。

```
SELECT sup_id, sup_name,
       (SELECT COUNT(*)
          FROM Orders
         WHERE Suppliers.sup_id = Orders.sup_id)
  FROM Suppliers;
```

普通に外部結合で書けばこうだ。

```
SELECT sup_id, sup_name, COUNT(*)
  FROM Suppliers
         LEFT OUTER JOIN
           Orders
    ON Suppliers.sup_id = Orders.sup_id
 GROUP BY sup_id, sup_name;
```

結合構文の新旧対決

結合条件をON 句にまとめて書く標準SQL の構文は、ベンダーごとのバラバラな構文を統合する目的で導入された。これは絶対に必要な標準化だった。おかげで、内部結合と外部結合の演算子は、いまではどのベンダーでも統一的に使用できる。結合演算子は二項演算子であり、プログラマは二項演算子の扱いに慣れている――加算、減算、乗算、除算はすべて二項演算だ。ER図も2つのテーブルの間に線を引くことで関係を表すという点で、一種の二項演算と言える。

しかし、この状況はまた、プログラマに問題解決のために線形アプローチを取らせてしまう一因にもなっている。これは、SQLではうまくいかないこともある。たとえば、次のような古い構文で書かれたSQL文を見てみよう。

```
SELECT a, b, c
  FROM Foo, Bar, Flub
 WHERE Foo.y BETWEEN Bar.x AND Flub.z;
```

ON句を使った新構文で書けば（いくつか書き方はあるが）、次のようになる。

```
SELECT a, b, c
  FROM (Foo INNER JOIN Bar
            ON Foo.y >= Bar.x)
         INNER JOIN Flub
    ON Foo.y <= Flub.z;
```

あるいは、次のようにも書ける。

```
SELECT a, b, c
  FROM Foo INNER JOIN
       (Bar INNER JOIN Flub
              ON Foo.y <= Flub.z)
    ON Foo.y >= Bar.x;
```

テーブルの順列のすべての場合を（括弧のありなしも含め）列挙するのは、読者に任せよう。重要なことは、そのいずれの書き方も、三項演算ではないということだ。

人間には、物理的に近い距離にあるもの同士を1つのまとまりであったり、何か関係を持っているものだと考える傾向がある。これは視覚心理学や組版において「近接の法則（Law of Proximity）」と呼ばれている。`ON`句を使った書き方は、これが近接の法則を妨げる方向に作用する。つまり、本来ひとまとまりとしてみなすべき単位の各部分が、複数の箇所に散らばってしまうのだ。

`ON`句の表記法を使うと、プログラマはどうしても、1つずつテーブルを付け加えていく「結合連鎖」のモデルで考えるようになる。最初私がこのサンプルを考えたときは、まず`Foo`と`Bar`の2つのテーブルだけを使ってテストしていた。そのときは何の疑問も感じなかった。違和感を感じたのは、`Flub`テーブルを追加したときだ。構造化プログラミングを教えられたプログラマにとって、一段ずつプログラムを洗練させていくのは良いことだ、というのはマントラの1つであり、私もまた例外ではない。だがこれはやはり手続き型のアプローチであり、SQLには馴染まない。我々は、SQLにおいては宣言的アプローチを採用したい。すなわち、手続きを積み重ねるのではなく、最終結果を一挙に記述するのだ。

だが、オリジナルのコードを見てほしい。読者は3 つのテーブルの関係を表すBETWEEN が見えるだろうか？ そのような見方をするのは、簡単ではないと感じないだろうか？ 実際に我々が見ているのは、テーブルのペアにすぎず、問題の全体ではないのだ。SQLは「すべてを一挙に」行う集合指向言語であって、「一段ずつ」積み上げるボトムアップ的な言語ではない。これはたとえて言うなら、単純な二項演算子の「+」と、一般化されたn 項の合計演算子「Σ」の間にある概念的相違に似ている。

私は別に、ONを使った結合構文に反対しているのではない。ただ、この問題は一見したよりも少し複雑だと言いたいだけだ。もしここに外部結合まで絡んできたら、事態はより一層複雑になる。新機能には常に注意が必要なのだ。

25.7

条件付きの結合

2 つのテーブルを結合する際、それぞれのテーブルの量に基づいて結合することが可能である。この問題の典型例は、ある種類の品物をペアリングして1つの集合として扱うという形式を取る。行と行の結合の際にはなかった制限事項としては、ペアの集合は結果のレベルで制約を持つことである。こういう場合、値は識別子であって、結果の中には現れない。

いま、2つのテーブルXとYがあるとしよう。考えられる状況は次のとおりだ。

状況1

テーブルXのある行が、テーブルYのある1行とだけ一致する。この場合、1つの集合に適用できるマッチング関数が1つ存在するか、あるいは各集合がそれぞれのマッチング関数を持つことが考えられる。マッチング関数の一例は、制約の最適化である。たとえば、卵のパックに色の付いた卵を、どういうふうに色を並べるかを決めたルールに従って詰めていく場合だ。論理パズルの多くはこのモデルを使っている。2 つのマッチング関数の古典は、「安定的な結婚」問題である。一群の男女が、互いに結婚したい異性をランク付けする。この問題はあとで詳しく見る。

状況2

テーブルX のある行がテーブルY の1 行以上の行と一致する。これはナップザック問題とかビン詰め問題と呼ばれるものだ。ビ

ンには1つ以上の物が詰められ、その並びを最適化する。一意な解がある場合もあれば、複数の解がある場合もあるし、妥当な解が見つからないこともある。いまから、この問題について例題とコードを見てみよう。

25.7.1　在庫と注文

非常にシンプルな例を使おう。顧客から注文を受け、それに対して各店舗の在庫にある商品をかき集めて顧客に発送する。問題を簡潔にするため、商品の種類は1つしかなく、注文番号は顧客名で管理すると仮定しよう。また、商品はなるべく店舗番号の小さいほうから優先的に取ってくるとする（そのほうが近いのだ）。

```
CREATE TABLE Inventory
(store_id INTEGER NOT NULL PRIMARY KEY,
 item_qty INTEGER NOT NULL CHECK (item_qty >= 0));
```

Inventory：店舗在庫テーブル　　store_id：店舗ID　　item_qty：在庫数

```
INSERT INTO Inventory (store_id, item_qty)
VALUES (1, 2), (2, 3), (3, 2);
```

```
CREATE TABLE Orders
(customer_id CHAR(5) NOT NULL PRIMARY KEY,
 item_qty INTEGER NOT NULL CHECK (item_qty > 0));
```

Orders：注文テーブル　　customer_id：顧客ID　　item_qty：在庫数

```
INSERT INTO Orders (customer_id, item_qty)
VALUES ('Bill', 4), ('Fred', 2);
```

いまやりたいことはこうである。Bill から4 個の注文を受けており、これを店舗#1 から2 個、店舗#2 から2 個調達して彼に届ける。次に、Fred の注文に対しては、店舗#2 に残っている1 個と、店舗#3から1個を調達する。

```
SELECT I.store_id, O.customer_id,
       (CASE WHEN O.end_running_qty <= I.end_running_qty
             THEN O.end_running_qty
             ELSE I.end_running_qty END
      - CASE WHEN O.start_running_qty >= I.start_running_qty
             THEN O.start_running_qty
             ELSE I.start_running_qty END) AS items_consumed_tally
  FROM (SELECT I1.store_id,
               SUM(I2.item_qty) - I1.item_qty,
               SUM(I2.item_qty)
          FROM Inventory AS I1, Inventory AS I2
         WHERE I2.store_id <= I1.store_id
         GROUP BY I1.store_id, I1.item_qty)
            AS I (store_id, start_running_qty, end_running_qty)
       INNER JOIN
       (SELECT O1.customer_id,
               SUM(O2.item_qty) - O1.item_qty,
               SUM(O2.item_qty) AS end_running_qty
          FROM Orders AS O1, Orders AS O2
         WHERE O2.customer_id <= O1.customer_id
         GROUP BY O1.customer_id, O1.item_qty)
            AS O (customer_id, start_running_qty, end_running_qty)
            ON O.start_running_qty < I.end_running_qty
               AND O.end_running_qty > I.start_running_qty;
```

結果

```
 store_id    customer_id    items_consumed_tally
==================================================
 1           Bill           2
 2           Bill           2
 2           Fred           1
 3           Fred           1
```

なお、このクエリは、順序関数を使って書くことも可能だ。考えてみてほしい。

25.7.2　安定な結婚

これは手続き型言語のクラスで習う古典的な問題の1 つだ。準備はいたって簡単だ。まず夫の候補となる男性の集合と、男性と同数の妻の候補となる女性の集合を用意する。問題は、彼らが安定な結婚生活を送れるように組み合わせてやることだ。

では「安定な結婚」とは何か？ 手短に言えば、男性も女性もこれ以上良い相手を見つけられないような結婚のことだ。n人の男性の集合と、やはりn人の女性の集合があったとすると、すべての男性は、すべての女性に対して好みの優先順位を付けている。このランキングは歯抜けも重複もない1 からnまでの数値である。そして女性の側も同様に、男性の好みの順位付けを行っている。

このパズルの最終目標は、どのペアの男女も「ほかの人と結婚すればよかった」と思うことがないように、n組の結婚を成立させることである。

たとえば、夫の候補として('ジョー・セルコ', 'ブラッド・ピット')、妻の候補として('ジャッキー・セルコ', 'アンジェリーナ・ジョリー')を考えよう。もしジャッキーがブラッド・ピットと結婚したら、彼女としては文句なしだろう。私もまた、アンジェリーナ・ジョリーを伴侶とできるのは光栄だ。しかし、ブラッド・ピットとアンジェリーナ・ジョリーが結婚したほうが、我々2人と結婚するよりもナイスカップルになるのは間違いない。ひとたび彼らが結婚すれば、彼らは安定状態に入り、ジャッキーと私は離婚のチャンスを逃すことになる。

安定な結婚を解くための古典的なアルゴリズムは、バックトラック法に基づいている。つまり、カップルの組み合わせを作っては、不幸な結婚をした組みを修正するやり方である。もし、誰一人としてこれ以上の状況の改善が望めない状況に到達したら、アルゴリズムは停止して、答えを返す。

この問題には、よく知られている重要なポイントが2 つある。それは、①必ず解があって、②しかも多くの場合に2つ以上の解があるということである。安定な結婚は、男性にとっても、女性にとっても、集合レベルで幸福を増大させる。これは必ずしも特定の個人の幸福を最大化する解を意味するわけではない。

```
CREATE TABLE Husbands
(man VARCHAR(5) NOT NULL,
 woman VARCHAR(5) NOT NULL,
 ranking INTEGER NOT NULL CHECK (ranking > 0),
     PRIMARY KEY (man, woman));
```

Husbands：夫テーブル　man：男性　woman：女性　ranking：ランキング

```
CREATE TABLE Wives
(woman VARCHAR(5) NOT NULL,
 man VARCHAR(5) NOT NULL,
 ranking INTEGER NOT NULL CHECK (ranking > 0),
 PRIMARY KEY (man, woman));
```

Wives：妻テーブル　woman：女性　man：男性　ranking：ランキング

```
CREATE TABLE Wife_Perms
(wife_name VARCHAR(5) NOT NULL PRIMARY KEY,
 perm INTEGER NOT NULL);
```

Wife_Perms：妻組み合わせテーブル　wife_name：妻　perm：順序

```
--男性の好みの順位付け
INSERT INTO Husbands VALUES ('Abe', 'Joan', 1);
INSERT INTO Husbands VALUES ('Abe', 'Kathy', 2);
INSERT INTO Husbands VALUES ('Abe', 'Lynn', 3);
INSERT INTO Husbands VALUES ('Abe', 'Molly', 4);
INSERT INTO Husbands VALUES ('Bob', 'Joan', 3);
INSERT INTO Husbands VALUES ('Bob', 'Kathy', 4);
INSERT INTO Husbands VALUES ('Bob', 'Lynn', 2);
INSERT INTO Husbands VALUES ('Bob', 'Molly', 1);
INSERT INTO Husbands VALUES ('Chuck', 'Joan', 3);
INSERT INTO Husbands VALUES ('Chuck', 'Kathy', 4);
INSERT INTO Husbands VALUES ('Chuck', 'Lynn', 2);
INSERT INTO Husbands VALUES ('Chuck', 'Molly', 1);
INSERT INTO Husbands VALUES ('Dave', 'Joan', 2);
INSERT INTO Husbands VALUES ('Dave', 'Kathy', 1);
INSERT INTO Husbands VALUES ('Dave', 'Lynn', 3);
INSERT INTO Husbands VALUES ('Dave', 'Molly', 4);

--女性の好みの順位付け
INSERT INTO Wives VALUES ('Joan', 'Abe', 1);
INSERT INTO Wives VALUES ('Joan', 'Bob', 3);
INSERT INTO Wives VALUES ('Joan', 'Chuck', 2);
INSERT INTO Wives VALUES ('Joan', 'Dave', 4);
INSERT INTO Wives VALUES ('Kathy', 'Abe', 4);
INSERT INTO Wives VALUES ('Kathy', 'Bob', 2);
INSERT INTO Wives VALUES ('Kathy', 'Chuck', 3);
INSERT INTO Wives VALUES ('Kathy', 'Dave', 1);
INSERT INTO Wives VALUES ('Lynn', 'Abe', 1);
INSERT INTO Wives VALUES ('Lynn', 'Bob', 3);
INSERT INTO Wives VALUES ('Lynn', 'Chuck', 4);
INSERT INTO Wives VALUES ('Lynn', 'Dave', 2);
INSERT INTO Wives VALUES ('Molly', 'Abe', 3);
INSERT INTO Wives VALUES ('Molly', 'Bob', 4);
INSERT INTO Wives VALUES ('Molly', 'Chuck', 1);
INSERT INTO Wives VALUES ('Molly', 'Dave', 2);
```

女性のすべての順列を作るには、次の補助テーブルを使うのが便利だ。

```
INSERT INTO Wife_Perms
VALUES ('Joan', 1), ('Kathy', 2), ('Lynn', 4), ('Molly', 8);
```

次のクエリは、女性のすべての順列を作り、その結果から、男女が互いに満足しているペアの有無をNOT EXISTS述語で検査してい

る。

```
SELECT W1.wife_name AS abe_wife, W2.wife_name AS bob_wife,
       W3.wife_name AS chuck_wife, W4.wife_name AS dave_wife
  FROM Wife_perms AS W1, Wife_perms AS W2,
       Wife_perms AS W3, Wife_perms AS W4
 WHERE (W1.perm + W2.perm + W3.perm + W4.perm) = 15
   AND NOT EXISTS
       (SELECT *
          FROM Husbands AS H1, Husbands AS H2,
               Wives AS WV1, Wives AS WV2
         WHERE H1.man = H2.man
           AND H1.ranking > H2.ranking
           AND (H1.man || H1.woman)
                IN ('Abe'   || W1.wife_name,
                    'Bob'   || W2.wife_name,
                    'Chuck' || W3.wife_name,
                    'Dave'  || W4.wife_name)
           AND H2.woman = WV1.woman
           AND WV1.woman = WV2.woman
           AND WV1.ranking > WV2.ranking
           AND (WV1.man || WV1.woman)
                IN ('Abe'   || W1.wife_name,
                    'Bob'   || W2.wife_name,
                    'Chuck' || W3.wife_name,
                    'Dave'  || W4.wife_name));
```

結果

```
 abe_wife    bob_wife    chuck_wife    dave_wife
================================================
 Joan        Lynn        Molly         Kathy
```

25.7.3　ボールの箱詰め問題

以下に示すサンプルは、BeyondRelational（beyondrelational.com）というWeb サイトの、2010年1月のSQL問題22番から拝借したものだ。まず、何個かの箱とボールがある。そのボールを箱に詰めていく。ただし条件がある。ボールは以下のようなルールと優先順位に従って詰める必要があるのだ。

条件1　1つの箱には1つのボールしか詰められない。

条件2　1つのボールは1つの箱にしか詰められない。

条件3　ボールと箱の数は同じである。

条件4　1 つの箱には1 つのボールしか詰められない。すべての箱が詰められなければならないし、ボールが余ってもいけない。

条件5　ボールを箱に詰める際に使う優先順位を設定したテーブルがある。

```
CREATE TABLE Boxes
(box_nbr  INTEGER NOT NULL PRIMARY KEY,
 box_name VARCHAR(20) NOT NULL);
```

Boxes：箱テーブル　box_nbr：箱番号　box_name：箱の名前

```
INSERT INTO Boxes (box_nbr, box_name)
VALUES (1, 'Box 1'), (2, 'Box 2'), (3, 'Box 3'),
       (4, 'Box 4'), (5, 'Box 5'), (6, 'Box 6');
```

```
CREATE TABLE Balls
(ball_nbr INTEGER NOT NULL PRIMARY KEY,
 ball_name VARCHAR(20) NOT NULL);
```

Balls：ボールテーブル　ball_nbr：ボール番号　ball_name：ボールの名前

```
INSERT INTO Balls (ball_nbr , ball_name)
VALUES (1, 'Ball 1'), (2, 'Ball 2'), (3, 'Ball 3'),
       (4, 'Ball 4'), (5, 'Ball 5'), (6, 'Ball 6');
```

```
CREATE TABLE Preferences
(box_nbr INTEGER NOT NULL
     REFERENCES Boxes (box_nbr),
 ball_nbr INTEGER NOT NULL
     REFERENCES Balls (ball_nbr),
     PRIMARY KEY (box_nbr, ball_nbr));
```

Preferences：優先順位テーブル　　box_nbr：箱番号　　ball_nbr：ボール番号

```
INSERT INTO Preferences (box_nbr, ball_nbr)
VALUES (1, 1),
       (2, 1), (2, 3),
       (3, 2), (3, 3),
       (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6),
       (5, 4), (5, 5),
       (6, 5);
```

欲しい結果は次のとおりである。

```
box_name  ball_name
====================
Box 1     Ball 1
Box 2     Ball 3
Box 3     Ball 2
Box 4     Ball 6
Box 5     Ball 4
Box 6     Ball 5
```

この問題を解くには、次のように共通表式を使ってロジックを実装するのがいい。共通表式`BallsInBoxes`は、6個のボールと6個の箱の可能なすべての組み合わせを与える。これが共通表式`PreferredBallsInBoxes`に渡され、ルールに適合する組み合わせだけがフィルタされる。ただし、これだけではまだ、2 つ以上の箱が 1 つのボールを取り合うケースが残っている。最後にこれをメインのクエリで削除して、箱とボールが1対1で対応するようにすれば完成だ。

IN述語は、重複する列が存在しないことを保証するためのものだ。これはボールと箱が何個に増えても、コードを簡単に拡張できるやり方だ（いささか読むには量が多くて骨が折れるが）。私が試した限り、パフォーマンスも決して悪くない。

```
WITH
BallsInBoxes (bx1, bx2, bx3, bx4, bx5, bx6)
AS
(SELECT B1.ball_nbr, B2.ball_nbr, B3.ball_nbr,
        B4.ball_nbr, B5.ball_nbr, B6.ball_nbr
   FROM Balls AS B1, Balls AS B2, Balls AS B3,
        Balls AS B4, Balls AS B5, Balls AS B6
  WHERE B1.ball_nbr NOT IN (B2.ball_nbr, B3.ball_nbr, B4.ball_nbr, B5.ball_nbr,
        B6.ball_nbr)
    AND B2.ball_nbr NOT IN (B3.ball_nbr, B4.ball_nbr, B5.ball_nbr, B6.ball_nbr)
    AND B3.ball_nbr NOT IN (B4.ball_nbr, B5.ball_nbr,B6.ball_nbr)
    AND B4.ball_nbr NOT IN (B5.ball_nbr, B6.ball_nbr)
    AND B5.ball_nbr NOT IN (B6.ball_nbr)),
PreferredBallsInBoxes (bx1, bx2, bx3, bx4, bx5, bx6)
AS
(SELECT bx1, bx2, bx3, bx4, bx5, bx6
   FROM BallsInBoxes AS BB
  WHERE BB.bx1
          IN (SELECT ball_nbr
                FROM Preferences AS P
               WHERE box_nbr = 1)
    AND BB.bx2
          IN (SELECT ball_nbr
                FROM Preferences AS P
               WHERE box_nbr = 2)
    AND BB.bx3
          IN (SELECT ball_nbr
                FROM Preferences AS P
               WHERE box_nbr = 3)
    AND BB.bx4
          IN (SELECT ball_nbr
                FROM Preferences AS P
               WHERE box_nbr = 4)
    AND BB.bx5
          IN (SELECT ball_nbr
                FROM Preferences AS P
               WHERE box_nbr = 5)
    AND BB.bx6
          IN (SELECT ball_nbr
                FROM Preferences AS P
               WHERE box_nbr = 6))
```

```
SELECT bx1, bx2, bx3, bx4, bx5, bx6
  FROM PreferredBallsInBoxes AS PBB
 WHERE PBB.bx1 NOT IN (PBB.bx2, PBB.bx3, PBB.bx4, PBB.bx5, PBB.bx6)
   AND PBB.bx2 NOT IN (PBB.bx3, PBB.bx4, PBB.bx5, PBB.bx6)
   AND PBB.bx3 NOT IN (PBB.bx4, PBB.bx5, PBB.bx6)
   AND PBB.bx4 NOT IN (PBB.bx5, PBB.bx6)
   AND PBB.bx5 NOT IN (PBB.bx6);
```

結果

```
 bx1    bx2    bx3    bx4    bx5    bx6
=======================================
 1      3      2      6      4      5
```

25.8

コッドのT-結合

コッドは、T-演算子という新しいシータ演算子の集合を導入した（Second Version of the Relational Model, 1990）。これは、最適フィットあるいは近似の概念に基礎を置いている。この演算子のアルゴリズムは、コッドの使ったオリジナルよりも、修正版を使うとわかりやすい。

いま、利用可能な教室にクラスを割り当てる問題を考える。満たすべき条件は、`(class_size <room_size)`である。この条件に従って割り当てを行うと、何席か空席の出る教室もあるかもしれないが、かまわない。次のようなテーブルを使って考えよう。

```
CREATE TABLE Rooms
(room_nbr CHAR(3) PRIMARY KEY,
 room_size INTEGER NOT NULL);
```

Rooms：教室テーブル　　room_nbr：教室番号　　room_size：教室サイズ

```
CREATE TABLE Classes
(class_nbr CHAR(3) PRIMARY KEY,
 class_size INTEGER NOT NULL);
```

Classes：クラステーブル　　class_nbr：クラス番号　　class_size：クラスサイズ

テーブルのデータは次のものを使う。

Rooms

room_nbr	room_size
r1	70

r2	40
r3	50
r4	85
r5	30
r6	65
r7	55

Classes

class_nbr	class_size
c1	80
c2	70
c3	65
c4	55
c5	50
c6	40

　この問題を解く方法は2 つある。1 つはテーブルを教室のサイズとクラスの生徒数の昇順でソートする方法。もう1つは反対に降順でソートする方法だ。

方法1　昇順アルゴリズム

両方のテーブルを昇順でソートし、その後、テーブルを頭から読み込み、最初に条件にマッチしたクラスと教室の組み合わせを保存する。

ソートの結果

Classes	
class_nbr	class_size
c6	40
c5	50
c4	55
c3	65
c2	70
c1	80

Rooms	
room_nbr	room_size
r5	30
r2	40
r3	50
r7	55
r6	65

r1	70
r4	85

結果は次のようになる。

class_nbr	class_size	room_nbr	room_size
c2	70	r4	85
c3	65	r1	70
c4	55	r6	65
c5	50	r7	55
c6	40	r3	50

方法2　降順アルゴリズム

両方のテーブルを降順でソートし、その後、テーブルを頭から読み込み、最初に条件にマッチしたクラスと教室の組み合わせを保存する。

Classes	
class_nbr	class_size
c1	80
c2	70
c3	65
c4	55

c5	50
c6	40

Rooms	
room_nbr	room_size
r4	85
r1	70
r6	65
r7	55
r3	50
r2	40
r5	30

結果は次のようになる。

class_nbr	class_size	room_nbr	room_size
c1	80	r4	85
c3	65	r1	70
c4	55	r6	65
c5	50	r7	55
c6	40	r3	50

どちらのアルゴリズムを採用するかによって、答えが変わる点に注意してほしい。コッドは、どちらの定義をT-結合の関係代数の定義にすべきかは語らなかったが、私は、どちらかに決めたほうがよいと思う。この2つのアルゴリズムを非形式的に解説すれば、どのクラスにも、それを収容可能な最小の教室を割り当てるか、どの教室にもそれが収容可能な最大のクラスを割り当てるかの違いである。これが結果の違いを生む理由であり、それゆえT-結合を使うときはどちらのテーブルを駆動にするか決めなければならない。だがどちらにせよ、コッドが提唱した「最初にフィットする」組み合わせを探すアプローチよりも「最適にフィットする」組み合わせを探すアプローチを提唱したい。

この問題では、「～より小さい」の代わりに別のシータ演算子を使うこともできる。もし「～以下」という演算子を使うと、このケースではすべてのクラスが1つの教室に割り当てられることになる。だがこれは、常にそうなるわけではない。このあたりの考察は、読者の練習問題として残しておこう。

標準SQLでこの問題を解く方法として最初に思いつくのは、`GROUP BY`を使うクエリだ。

```
SELECT class_nbr, class_size, MIN(room_size)
  FROM Rooms, Classes
 WHERE Classes.class_size < Rooms.room_size
 GROUP BY class_nbr, class_size;
```

結果

```
 class_nbr    class_size    MIN(room_size)
============================================
 c1           80            85  ⬅ 教室r4
 c2           70            85  ⬅ 教室r4
 c3           65            70
 c4           55            65
 c5           50            55
 c6           40            50
```

このクエリの欠点は、教室のサイズは結果に含められるが、教室番号は含められないことだ。GROUP BY 句に含まれていない列をSELECT句に含めることはできない。それに、85 席ある教室r4がクラスc1とc2の2回使われてしまう。これではダブルブッキングだ。

クラスを収容可能な最も小さいサイズの教室を割り当てるようにすれば、ダブルブッキングの問題は解消する。

```
SELECT class_nbr, class_size, MIN(room_size)
  FROM Rooms, Classes
 WHERE Classes.class_size <= Rooms.room_size
 GROUP BY class_nbr, class_size;
```

結果

```
 class_nbr    class_size    min
================================
 c1           80            85
 c2           70            70
 c3           65            65
 c4           55            55
 c5           50            50
 c6           40            40
```

25.8.1　手続き型のアプローチ

まず手始めに、クラスと教室の考えられるすべての組み合わせを作ることから考える。これはすでに見たように次のクエリで可能

だ。

```
SELECT R.room_nbr, C.class_nbr
  FROM Rooms AS R, Classes AS C
 WHERE C.class_size <= R.room_size
 ORDER BY R.room_nbr, C.class_nbr;
```

極端な話、すべての教室とクラスが同じサイズなら、解の数はn!通りになる。一方、すべての教室が違うサイズなら、組み合わせの爆発を抑えられる。あくまでサンプルということで、教室のサイズはすべて違うという前提で考えよう。次のクエリは、教室とクラスのサイズが完璧に一致する組み合わせを返す。

```
SELECT R.room_nbr, C.class_nbr
  FROM Rooms AS R, Classes AS C
 WHERE C.class_size <= R.room_size
 GROUP BY R.room_nbr, C.class_nbr
HAVING MIN(R.room_size - C.class_size) = 0;
```

結果

```
 room_nbr    class_nbr
=======================
 r1          c2
 r2          c6
 r3          c5
 r6          c3
 r7          c4
```

このままだと、クラス`c1`と教室{r4, r5}は使われないままだ。そこで先ほどのペアを求めるクエリにより限定的な条件を付与して繰り返す。

```
SELECT R.room_nbr, C.class_nbr
  FROM Rooms AS R, Classes AS C
 WHERE C.class_size <= R.room_size
   AND R.room_nbr IN ('r4', 'r5')
   AND C.class_nbr IN ('c1');
```

結果

```
 room_nbr    class_nbr
=======================
 r4          c1
```

この結果をUNIONすれば、答えが得られる。このベストフィットのアルゴリズムは、貪欲法の一種だ【訳注3】。これがより大きなデータに対してうまく動くかどうか見るために、データを変えて試してみよう。クラスと教室に次のようなデータを投入する。

```
INSERT INTO Classes (class_nbr, class_size)
VALUES ('c01', 106), ('c02', 105), ('c03', 104), ('c04', 100), ('c05', 99),
       ('c06', 90), ('c07', 89), ('c08', 88), ('c09', 83), ('c10', 82),
       ('c11', 81), ('c12', 65), ('c13', 50), ('c14', 49), ('c15', 30),
       ('c16', 29), ('c17', 28), ('c18', 20), ('c19', 19);

INSERT INTO Rooms (room_nbr, room_size)
VALUES ('r01', 102), ('r02', 101), ('r03', 95), ('r04', 94), ('r05', 85),
       ('r06', 70), ('r07', 55), ('r08', 54), ('r09', 35), ('r10', 34),
       ('r11', 25), ('r12', 18);
```

今度は、どの程度うまくクラスと教室のサイズが合っているかを調べるために、結果にマッチング度を評価するfitという列（R.room_size - C.class_size）を追加する。これが小さいほどマッチング度が高く、最高は0である。

```
WITH Pairs (room_nbr, class_nbr, fit)
AS
(SELECT R.room_nbr, C.class_nbr,
       (R.room_size - C.class_size)
  FROM Rooms AS R, Classes AS C
 WHERE C.class_size <= R.room_size)
 SELECT P1.room_nbr, P1.class_nbr, fit
   FROM Pairs AS P1
  WHERE P1.fit
        = (SELECT MIN(P2.fit)
             FROM Pairs AS P2
            WHERE P1.room_nbr = P2.room_nbr);
```

結果

```
 room_nbr    class_nbr    fit
==============================
 r01         c04          2
 r02         c04          1
 r03         c06          5
 r04         c06          4
 r05         c09          2
 r06         c12          5
 r07         c13          5
 r08         c13          4
 r09         c15          5
 r10         c15          4
 r11         c18          5
```

今度は、完全にサイズが一致するペアは存在しない。そこで、`(fit = 1)`のデータを一時テーブルに入れて、もとのテーブルから`r02`と`c04`を削除する。結果は次のようになる。

結果

```
 room_nbr    class_nbr    fit
==============================
 r02         c04          1
```

オリジナルのテーブルから最適な教室とクラスのペアは除外されているので、残る次善のペアを見つけよう。

```
WITH Pairs (room_nbr, class_nbr, fit)
AS
(SELECT R.room_nbr, C.class_nbr,
        (R.room_size - C.class_size)
  FROM Rooms AS R, Classes AS C
 WHERE C.class_size <= R.room_size)
(SELECT P1.room_nbr, P1.class_nbr, fit
   FROM Pairs AS P1
  WHERE P1.fit = 2);
```

結果

```
 room_nbr    class_nbr    fit
==============================
 r05         c09          2
```

あとは繰り返しだ。`fit`列が3、4、5、6のケースについては、結果だけを記載しておく。

```
r01    c05    3

r04    c06    4
r08    c13    4
r10    c15    4

r06    c12    5
r11    c18    5

r03    c07    6
r07    c14    6
r09    c16    6
```

ここまでやると、もうオリジナルのテーブルに、ペアになっていないクラスと教室は残っていない。なお、ステップを繰り返すたびに`fit`列の数字は変わっていくことに注意してほしい（これはオリジナルのテーブルから行を削除しているからだ）。

【訳注1】　自然結合を使うには、結合対象のテーブルに同じ名前の列が存在することが前提となる。

【訳注2】　外部結合をスカラサブクエリで置き換える場合は、パフォーマンスによく注意する必要がある。SELECT句に記述されたスカラサブクエリは、クエリが選択する行数だけ繰り返し実行されることがあるからだ。

【訳注3】　貪欲法は欲張り法とも呼ばれる。近似的アルゴリズムの基本的な考え方の1つ。問題を複数の部分問題に分割し、個々の問題について評価値の高い順に解を得ていくことで解にたどり着く。アルゴリズムを単純なステップに分割できるため、わかりやすいという利点がある反面、必ずしも最適解を得られないという欠点がある。

26

仮想テーブル——ビュー、導出テーブル、共通表式、マテリアライズドクエリテーブル

Virtual Tables: VIEWs, Derived Tables, CTEs and MQTs

　ビュー、導出テーブル、共通表式はいずれもクエリを名前付きのスキーマオブジェクトにする方法だ。この機能によってクエリの結果ではなくクエリのテキストを保持することができるようになる。これらは必要なときに実行され、結果が返される。

　ビューは仮想テーブルとも呼ばれる。これは一時テーブルや基底テーブルと区別するための名前だ。標準SQLにおけるビューの定義では、ビューが呼び出されたときは、あたかも実際の物理的テーブルが呼び出されたときと同じように振る舞うことが求められている。データベースが結果を実体化しているかいないかはユーザーにとっては関係ない。物理的なテーブルを使ってもビューを使っても、ユーザーからは同じ動作に見えることが重要なのだ。ビューの定義はスキーマテーブルに保存されており、テーブルが使える場所ならばどこでもビューを呼び出すことができる。なお、もしビューを更新可能に定義するならば、いくつか追加のルールが適用されるので注意が必要だ（26.2節で説明する）。

標準SQLでは、ユーザー権限と管理者権限を区別している。テーブルを作るには管理者権限が必要だが、クエリの実行はユーザー権限だけでよい。逆に言うと、管理者権限がなくてはビューや一時テーブルは作れない。ただしローカルの一時的な仮想テーブルとも言うべき共通表式を作るには、ユーザー権限だけでよい。

26.1

クエリにおけるビュー

ビュー定義の標準SQL構文は次のとおりだ。

```
CREATE VIEW <table name> [(<view column list>)]
AS <query expression>
[WITH [<levels clause>] CHECK OPTION]
<levels clause>::= CASCADED | LOCAL
```

`<levels clause>`オプションはSQL-89には存在していなかった。現在でもあまり実装されていない。この句については26.5節で詳しく述べる。この句はクエリには何の影響も与えないが、更新、挿入、削除に影響を与える。

ビューは一時テーブル、導出テーブル、基底テーブルとは異なる点がいくつかある。たとえば、ビューには制約を付けることはできないが、一時テーブルには付けられる。またビューは呼び出されるまではデータベース内に実体を持たないが、一時テーブルは持続性がある。一方、導出テーブルが存在できるのは、それが作られたクエリの中だけである。

テーブル名と同じように、ビューの名前はスキーマ内で一意でなくてはならない。かつ、ビュー定義はその中で自身を参照することはできない。これは上で述べたように呼び出されるまではビューは存在しないからだ。ただし、ビュー定義で他のビューを参照することはできる。入れ子になったビューは最終的には土台になっている基底テーブルまでさかのぼる必要がある。これは考えてみれば当たり前のことで、もしビューの中に1つも基底テーブルが含まれていなければ、いったい何を見ているのかわからない。

26.2

更新可能なビューと読み取り専用ビュー

基底テーブルと違って、ビューは更新可能か読み取り専用かを選ぶことができる。これは排他的な選択肢なのでどちらか一方しか選べない。挿入、更新、削除の操作は更新可能なビューと基底テーブルに対して行うことができる。これらは当然制約による整合性チェックの対象となる。読み取り専用ビューに対しては、これらすべての変更操作は行うことができない。ただ、もとになっている基底テーブルを変更することでビューの中身を変えることはできる。

更新可能なビューとは、基底テーブルの行と正確に1 対1 対応しているようなビューのことだ。ビューが変更を受けると、その変更は基底テーブルにも及ぶ。標準SQLでは、更新可能なビューであるためには次の条件を満たしている必要がある。

条件1　1つだけのテーブルから作られていること。

条件2　GROUP BY句を使っていないこと。

条件3　HAVING句を使っていないこと。

条件4　集約関数を使っていないこと。

条件5　計算列を使っていないこと。

条件6　UNION、INTERSECT、EXCEPTを使っていないこと。

条件7　SELECT DISTINCT句を使っていないこと。

条件8　ビューに含まれていない基底テーブルのすべての列はNULLを許可するかデフォルト値が指定されていること。これは基底テーブルに行を挿入する際、値を一意に決めるためである。

また、ビューは暗黙のうちに基底テーブルのキーを含むことになる。要するにビューの各行は基底テーブルの各行と厳密に1対1で対応することになるのだ。更新の中には、ビュー経由ではなく基底テーブルの参照整合性制約のCASCADEオプションによって行われるケースもある。

実際のところ、標準SQLにおける更新可能なビューの定義は、かなり制限がきつく極めて安全なものになっている（ここでいう安全とは、データの整合性を重視しているという意味だ）。やろうと思えば、参照整合性制約から得られる情報を調べて、更新可能なビューに許される条件をゆるめられるケースもあるだろう。実際そういうことをしている実装もあるのだが、まだ一般的ではない。

つまり、標準SQL の更新可能なビューの定義は、かなり条件を厳しくしてあり、実務に適用できるケースは少ない。条件が厳しいことの主な利点は、意味を考えなくてもビュー定義の構文を見れば更新可能にできるかどうか機械的に判断できるところだ。たとえば、次の2 つのビューは論理的には同値だが、一方は更新可能でもう一方はそうではない。これは純粋に構文上の違いによるものだ。

```
CREATE VIEW Foo1 -- 更新可能。キーを持っている！
AS SELECT *
     FROM Foobar
    WHERE x IN (1,2);

CREATE VIEW Foo2 -- 更新不可能！
AS SELECT *
     FROM Foobar
    WHERE x = 1
   UNION ALL
   SELECT *
     FROM Foobar
    WHERE x = 2;
```

Foo1は更新可能だがFoo2はそうではない。Foo2が`UNION ALL`を使っていることから、すぐにわかる。おそらく、複雑なビュー定義が更新可能かどうかを判断するのは（私も形式的な証明を行う自信はないが）、NP 完全問題ではないかと思う。だが、ここでは詳細に踏み込むのはやめて、更新可能なビューになりうるクエリのタイプをざっと挙げてみよう。この一覧は論文"VIEW Update Is Practical"（Goodman, 1990）から得たものだ。

1. 単一のテーブルからの射影（標準SQL）
2. 単一のテーブルからの制限と射影（標準SQL）
3. ビューのUNION
4. ビューの差集合演算
5. 1対1の結合
6. 1対1の外部結合
7. 1対多の結合
8. 1対多の外部結合
9. 多対多の結合
10. ある種のルールで変換された、またはコード化された列

トリガーを使うと、ビューに対する更新を基底テーブルに対する更新に振り替えることができる。`INSTEAD OF`トリガーは、ビューに対する挿入や更新、削除の操作をキャッチして、ビュー自身に対して更新を行う代わりに基底テーブルに更新を実施する機能だ。これによって、データベース設計者はビューに対する操作を完全に管理できるようになる。

26.3

ビューの分類

ビューは、その定義で使われるSELECT文と、使用目的によって分類される。ビューの大きな利点は、呼び出されたタイミングでそのときのデータに基づいた正しい結果を返すことだ。これと同じことを一時テーブルや計算列で行おうとすると、エラーになったり、データの同期が取れていなかったりする。

26.3.1　単一のテーブルからの射影および制限

実際にビューを使うケースの多くは、単一の基底テーブルに対する射影または制限である。これはデータにアクセス権限のない特定のユーザーから、見せたくない行や列を隠すことでセキュリティを管理する一般的な方法だ。こうしたビューは、たいていはインラインマクロに展開されて実装される。そのほうがオプティマイザも実行計画を立てる際に簡単だからだ。

26.3.2　計算列

ビューのよくある使い方として、1行内の複数列の合計値を計算するというものがある。たとえば、メーター単位で測定したいくつかの列を持つテーブルがあったとして、これをある種の計算を行って算出した値をビューに持てば、その計算ロジックは隠蔽される。

ここで注意してほしいのが、計算列を作る際にNULLが含まれていても大きな問題にならないことだ。たとえば、基本給（salary）と歩合（commission）の2 つの列を持つ給与テーブルを考えると、ある社員の給与総額を求めるには、次のようなビューが考えられる。

```
CREATE VIEW Payroll (emp_nbr, paycheck_amt)
AS
SELECT emp_nbr, (salary + COALESCE(commission, 0.00))
FROM Personnel;
```

社員の中には歩合が払われない者もいる。そうした社員のcommission列はNULLになる。だが、そういう場合もCOALESCE関数でNULLを0に変換してしまえば問題ない。

26.3.3 コード化された列

ビューのもう1つの有名な使い方として、コードをテキストや別のコードに変換するというものがある。これは、2つのテーブルを外部キーに基づいて結合するビューの特殊例である。たとえば、注文管理の業務を考えよう。テーブル内では部品は番号で管理されているが、画面に表示するときは部品名で表示したい。このようなときは、注文テーブルと在庫テーブルを結合したビューを作ればよい。

```
CREATE VIEW Screen (part_nbr, part_name, ...)
AS SELECT Orders.part_nbr, Inventory.part_name, ...
     FROM Inventory, Orders
    WHERE Inventory.part_nbr = Orders.part_nbr;
```

元の部品コードも画面に表示したいケースもあるだろうし、不要なケースもあるだろう。基本的には両方表示しておくほうが冗長ではあるが間違いがない。

コードを変換するための結合ビューは、2 つ以上のコード化列を変換するように拡張できる。その結果は、しばしば「スター型」のクエリになる。これは1 つのテーブルを中心として、外部キーでつなが

る多くのテーブルと結合するクエリである。ER図が星型の配置になることからこのような名前が付いている【訳注1】。

コードの変換で注意すべき点は、失われた値である。入力されたコードに対して適切な変換ができなかった場合、ビューにはそのコードの行は現れないか、外部結合が使われていればNULLの行が現れる。この問題を避けるためにも、CASCADEオプション付きの参照整合性制約が重要なのである。

26.3.4 グループ化されたビュー

グループ化されたビューはGROUP BY 句を伴うクエリを使うものだ。グループの各行は基底テーブルの複数行と対応する可能性があるので、必然的にこのタイプのビューは読み取り専用となる。グループ化されたビューは、通常レポーティング用の集約関数を含んでいる。また、この種のビューはSQLの表現力の弱さを補うのにも便利だ。たとえば、州別の最大の売上げを表示するビューを考えよう。クエリは次のようになる。

```
CREATE VIEW BigSales (state, sales_amt_total)
AS SELECT state_code, MAX(sales_amt)
     FROM Sales
    GROUP BY state_code;
```

SQL ではグルーピングの列をSELECT 句に含める必要は必ずしもないが、この場合はそうしておくほうがよい。

グループ化されたビューは、1 対多の関係を「水平展開」するのにも便利だ。たとえば、社員番号で一意になる社員テーブルと、社員番号と部下の社員番号の組み合わせで一意になる部下テーブルを考

えよう。ここから各社員がどれだけの部下を持っているかを示すレポートを作る。

```
CREATE VIEW DepTally1 (emp_nbr, dependent_cnt)
AS SELECT emp_nbr, COUNT(*)
     FROM Dependents
    GROUP BY emp_nbr;
```

次に、このビューと社員テーブルを単純に外部結合すれば、レポートのできあがりだ。ここで外部結合を使っているのは、部下を1人も持たない社員をNULLで表示するためだ。

```
SELECT emp_name, dependent_cnt
  FROM Personnel AS P1
         LEFT OUTER JOIN
           DepTally1 AS D1
    ON P1.emp_nbr = D1.emp_nbr;
```

26.3.5　UNION を使ったビュー

最近まで、UNIONまたはUNION ALLを使うビューは読み取り専用とされてきた。その理由はビューに対する更新を基底テーブルの1 行に対応させる方法がなかったからだ。UNIONは結果から重複行を排除する。UNIONとUNION ALLはどちらも、結果がどちらのテーブルのものを使っているのかを隠してしまう。こうしたビューは<view column list>を必要とする。なぜならUNION [ALL]が使われると、その結果に含まれる列は自身の名前を持たないからだ。だが理論

的には、2つの重複を持たないテーブルは更新可能であるべきだ。26.3.4 項で使ったサンプルをもう一度使えば、部下を1人も持たない

社員を割り当てるクエリはUNIONを使って次のようにも書ける。

```
CREATE VIEW DepTally2 (emp_nbr, dependent_cnt)
AS (SELECT emp_nbr, COUNT(*)
      FROM Dependents
     GROUP BY emp_nbr)
   UNION
   (SELECT emp_nbr, 0
      FROM Personnel AS P2
     WHERE NOT EXISTS (SELECT *
                         FROM Dependents AS D2
                        WHERE D2.emp_nbr = P2.emp_nbr));
```

こうするとこのレポートは、ビューと社員テーブルを単純に内部結合しただけのものになる。部下を1 人も持たない社員に対してはNULLの代わりに0 が割り当てられる。レポートを出力するクエリは次のようになる。

```
SELECT empart_name, dependent_cnt
  FROM Personnel, DepTally2
 WHERE DepTally2.emp_nbr = Personnel.emp_nbr;
```

OracleやDB2など主要なデータベースの最近のバージョンでは、UNIONを使ったビューに対する挿入、更新、削除がサポートされている【訳注2】。内部的には、パーティションを区切ることでテーブルを分離している。そのパーティションはテーブルの中身によって分割のキーを決めているが、一番よくある切り分け方は時間の列を使う。つまり、各パーティションは日付の範囲に応じて分割されることになる。こうすることの目的は、パーティションに並列アクセスを行って、クエリのパフォーマンスを改善することである。

ただし、ここにはトレードオフもある。たとえば、DB2ではUNION ALLを使ったビューの各基底テーブルに対して、与えられた行をすべ

て挿入しようとする。その上でいくつのテーブルがその行を受け入れたかをカウントするのだ。このプロセスをビュー全体に適用すると、1テーブルずつ処理を行い、結果を数える必要がある。

1. もし厳密に1テーブルだけが行を受け入れれば、挿入は成功とみなされる。
2. 1テーブルも行を受け入れなければ、「対象なし」のエラーが返される。
3. 2つ以上のテーブルが行を受け入れた場合は、「あいまいな対象」のエラーが返される。

INSTEAD OFトリガーを使えば、あたかも単一のテーブルを扱っているのと同じような効果が期待できるが、それでも少しやっかいなところがある。いま、A、B、Cの3つのテーブルを考えよう。テーブルAとBは互いに重なる行を持っている。だがCは、A、Bと重なる行を持っていない。そうすると、Cに対しては常に挿入できるが、AとBに対しては重複行のせいで保証されない。2000 年問題がクローズアップされていた頃、カレンダーの日付によるこの種の問題に出くわしたことがある。テーブルCは会計の四半期ごとに区切られたテーブルで、閏年をうまく計算できなかった。なぜなら、四半期の1つが2月の最終日で終わっていたからだ。

これと似ているが少し違うアプローチとしては、DDLで明示的なパーティショニングのルールを宣言してやることだ（ただし、これをやると実装依存の構文を使うことになる）。Oracleの例で示すと次のようになる。

```
CREATE TABLE Sales
(invoice_nbr INTEGER NOT NULL PRIMARY KEY,
 sale_year INTEGER NOT NULL,
 sale_month INTEGER NOT NULL,
 sale_day INTEGER NOT NULL)
     PARTITION BY RANGE (sale_year, sale_month, sale_day)
    (PARTITION sales_q1 VALUES LESS THAN (1994, 04, 01)
       TABLESPACE tsa,
     PARTITION sales_q2 VALUES LESS THAN (1994, 07, 01)
       TABLESPACE tsb,
     PARTITION sales_q3 VALUES LESS THAN (1994, 10, 01)
       TABLESPACE tsc,
     PARTITION sales q4 VALUES LESS THAN (1995, 01, 01)
       TABLESPACE tsd);
```

くどいようだが、これは実装依存の構文である。実装は常に物理的なデータベースを扱わなければならないので、論理モデルの世界だけで完結するわけにはいかないのだ。

26.3.6　結合を使ったビュー

理論的に見ても、結合を使ったビューは常に更新可能にはならない。ビューの中で結合を使う主な理由の1 つは、1 対多、または多対多の関係を「水平展開」することだ。このような関係においては、ビューの中の1 行を、結合で使われている「多」のほうのテーブルの1 行にマッピングすることができない。また、結合について当てはまることは、結合を使ったビューについても同様に当てはまるので、ここでは詳細を論じない。結合について解説した第25章を参照してほしい。

26.3.7　入れ子のビュー

しばしば熟練のSQLプログラマにも見落とされている観点だが、ビューの上にビューを作ることができる。唯一の制限は、ビューの中で同一のビューを見てはいけないという循環参照のルールだけだ。

入れ子のビューにおける問題の1つが、複数の更新可能なビューが同じ基底テーブルを同時に参照できてしまうことだ。もしこのようなビューが他のビューに含まれた場合、上位のビューが変更されたときに何が起こるかを把握するのは容易なことではない。例として、次のような2つのキーを持つテーブルを考えよう。

```
CREATE TABLE Canada
(english INTEGER NOT NULL UNIQUE,
 french INTEGER NOT NULL UNIQUE,
 eng_word CHAR(30),
 fren_word CHAR(30));

INSERT INTO Canada
VALUES (1, 2, 'muffins', 'croissants'),
       (2, 1, 'bait', 'escargots');

CREATE VIEW EnglishWords
AS SELECT english, eng_word
     FROM Canada
    WHERE eng_word IS NOT NULL;

CREATE VIEW FrenchWords
AS SELECT french, fren_word
     FROM Canada
    WHERE fren_word IS NOT NULL;
```

さて、エスカルゴを食べてみて、これは餌（bait）ではなく立派な前菜（appetizer）だと認識を改めたとしよう。そこで次のような更新を行うことにした。

```
UPDATE EnglishWords
   SET eng_word = 'appetizer'
 WHERE english = 2;
```

その後、フランス人のユーザーがハギスを食べて、その経験から新しい行を挿入しようと考えた。

```
UPDATE FrenchWords
   SET fren_word = 'Le swill'
 WHERE french = 3;
```

この更新は基底テーブルに(NULL, 3, NULL, 'Le swill')という行を作る。というのも、FrenchWordsビューから見ると、EnglishWordsビューの列を取得する方法がないからだ。同様に、EnglishWords ビューのユーザーはこのフランス人の経験を翻訳した(3, NULL, 'Haggis', NULL)という行を作ることができる。だがどちらのユーザーも、この2 つの行を有意味なデータとして統合することはできない。

さらに、行を削除すると、データを壊すことになる。たとえば、フランス人がテーブルから'croissants'を削除すると、EnglishWordsビューから'muffins'も消えてしまう。

26.4

データベースエンジンにおけるビューの扱い方

標準SQLでは、システムスキーマのテーブルにビュー定義のテキストが保持されることになっている。いくつかの実装は、ビューから基底テーブルまでの入れ子の構造を把握できるツリー図を出力する便利な機能を持っている。読者も自分の製品をチェックして、そのような便利なライブラリがないか調べてみるといい。ビューの扱い方にはいくつかの方法があり、システムはそれらを組み合わせて使うことが多い。その方法は、主に実体化とインライン展開に分類される。

26.4.1　ビューの列リスト

<view column list>は必須ではない。定義でこれが指定されなかった場合、ビューはクエリから列名を引き継ぐ。<view column list>に含まれる列数は、クエリが返す列数と同じでなければならない。クエリが複数の列に同じ名前を付けていた場合、あいまいさを解決するために<view column list>は必須になる。当然だが、<view column list>において列名の重複は許されない。

26.4.2　ビューの実体化

実体化とは、ビュー定義に含まれる適切な列名と適切なデータ型を持ったワークテーブルを作ることである。このテーブルには、ビュー定義の本体に記述されているSELECT文の結果が挿入される。

ビューを実体化する際に、実際に物理テーブルを作るかどうかの判断は、標準SQLでは実装に任されている。ただし、クエリからアク

セスされる際は、ビューはあたかもテーブルであるかのように振る舞わなくてはならない。実体化されたビューが更新可能ではない場合、自動的に、基底テーブルはいかなる不適切な変更も受けないため、完全に正しいことが保証される。

ビューを実体化する方法はさまざまだが、テーブルを作ったり、クエリから挿入するなど、データベースエンジンが持っている既存の内部的なプロシージャを使うのであれば、データベースにとっては簡単な仕事である。一方、実体化の欠点は以下のとおりだ。

- 大きなビューを作るときには構築もアクセスもあまり速くない。
- 追加のストレージ容量が必要となる。
- 基底テーブルに存在するインデックスを利用できない。
- たいていは新しいテーブルにインデックスを作ることができない。

ただし、実体化がうまくはまるタイプのビューもある。たとえば、内部的にソートを行うようなビューである。`SELECT DISTINCT`、`UNION`、`GROUP BY`、`HAVING`を使うクエリは、重複行を削除したり、グループを作るためにソートを行うよう実装されていることが多い。こうしたビューを実体化しておくことは、パフォーマンス改善のうまいやり方である。

いくつかの製品では、実体化をユーザーが管理する機能を用意していることもある。ベンダーの用語はさまざまだが「スナップショット」や「結果セット」などと呼ばれている。前者はタイムスタンプ付きの実体化されたテーブルであり、後者はフロントエンドのア

プリケーションに渡すための実体化されたテーブルである。こうしたベンダー独自の用語については、読者の使っている製品のマニュアルを調べてほしい。

26.4.3　インライン展開

データベースがビューを扱う2 つ目の方法は、CREATE VIEW 文のテキストを保存し、それをSELECT、INSERT、UPDATE、DELETEといったSQL文のパースを行う際、解析対象のツリーに追加することだ。こうすることで、オプティマイザは、ビュー定義を最終的なクエリの実行計画に統合することができる【訳注3】。たとえば、特定の部署の売上げを保持する次のようなビューを作るとする。

```
CREATE VIEW SalesDept (dept_name, city_name, ...)
AS SELECT 'Sales', city_name, ...
     FROM Departments
    WHERE dept_name = 'Sales';
```

次に、このビューを使って、ニューヨーク市の情報を求めるクエリを実行する。

```
SELECT *
  FROM SalesDept
 WHERE city_name = 'New York';
```

するとパーサは、ビューをテキストとしてFROM 句に展開する。実質的に、実行されるクエリはこういう形を取る。

```
SELECT *
  FROM (SELECT 'Sales', city_name, ...
          FROM Departments
         WHERE dept_name = 'Sales') AS SalesDept (dept_name, city_name, ...)
 WHERE city_name = 'New York';
```

このクエリを見たオプティマイザは、次のように「展開」してクエリをシンプルな形に同値変換するだろう。

```
SELECT *
  FROM Departments
 WHERE (dept_name = 'sales')
   AND (city_name = 'New York');
```

これは賢いアプローチだと思うかもしれない。だが昔のデータベースではなかなかうまくいかないこともあった。たとえば、私がDB2の古いバージョンで遭遇した問題は、次のようなものだった。これは、スーザン・フォンブラックから寄せられたサンプルだ。

長い識別番号の列（long_id）と、それに紐付けられた数値列を格納するテーブルがあると想像してほしい。識別番号の長さは40 文字近くにも及ぶが、幸いなことに、最初の6 文字だけでも一意になる（ただ、どのみち画面に表示するために全長を取得しなければならないのだが）。やりたいことは、識別子の最初の6文字でグルーピングされた帳票を作ることだ。まっさきに思いつくのは、SUBSTRING関数で文字を切り出す解だろう。

```
CREATE TABLE TableA
(long_id CHAR(40) PRIMARY KEY,
 amt1    INTEGER);
```

TableA：テーブルA　　long_id：長いID　　amt1：合計1

```
SELECT SUBSTRING(long_id FROM 1 FOR 6), SUM(amt1)
  FROM TableA
 GROUP BY id;
```

だがこれは適法なSQL ではないため、うまく動かない。理由は、SELECT 句の列とGROUP BY 句の集約キーが一致しないからである。かといって、「GROUP BY SUBSTRING(long_id FROM 1 FOR 6)」としてもダメだった。当時のDB2では、GROUP BY句に書けるのは単純な列だけだったのだ（いまならこのやり方で問題ない）。「GROUP BY 1」という列位置を指定するやり方もうまくいかない。列位置を指定できるのは、UNIONを使った文といくつかの製品におけるORDER BY句だけである（かつ、列位置を指定する機能は、標準SQLでは将来的に廃止される機能のリストに挙げられている）。

そこで私は、やむをえず、次のようなビューを作ることにした。

```
CREATE VIEW BadTry (short_id, amt1)
AS SELECT SUBSTRING(long_id FROM 1 FOR 6), amt1
     FROM TableA;
```

あとは、このビューのshort_id列をGROUP BY句のキーに使うクエリを書けばいい――はずだった。ところが、これは適法なSQLであるにもかかわらず、昔のDB2では動かなかった。どうもコンパイラは、BadTryビューをFROM句にインライン展開しているようだった。そうすると、結局最初に考えたクエリと同じ理由で、SELECT 句とGROUP BY 句の列が一致しないためエラーになる。結局このときは、ビューを実体化することでことなきを得た。

同じ問題は、内部的にソートを行う演算――たとえば、SELECT DISTINCT 句、UNION、HAVING句でも起きる。

短い識別番号でもキーになることから、次のようなビューを作ることができる。

```
CREATE VIEW Shorty (short_id, amt1)
AS SELECT DISTINCT SUBSTRING(long_id FROM 1 FOR 6), amt1
     FROM TableA;
```

レポート作成のクエリはこうだ。

```
SELECT short_id, SUM(amt1)
  FROM Shorty
 GROUP BY short_id;
```

もちろん、この解は標準SQLで導出テーブルを使って書くこともできる。

```
SELECT *
  FROM (SELECT SUBSTRING(long_id FROM 1 FOR 6) AS short_id,
               SUM(amt1)
          FROM TableA
         GROUP BY SUBSTRING(long_id FROM 1 FOR 6)) AS Foo;
```

サブクエリ内のSUBSTRING関数の列に、short_idという列名を付けることで、外側のクエリを評価するときに、パーサはこれを認識することができる。

26.4.4　ポインタ構造

最後に、システムがビューを扱う際の方法論として特別なデータ構造を使うことがあるので、それを紹介しよう。それは通常、ビュー定義の中に含まれる基底テーブルへのポインタ配列である。これは更新可能なビューを扱うには便利な方法だ。なぜなら、基底テー

ブルの更新対象の行はポインタチェーンの先をたどればわかるからだ。アクセスも極めて高速である。

ポインタ構造を使ったアプローチでは、基底テーブルに存在するインデックスを使うことは簡単ではない。しかし、ポインタ構造は、いくつかの制限付きだが、インデックスとして実装することもできる。さらに、複数のテーブルを使ったビューをポインタ構造として構築することもでき、結合に含まれるテーブルの関係する行に直接アクセスすることができる。ただし、この構造は極めて実装依存なので、一般論を述べることはできない。

26.4.5　インデックスとビュー

ビュー自身にインデックスを付けることはできない点には注意が必要だ。ただし、いくつかの実装においては、ビューは基底テーブルからインデックスを引き継ぐことができる。テーブル同様、ビューは行の順序を持たないが、実装依存の構文を使えば、ビューの行を順序づけることができる。特に、ビューの中で`ORDER BY`句の使用を許可しているものが典型的だ（標準SQLでは、`ORDER BY`句を書けるのはカーソルに対してだけである）。`ORDER BY`はソートを強制し、ビューをワークテーブルとして実体化させることもある。SQLエンジンがテーブル全体を読み込まなければならないケースでは、あらかじめビューをソートしておくことで、特定のクエリを隠蔽したり高速化したりできる。ただし、こうした実装依存のトリックを使った場合の結果については保証の限りではない。

26.5

WITH CHECK OPTION 句

ビューにWITH CHECK OPTIONが指定されていた場合、ビュー定義のSELECT文の検索条件に該当しない行の挿入や更新はできなくなる。逆に言うと、それ以外の変更はビュー経由で行うことができる。このオプションは、特定の製品が更新可能なビューをサポートしているかどうかを調べる手っ取り早い方法でもある。このオプション付きでビューを作ることができれば、更新可能ビューを何らかの形でサポートしているということがわかる【訳注4】。

WITH CHECK OPTION は、SQL-89 から存在するオプションだが、その後拡張され、<levels clause>も指定できるようになった。LEVEL句がない場合は、暗黙にCASCADEDが前提とされる。いま、次のような定義のビューを考えよう。

```
CREATE VIEW V1
AS SELECT *
     FROM Foobar
    WHERE col1 = 'A';
```

このビューを次のUPDATE文で更新する。

```
UPDATE V1 SET col1 = 'B';
```

このUPDATEは、何の問題もなく実行されるが、実行前にV1に含まれていた行は、もう一度V1を検索したときには、すべて消え去る。更新によって、WHERE 句を満たす行がなくなってしまったからだ！

同様に、次のINSERT文もまったく問題ないが、V1には行は追加されない。

```
INSERT INTO V1 VALUES (col1 = 'B');
```

WITH CHECK OPTIONなしで作られたビューは、このようにWHERE句の条件に合致しない行であっても更新することができる。これはこれで便利である。たとえば、'to be done'というステータスコードを持つ作業待ちのタスクを表示するビューを作ったとする。そして、作業が終わったら、コードを'finished'に変える。すると、ビューからはデータがきれいに消え去る。重要なポイントは、WHERE句の条件は、ビューが呼び出されるときにだけチェックされ、更新の際にはまったくチェックされないことだ。

WITH CHECK OPTIONは、更新時にもWHERE句の条件をチェックさせるオプションである。もし新しく追加された行、または変更された行がWHERE 句の条件を満たさなかった場合、その変更は拒否され、ビューの状態は変わらない。たとえば、先のUPDATE文はエラーメッセージを表示するだろう。WITH CHECK OPTIONは、たとえば、給与が3 万ドル以下の社員を表示するようなビューに付けて、このビューに含まれる社員の給与が決して3万ドルを超えないような制限をかけたい場合に使う。

WITH CHECK OPTIONは、CHECK 制約とは少し違う。違いが起きるのはNULL の場合だ。例を使って見てみよう。

```
CREATE TABLE Foobar
(col_a INTEGER);

CREATE VIEW TestView (col_a)
AS
SELECT col_a FROM Foobar WHERE col_a > 0
WITH CHECK OPTION;

INSERT INTO TestView VALUES (NULL); -- これは失敗する！

CREATE TABLE Foobar_2
(col_a INTEGER
     CHECK (col_a > 0));

INSERT INTO Foobar_2(col_a) VALUES (NULL); -- これは成功する！
```

WITH CHECK OPTIONはTRUEにならないと更新を受け付けないが、CHECK制約は、TRUE以外にUNKNOWNの更新も許可する。ここでもNULLはやっかいな問題を引き起こすのだ。

WITH CHECK OPTION に付与することのできる<levels clause>句は、CASCADED またはLOCAL のどちらかである。もし<levels clause>が明示されていなければ、暗黙にCASCADED とみなされる。CASCADEDが指定されると、システムはビューそのものに含まれているWHERE句以外にも、そのビューの中で使われているすべてのビューのWHERE句もチェックしにいく。一方、LOCALの場合は、そのビューのWHERE句だけがチェックされる。

社員の給与を管理するテーブルをもとにした2 つのビューを使って考えてみよう。ただし、Mediumpayビューは、Lowpayビューの上に定義されている。

```
CREATE TABLE Personnel
(emp_nbr CHAR(4) PRIMARY KEY,
 salary  INTEGER);
```

Personnel：社員テーブル　　emp_nbr：社員番号　　salary：給与

```
emp_nbr  salary
===============
0001     90
0002     120
0003     400
```

```
CREATE VIEW Lowpay
AS SELECT *
     FROM Personnel
    WHERE salary <= 250;

CREATE VIEW Mediumpay
AS SELECT *
     FROM Lowpay
    WHERE salary >= 100;
```

どちらのビューもWITH CHECK OPTIONを持っていないと仮定しよう。すると、Mediumpayに含まれる社員の給与を一律1000ドル昇給させる更新は、Lowpayに対するチェックなしで適用される。Lowpay に対する変更は、そのまま最下層のPersonnel テーブルに反映される。次にMediumpayが参照されるときは、ビューが再構築され、Mediumpayからは全社員が消える。

次に、Mediumpay だけにWITH CASCADED CHECK OPTION を付けてみる。すると、先ほどのUPDATE 文は失敗する。Mediumpay 自身は、社員の給与が上がっても何の問題もないのだが、Lowpayからは行が消えることになる。したがってMediumpayはこの更新を拒否する。ただし、Mediumpay だけがWITH LOCAL CHECK OPTION を持っていれば、UPDATE は成功する。Mediumpayは社員の給与が増えても何の問題もないからだ。Mediumpayは更新をLowpayに渡

し、LowpayはPersonnelテーブルに更新を渡し、実際の更新が行われる。

もし両方のビューがWITH CASCADED CHECK OPTIONを持っていたらどうなるだろう。今度は、両方のビューのWHERE 句の条件がともに満たされなければならない。つまり、Personnelテーブルに対して変更または挿入可能な社員は、給与が100ドルから250ドルの範囲にある者に限られる。

この問題は非常に複雑だ。一例として、IBM のネルソン・マットスが書いたANSI X3H2 の論文（Celko, 1993）に載っているサンプルを使おう。まず5層からなるV1からV5のビューを作る。構文の中にあるxxとyyは、CASCADEDまたはLOCALを意味する。基底テーブルT1は列c1 からc5を持っている。初期値としてすべての列に10をセットしておく。

```
CREATE VIEW V1 AS SELECT * FROM T1 WHERE (c1 > 5);

CREATE VIEW V2 AS SELECT * FROM V1 WHERE (c2 > 5)
WITH xx CHECK OPTION;

CREATE VIEW V3 AS SELECT * FROM V2 WHERE (c3 > 5);

CREATE VIEW V4 AS SELECT * FROM V3 WHERE (c4 > 5)
WITH yy CHECK OPTION;

CREATE VIEW V5 AS SELECT * FROM V4 WHERE (c5 > 5);
```

ここでV5のいずれかの列を0に更新する。すると下表に示すような結果の違いが現れる。ここでSは更新の成功、Fは更新の失敗を意味する。

xx/yy	c1	c2	c3	c4	c5
cascade/cascade	F	F	F	F	S
local/cascade	F	F	F	F	S
local/local	S	F	S	F	S
cascade/local	F	F	S	F	S

この表を理解するには、一番最後の行を見てみることだ。xx=CASCADED、yy=LOCALの場合、V5経由で列c1を0に更新するのは失敗する。だが一方、c5に対する更新は成功する。この更新が成功するということはV5からその行は消えてしまうということだ。

V5経由でc1を0にしようとする際、V5自身は何のチェックも行わない。したがって、変更された行は即座にV4 へ送られる。V4はWITH LOCAL CHECK OPTIONを持っているが、列c1は関与していない。したがってV4も行をV3に渡す。V3はWITH CHECK OPTIONを持っていないので、行は即座にV2に送られる。V2はWITH CASCADED CHECK OPTIONを持っているので、行をV1に渡して結果を待つ。V1は基底テーブルT1の上に作られており、条件「c1>5」を持っている。更新はこの条件に違反するので、V1は基底テーブルに対する更新を拒否する。こうしてV5に対する更新は失敗する。

今度は次のような更新を考えよう。

```
UPDATE V5 SET c3 = 0;
```

V5はWITH CASCADED CHECK OPTION を持っていないので、先ほどと同様、変更された行は即座にV4 へ送られる。V4はWITH LOCAL CHECK OPTIONを持っているが、列c3は関与していないので行をV3 に送り結果を待つことはない。V3 はc3 に対する条件を持っているが、WITHLOCAL CHECK OPTION を持っていないので、V2 およびV1に送られる。どちらもc3に対する条件を持っていないので、基底テーブルの更新が成功する。次にV5が呼び出されたときは、V3の条件に合致しなくなっているので、行は表示されなくなる。ここまではよいだろう。本当の問題が起きるのは、一度に複数の列を更新するUPDATE 文の場合だ。たとえば、次のような更新は <levels clause>句のあらゆる可能な組み合わせに対して失敗する。

```
UPDATE V5 SET c1 = 0, c2 = 0, c3 = 0, c4 = 0, c5 = 0;
```

標準SQLは、条件の集合を入れ子のレベルによって引き継がれると定義している。いまのサンプルであれば、次のような暗黙のテストがそれぞれのビュー定義に追加されることになるのだ。

```
local/local
V1 = none
V2 = (c2 > 5)
V3 = (c2 > 5)
V4 = (c2 > 5) AND (c4 > 5)
V5 = (c2 > 5) AND (c4 > 5)

cascade/cascade
V1 = none
V2 = (c1 > 5) AND (c2 > 5)
V3 = (c1 > 5) AND (c2 > 5)
V4 = (c1 > 5) AND (c2 > 5) AND (c3 > 5) AND (c4 > 5)
V5 = (c1 > 5) AND (c2 > 5) AND (c3 > 5) AND (c4 > 5)

local/cascade
V1 = none
V2 = (c2 > 5)
V3 = (c2 > 5)
V4 = (c1 > 5) AND (c2 > 5) AND (c4 > 5)
V5 = (c1 > 5) AND (c2 > 5) AND (c4 > 5)

cascade/local
V1 = none
V2 = (c1 > 5) AND (c2 > 5)
V3 = (c1 > 5) AND (c2 > 5)
V4 = (c1 > 5) AND (c2 > 5) AND (c4 > 5)
V5 = (c1 > 5) AND (c2 > 5) AND (c4 > 5)
```

26.5.1　CHECK 制約としてのWITH CHECK OPTION

スイスのオラクル社でインストラクターをしているロタール・フラッツは、OracleではサブクエリをCHECK制約に入れられないし、トリガーを使おうにも、当のトリガーを起動した文が変更中のテーブルには触れられないが、WITH CHECK OPTION 付きのビューを使えば、サブクエリを使った制約を施行可能なことに気づいた。

たとえば、他の客が宿泊中、または予約している日には部屋を予約できないというルールを持つホテルの予約システムを考えよう。次のように別テーブルを参照する制約を付与したい。

```
CREATE TABLE Hotel
(room_nbr INTEGER NOT NULL,
 arrival_date DATE NOT NULL,
 departure_date DATE NOT NULL,
 guest_name CHAR(30) NOT NULL,
     CONSTRAINT schedule_right
       CHECK (H1.arrival_date <= H1.departure_date),
     CONSTRAINT no_overlaps
       CHECK (NOT EXISTS
                  (SELECT *
                     FROM Hotel AS H1, Hotel AS H2
                    WHERE H1.room_nbr = H2.room_nbr
                      AND H2.arrival_date < H1.arrival_date
                      AND H1.arrival_date < H2.departure_date)));
```

Hotel：ホテルテーブル　room_nbr：部屋番号　arrival_date：到着日
departure_date：出発日　guest_name：宿泊客

schedule_right 制約は、サブクエリを持っていないので問題ない。しかし、no_overlaps 制約は、ほとんどの製品で動かないだろう。そこで、この制約をテーブルからはずし、これをWHERE句の条件に持つビューを作り、WITH CHECK OPTIONを付ける。

```
CREATE TABLE Hotel
(room_nbr INTEGER NOT NULL,
 arrival_date DATE NOT NULL,
 departure_date DATE NOT NULL,
 guest_name CHAR(30) NOT NULL,
 PRIMARY KEY (room_nbr, arrival_date),
 CHECK (departure_date >= arrival_date));

CREATE VIEW HotelStays
(room_nbr, arrival_date, departure_date, guest_name)
AS SELECT room_nbr, arrival_date, departure_date, guest_name
     FROM Hotel AS H1
    WHERE NOT EXISTS
                (SELECT *
                   FROM Hotel AS H2
                  WHERE H1.room_nbr = H2.room_nbr
                    AND H2.arrival_date < H1.arrival_date
                    AND H1.arrival_date < H2.departure_date)
     WITH CHECK OPTION;
```

すると、次のような行の挿入は、3 番目のINSERT 文がWITH CHECK OPTION 違反となり失敗する。これは、施行したかった制約と同じ効果である。

```
-- HotelStaysビューにはCoe氏だけが表示される
INSERT INTO HotelStays
VALUES (1, '2008-01-01', '2008-01-03', 'Coe');

-- HotelStaysビューにはCoe氏のほか、Doe氏も表示される
INSERT INTO HotelStays
VALUES (1, '2008-01-03', '2008-01-05', 'Doe');

-- Roe氏の予約（INSERT）は失敗しする
INSERT INTO HotelStays
VALUES (1, '2008-01-02', '2008-01-05', 'Roe');
```

26.6

ビューの削除

ビューは、テーブルや他のスキーマオブジェクトと同様、スキーマから削除することができる。標準SQLの構文は次のとおりだ。

```
DROP VIEW <table name> <drop behavior>
<drop behavior>:: = [CASCADE | RESTRICT]
```

<drop behavior>句はSQL-86 にはなかったので、ベンダーは独自の動作を実装していた。ビューを保存する通常の方法は、スキーマレベルのテーブルにビューの名前、テキスト、その他の情報を格納することだ。ビューを削除するときは、SQL エンジンはこのスキーマテーブルから該当の行を削除する。

CASCADEオプションは削除対象のビューを使っているほかのすべてのビューを見つけ、それらも合わせて削除する。もし、RESTRICTが指定されていた場合、削除対象のビューに依存するビューがあった場合はビューを削除することはできない。こうした動作を可能にするために、スキーマテーブルは少し複雑な作りをしている。ビューの上に作られたビューを参照しようとしたときに、もとになっているビューが削除されていたり、基底テーブルが削除されていてビューにアクセスできなかったという経験は多くの人が持っているだろう。

時として問題になるのが、ビューのもとになっている基底テーブルを削除したときに、ビュー自身は残ってしまうことだ。

```
CREATE TABLE Foobar (col_a INTEGER);

CREATE VIEW TestView
AS SELECT col_a
     FROM Foobar;
DROP TABLE Foobar; -- 基底テーブルの削除
```

この例では、DROP TABLE文にCASCADEオプションを付けていない限り、ビュー定義のテキストはスキーマ内に残存する。したがって、テーブルと列名を使いまわして別のテーブルを作ろうとすると、ビュー定義は呼び出されたときに解決されるので、元のテーブルとは違うテーブルを基底テーブルとすることになってしまう。

```
CREATE TABLE Foobar
(foo_key CHAR(5) NOT NULL PRIMARY KEY,
 col_a REAL NOT NULL);

INSERT INTO Foobar VALUES ('Celko', 3.14159);
```

TestView.col_a のデータ型は、テーブルが再作成されたことで、INTEGER からREAL に変わってしまった。これは潜在的なセキュリティリスクだし、標準SQL違反である。だが、こういう仕様が実装に残っていることには注意してほしい。

26.7

ビューと一時テーブルのどちらを使うかの判断についてのヒント

場合によっては、どちらを使うべきか判断の余地がない場合もある。標準SQLでは、ユーザーはビューも一時テーブルも作ることはできないからだ。任意のスキーマオブジェクトを作ることができるのは、DBAだけでありユーザーは与えられたものを使うしかない。とはいえ、それぞれの構造と、どちらがどんな状況に適しているかを知っておく必要はある。

26.7.1　ビューの使い方

まず、ビューをあまり深く入れ子にしてはいけない。複数の階層を構築するオーバーヘッドは時間がかかるし、ビューを実体化するストレージの容量も高くつく。また、複雑な入れ子は保守もたいへんだ。ビューの正しい使い方は、既存のクエリの中でよく使われるサブクエリや式を1か所で管理することだ。

ビューの主な使用目的の1つはセキュリティである。DBAはセキュリティ認証と、ビューを組み合わせることで、特定のユーザーから見せたくない行や列を隠すことができる。

もう1 つのセキュリティのトリックは特別なユーザー識別子、またはセキュリティレベルの識別子の列を、テーブルに追加することだ。ビューでこの列を隠し、ユーザーには権限的に見ることのできるデータだけを与えるというやり方だ。このとき起こりうる問題は、ユーザーが他のテーブルの制約に違反するような更新を、ビューに対して行おうとすることだ。ユーザーの更新が失敗すると出力

されるメッセージには、本来ユーザーには見せたくなかったセキュリティシステムに関する情報が含まれることがある。

ビューを使うときのベストプラクティスは、ユーザーがデータベースをどういうふうに見ようとしているかを考え、あたかもデータベースがユーザーの使うアプリケーションだけのために設計されているかのようにビューの集合を与えることだ。

26.7.2 一時テーブルの使い方

GLOBAL TEMPORARY TABLEは、ユーザー間でデータを受け渡すために使うことができる。これはビューではできないことだ。一方、LOCAL TEMPORARY TABLEにも大きな2つの利点がある。まず1つは非常に時間のかかる複雑なクエリの結果をロードしておけば、その結果をセッション内で何度でも使うことができる。これは大きくパフォーマンスを改善する。また、これによって、その複雑なクエリが使っている基底テーブルを利用しているほかのユーザーを、ロックで待たせることが少なくなる。

コッドは、かつてスナップショットの概念を論じたことがある。これは特定の時刻におけるテーブルのイメージである。だがここで重要なのは、その時刻を正確に知ることである。一時テーブルにCURRENT TIMESTAMPをデフォルト値として付け加えておけば、コッドの考えたスナップショットを作ることができる。

一部の製品では、ユーザーは一時テーブルを好きなときに作ることができる。こうしたテーブルはセッションの間しか持続せず、作ったユーザーにしか見ることができない場合もある。また、こうしたテーブルはインデックス、制約、ビュー、参照整合性制約といっ

た、通常のテーブルに宣言できるものを宣言できないこともある。要するに、こうしたテーブルは、作ったユーザーにとっての純粋な「使い捨てテーブル」なのだ。中には、別のユーザーからもアクセス可能なグローバル一時テーブルを作ることのできる製品もある。だが、繰り返すがこれはANSI/ISO のモデルではない。

26.7.3　ビューを使ったテーブルの展開

セールスマンの月々の売上げデータを列として持つテーブルがあるとしよう（各月の列は将来的に値が埋められる想定で、NULLを許可しておく）。

```
CREATE TABLE AnnualSales1
(salesman CHAR(15) NOT NULL PRIMARY KEY,
 jan DECIMAL(5,2),
 feb DECIMAL(5,2),
 mar DECIMAL(5,2),
 apr DECIMAL(5,2),
 may DECIMAL(5,2),
 jun DECIMAL(5,2),
 jul DECIMAL(5,2),
 aug DECIMAL(5,2),
 sep DECIMAL(5,2),
 oct DECIMAL(5,2),
 nov DECIMAL(5,2),
 "dec" DECIMAL(5,2)); -- decは予約語のため、ダブルクォートが必要
```

AnnualSales1：年間売上げテーブル1　　salesman：セールスマン　　jan～dec：1月～12月

このテーブルを「展開」して、次のような形のテーブルにデータを挿入したい。

```
CREATE TABLE AnnualSales2
(salesman CHAR(15) NOT NULL,
 month_name CHAR(3) NOT NULL
     CONSTRAINT valid_month_abbrev
       CHECK (month_name IN ('Jan', 'Feb', 'Mar', 'Apr',
                             'May', 'Jun', 'Jul', 'Aug',
                             'Sep', 'Oct', 'Nov', 'Dec')),
 sales_amount DECIMAL(5,2) NOT NULL,
     PRIMARY KEY(salesman, month_name));
```

AnnualSales2：年間売上げテーブル2　　salesman：セールスマン　　month_name：月名　　sales_amount：売上高

ここで使うトリックは、各月に整数を割り当てたビューを作ることだ。

```
CREATE VIEW NumberedSales
AS SELECT salesman,
    1 AS M01, jan,
    2 AS M02, feb,
    3 AS M03, mar,
    4 AS M04, apr,
    5 AS M05, may,
    6 AS M06, jun,
    7 AS M07, jul,
    8 AS M08, aug,
    9 AS M09, sep,
   10 AS M10, oct,
   11 AS M11, nov,
   12 AS M12, "dec"
 FROM AnnualSales1;
```

このビューと連番を持つ補助テーブルA（VALUES句を使ったテーブル構築子で作る）を使えば、展開したビューは簡単に作れる。

```
CREATE VIEW AnnualSales2 (salesman, month, sales_amt)
AS SELECT S1.salesman,
         (CASE WHEN A.nbr = M01 THEN 'Jan'
               WHEN A.nbr = M02 THEN 'Feb'
               WHEN A.nbr = M03 THEN 'Mar'
               WHEN A.nbr = M04 THEN 'Apr'
               WHEN A.nbr = M05 THEN 'May'
               WHEN A.nbr = M06 THEN 'Jun'
               WHEN A.nbr = M07 THEN 'Jul'
               WHEN A.nbr = M08 THEN 'Aug'
               WHEN A.nbr = M09 THEN 'Sep'
               WHEN A.nbr = M10 THEN 'Oct'
               WHEN A.nbr = M11 THEN 'Nov'
               WHEN A.nbr = M12 THEN 'Dec'
               ELSE NULL END),
         (CASE WHEN A.nbr = M01 THEN jan
               WHEN A.nbr = M02 THEN feb
               WHEN A.nbr = M03 THEN mar
               WHEN A.nbr = M04 THEN apr
               WHEN A.nbr = M05 THEN may
               WHEN A.nbr = M06 THEN jun
               WHEN A.nbr = M07 THEN jul
               WHEN A.nbr = M08 THEN aug
               WHEN A.nbr = M09 THEN sep
               WHEN A.nbr = M10 THEN oct
               WHEN A.nbr = M11 THEN nov
               WHEN A.nbr = M12 THEN "dec"
               ELSE NULL END)
     FROM NumberedSales AS S1
          CROSS JOIN
            (VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12))
                 AS A(nbr);
```

導出テーブルを使えば、この2つのクエリを1つにまとめることもできる。

このテクニックは、言ってみれば列を値に変換する方法であり、その観点から見れば極めて非リレーショナルである。だが、レポート作成においては、こうした技術はとても便利だ。一時テーブルと比較した場合のビューの利点は、基底テーブルのデータが変わればビューのデータも自動的に最新化されることである。一時テーブルを使う際は、常に同期のタイミングを考えなければならない。

26.8

導出テーブルの使い方

ユーザーは、クエリの中で導出テーブルを作ることができる。これは、それが作られたクエリの中だけを存在スコープとするテーブルである。AS演算子を使うことで、列とテーブルに名前を付けることもできる。構文は非常に簡単だが、新米プログラマはスコープ規則に混乱することがある。

```
(<query expression>) [[AS] <table name> [(<column list>)]]
```

26.8.1　FROM 句で導出テーブルを使う

導出テーブルの構文はほとんどオプションだが、常にすべて記述することが望ましい。基底テーブルを宣言するときと同じように、テーブル名や列名に意味のある名前を宣言することで、利用する際にもあいまいさがなくなる。導出テーブルのトリッキーなところは、サブクエリのスコープ規則である。サンプルとして、次のようなクエリを考えよう。

```
SELECT *
  FROM (Foo
        LEFT OUTER JOIN
          Bar
          ON Foo.x = Bar.x)
        INNER JOIN
           Floob
           ON Floob.y = x; -- 間違い
```

Foo、Bar、Floobはいずれも一番外側のクエリから参照できる。したがって、外側のON 句で使われているxという列名はあいまい

で、Foo.xなのかBar.xなのかわからない。これをテーブル名で修飾する必要がある。

```
SELECT *
  FROM (Foo AS F1
        LEFT OUTER JOIN
          Bar AS B1
          ON F1.x = B1.x)
        INNER JOIN Floob
           ON Floob.y = Foo.x; -- まだ間違い
```

これが間違いなのは、F1とB1という別名が基底テーブルを外側のクエリから隠蔽するため、Foo.xという列名は外側からは参照できないからだ。解決策は、サブクエリ全体に別名を付けて、あいまいな列名の名前を変えることだ。

```
SELECT *
  FROM ((SELECT x FROM Foo)
       LEFT OUTER JOIN
       (SELECT x FROM Bar)
            ON Foo.x = Bar.x) AS Foobar(x1, x2)
       INNER JOIN Floob
          ON Floob.y = x1;
```

今度は、外側のクエリはFoobarしか見ないため、FooとBarにアクセスすることはできない。代わりに、x1という導出テーブルの列を使うのだ。なお、ON句を使った結合を複数使った場合、実行順序は左から右である。内部結合とクロスの場合は順番がどうであれ結果は変わらないが、外部結合の場合は結果に影響する（詳細は25.3節参照）。

26.8.2 VALUES 構築子を使った導出テーブル

SQL-99から、VALUES構築子はINSERT文以外でも使えるようになり、これでテーブルを作れるようになった。26.7.3 項で使用した単一列を持つテーブルはその一例だ。この構築子は、現在では複数行からなるテーブルを構築できるように拡張されている。たとえば、次のようなイメージだ。

```
VALUES ('John', 7), ('Mark', 8), ('Fred', 10), ('Sam', 7)
```

ただし、こうしたテーブルは制約を持つことはできない。それどころか、AS句で与えない限りテーブル名も持たない【訳注5】。したがって、これは制約を持ち変更が可能なきちんとしたテーブルというよりは「テーブル定数」だと考えてほしい。このテーブル構築子においては、当然のことながら、すべての行はUNION互換でなければならないが、システムはデフォルトのデータ型を選択することもできる。これを回避する手段の1 つは、明示的に望ましいデータ型に変換することだ。

この列には任意のスカラ式を取ることもできる。

```
(VALUES ((SELECT MAX(x) FROM Foo), 42),
        ((SELECT MAX(y) FROM Bar), (12+2)),
         (12, 3)) AS Weird (a, b)
```

VALUES構築子を導出テーブルとして使うには、次のようにFROM句の中に入れるだけでよい。

```
SELECT <local name>.*
  FROM (VALUES (...), (...), ...) AS <local_name>(<column name list>)
```

26.9

共通表式

WITH句はSQL-99で追加された機能だ。その目的は、同じ導出テーブルがコードの複数の箇所に現れる場合に、1か所に定義をまとめて繰り返しを避けることである。

26.9.1　単純な共通表式

単純な共通表式（CTE：Common Table Expression）の使い方は、SELECT 文の先頭に1 つ以上の導出テーブルを宣言すればよい。その導出テーブルは、宣言された順に作られる。これはつまり、ある共通表式が参照できる共通表式は、自分より前に宣言されたものに限られるということだ。基本的な構文は次のとおりだ。

```
<cte element>::=
<table name> [(<column list>)]
AS <subquery expression>
<cte list>::= <cte element> |<cte list>, <cte element>
<cte select statement>::= WITH <cte list> <select statement>;
```

簡単な例として、一番高い製品をUPCコード（upc）単位で出力するレポートを考えよう。

これは次のような2段階を踏んで求めることができる。まず最初に、WITH句でUPCコード単位の価格を求める。次に、その結果を使ってどの製品が最も高額かを求める。

```
WITH ItemSummary (upc, item_price_tot)
AS
(SELECT upc, SUM(item_price)
   FROM OrderDetails
  GROUP BY upc)
 --メインのクエリ
 SELECT P1.upc, P1.item_price_tot
   FROM ItemSummary AS P1
  WHERE P1.item_price_tot
          = (SELECT MAX(P1.item_price_tot)
               FROM ItemSummary AS P2);
```

もし共通表式が使えないとすると、`ItemSummary` は2 回現れるか、あるいはビューにしなければならない。管理者権限を持っていないユーザーはビューを作ることはできないが、共通表式ならば使える。このサンプルでは簡単なビューを置き換えたが、実務ではもっと複雑なクエリに適用することを考えれば、そのメリットは大きい。

26.10

再帰的な共通表式

共通表式は再帰的に定義することもできる。再帰的定義は2 つの部分からなる。まず、不動点または不変式と呼ばれる開始点がある。第2 の部分は、前段の結果にルールを適用することで集合に要素を付け加えていくパートである。`UNION [ALL]`を含む構築の構文は次のとおりだ。

```
<cte select statement>::=
WITH RECURSIVE <fixed point cte element>
UNION [ALL]
<step cte element>
<select statement>;
```

手続き型言語の教科書に載っている古典的な再帰の問題に、階乗関数がある。たとえば、n!は次のように定義される。

```
CREATE OR REPLACE FUNCTION Fact (n IN INTEGER)
RETURN INTEGER
IS
BEGIN
  IF n IN (0, 1)
  THEN RETURN 1;
  ELSE RETURN (n * Fact(n - 1));
  END IF;
END;
```

これは単純な再帰関数で、ループを使って書くこともできる。標準SQLで線形再帰と呼ばれているもので、最も単純だ。再帰の種類はこれだけではない。もし再帰に興味を持ったなら、好きな手続き型言語でアッカーマン関数を書いてみてほしい【訳注6】。見た目は非常に簡単だが、奥深い性質を持っている。

```
CREATE FUNCTION Ack (m IN INTEGER, n IN INTEGER)
RETURN INTEGER
IS
BEGIN
  IF m = 0
  THEN RETURN (n + 1);
  END IF;

  IF m > 0 AND n = 0
  THEN RETURN Ack(m - 1, 1);
  END IF;

  IF m > 0 AND n > 0
  THEN RETURN Ack(m - 1, Ack(m, n - 1));
  END IF;
END;
```

(m =0)のときは、単純な加算関数である。(m =1)のときは、加算のために再帰呼び出しが行われ、mが増加するにつれて爆発的に大きくなっていく。各レベルは、すべてが加算されるまで先行のレベルを呼び出す。その計算結果と再帰の深さは非常に速く大きくなり、Ack(4, 2)は10進数で19, 729桁の整数になる。

26.10.1　単純な加算

再帰的な共通表式を利用する最もよくあるケースは、ループを代用する場合である。たとえば、ある関数の参照テーブルを作るための共通表式は以下のようになる。

```
WITH RECURSIVE Increment (i, function_value)
AS
(SELECT i, j
   FROM (VALUES (1,f(1))) AS X(i, j)
 UNION
 SELECT (i+1), f(i+1)
   FROM Increment
  WHERE (i+1) <= 1000)
SELECT i, function_value FROM Increment;
```

同じことをやるならシーケンスやROW_NUMBER 関数を使うほうが簡単なので、これは良い例ではないが、アイデア自体はよく利用されるので、知っておいてほしい。

26.10.2　単純なツリー探索

製品に使われる諸部品を管理する簡単な隣接リストモデルを作って、これを再帰的にパースするという問題を考える【訳注7】。

```
CREATE TABLE BillOfMaterials
(part_name VARCHAR(20) NOT NULL PRIMARY KEY,
 assembly_nbr INTEGER, -- NULLは完成品
 subassembly_nbr INTEGER NOT NULL);
```

BillOfMaterials：部品階層テーブル　part_name：部品名　assembly_nbr：部品番号　subassembly_nbr：下位部品番号

サンプルデータは次のようになる。

part_name	assembly_nbr	subassembly_nbr
A	11	12
B	12	10
C	12	9
D	10	8
E	8	7
F	20	11
G	6	5

H	20	7
I	10	4

完成品のレコードに対するassembly_nbrは、NULL が割り当てられる。これは、もはや他のいかなる製品の部品にもならない、という意味だ。次の再帰的SQLは、共通表式のトリックを使っている。

```
WITH RECURSIVE Explosion (assembly_nbr, subassembly_nbr, part_name)
AS
(SELECT BOM.assembly_nbr, BOM.subassembly_nbr, BOM.part_name
  FROM BillOfMaterials AS BOM
 WHERE BOM.assembly_nbr = 12 -- 探索の開始点
 UNION
 SELECT Child.assembly_nbr, Child.subassembly_nbr, Child.part_name
   FROM Explosion AS Parent, BillOfMaterials AS Child
  WHERE Parent.subassembly_nbr = Child.assembly_nbr)
-- メインのSELECT文
SELECT assembly_nbr, subassembly_nbr, part_name
  FROM Explosion;
```

結果

```
 assembly_nbr    subassembly_nbr    part_name
==============================================
 12              10                 B
 12              9                  C
 10              8                  D
 10              4                  I
 8               7                  E
```

これは、部品番号#12の製品を構成する部品をすべて見つける。BillOfMaterialsテーブルから最初の集合を選択し、その後、UNION で自身を付け加えていく。その際、オリジナルのBillOfMaterialsテーブルは式の中に隠される。標準SQL では、RECURSION の予約語は、コンパイラに対して、Explosionテーブ

ルに1行も追加されなくなるまで計算を続けることを指示する。それによって、メインのSELECT 文は、あたかもこれを別のテーブルのように扱うことができる。

26.11

マテリアライズドクエリテーブル

マテリアライズドクエリテーブル（MQT：Materialized Query Table）は、スナップショットテーブル、自動サマリテーブル、マテリアライズドビューといった名前でも知られている。要するにクエリの結果を事前に計算して保存しておく機能だ。これによってクエリを再計算することなく結果にアクセスすることができる。同じクエリを繰り返し計算するよりも高速である。

実装によって構文に差はあるが、基本的には`CREATE TABLE` 文の構文をベースに、データをリフレッシュするためのクエリを付け加える。マテリアライズドクエリテーブルは基底テーブルなので、普通に`INSERT`、`UPDATE`、`DELETE`で更新することができる。ただし、これを作るためのクエリにはいくつかの制限が存在する。基本的には基底テーブルの列と簡単な関数を使うにとどめるのがよいだろう。

【訳注1】　いわゆるスタースキーマである。データウェアハウスのモデリングにおいてよくこの形が使われる。

【訳注2】　DB2 9.7では、`UNION`を使ったビューは更新不可能だが、`UNION ALL`ならば更新可能である。

【訳注3】　実装にもよるが、オプティマイザはインライン展開されたビューを効率的に解釈するさまざまな方法を持っている、たとえば、サブクエリと外側のクエリを区別せずにテーブルの結合を行う「ビューマージ」や、外側のクエリの検索条件をサブクエリの中に繰り込む「述語プッシュ」という方法などがある。

【訳注4】 WITH CHECK OPTIONは、Oracle、SQL Server、MySQLなどでサポートされている。

【訳注5】 列名をAS句で与えなかった場合は、実装依存の適当な名前が割り当てられる。

【訳注6】 アッカーマン関数は、ドイツの数学者アッカーマンの考えた関数で、以下の簡単な定義で与えられる関数Ack(m, n)のこと。

0以上の整数m、nに対して、

m = 0 のとき、n + 1

n = 0 のとき、Ack(m−1, 1)

それ以外のとき、Ack(m−1, Ack(m, n−1))

【訳注7】 隣接リストモデルは、リレーショナルデータベースで木構造を扱うための方法論の1つで、親ノードへのポインタ列を持つことでノード間のつながりを表現する。詳しくは第37章参照。

27

クエリによるデータの分割

Partitioning Data in Queries

　本章では、他のユーザーに提示したり、さらに分割を進めたりすることで高度な分析を行うことを可能にするような、意味のあるサブセットにデータをSQLで分割する方法を見ていく。

27.1

被覆とパーティション

まず、2 つの基本的な集合の概念を定義しておく。被覆（covering）は、ある集合に対する部分集合の族（集合の集合のこと）で、その和が元の集合に戻るものである。類またはパーティション（partition）は、互いに共通の要素を持たない被覆である。ピザを切り分けたのはパーティションで、それらをペパロー二の生地の間に挟めば、被覆である【訳注1】。

パーティションは多くのレポートの基礎である。これがレポートに便利なのは、集約すると部分の合計が全体に戻ることだ。たとえば、企業の財務は部署に分割できる。部署は課に分割できる。逆に、部署の財務は課の合計であり、全社の財務は部署の合計である。もし1 つの課が2 つの異なる部署に属していたら、これはパーティションはではなく被覆となり、正しい合計は計算が困難だ。

27.1.1　範囲による分割

データ処理におけるよくある問題の1つに、物事を数値やアルファベットの範囲で分割するというものがある。コードを範囲内の値に変換する最善の方法は、範囲を上限と下限の値で定義するテーブルを作ることだ。この方法は補助テーブルについて解説した第8 章で詳しく述べたが、ここでもう一度簡単に振り返っておこう。このテーブルを使うと、失われた値も簡単に見つけることができ、テーブルは制約によって完全な状態が保証される。たとえば、郵便番号の範囲と2文字の州コードを表すテーブルを考えよう。

```
CREATE TABLE ZipCodes
(state_code CHAR(2) NOT NULL PRIMARY KEY,
 low_zip CHAR(5) NOT NULL UNIQUE,
 high_zip CHAR(5) NOT NULL UNIQUE,
     CONSTRAINT zip_order_okay CHECK(low_zip < high_zip));
```

ZipCodes：郵便番号テーブル　　state_code：州コード　　low_zip：下限番号
high_zip：上限番号

このテーブルを使ってAddressBookテーブルにある郵便番号から都市の名前と州コードを見つけるクエリは次のようになる。

```
SELECT A1.name, A1.street, SZ.city, SZ.state_code, A1.zip
  FROM ZipCodes AS SZ, AddressBook AS A1
 WHERE A1.zip BETWEEN SZ.low_zip AND SZ.high_zip;
```

この検索条件で気をつけるべきことは、3 つの列のうち1 つでもNULLであればBETWEEN 述語はUNKNOWN になり、このWHERE 句にはヒットしないことだ。また、範囲の上限値を別の行の下限値と重なるようなテーブルにすると、1 つの値に対して2 つの範囲がヒットしてしまうので、気をつけなければいけない。

27.1.2　単一列の範囲テーブル

もし範囲テーブルが、互いにオーバーラップすることのない範囲を定義していることが確実であれば、コードを変換するために上限値だけを使うことができる。たとえば、点数から成績を決めるテーブルが((100, 'A')、(89, 'B')、(79, 'C')、(69, 'D')、(59, 'F'))のようなデータを持っていた場合である。同様に、先ほどの郵便番号のテーブルも実は上限値の1 列だけ持つテーブルでも代用できる。

```
CREATE TABLE StateZip2
(high_zip CHAR(5) NOT NULL,
 state CHAR(2) NOT NULL,
     PRIMARY KEY (high_zip, state));
```

StateZip2：郵便番号テーブル2　　high_zip：上限番号　　state：州コード

我々が書きたいクエリは、ある郵便番号に対して最大の下限値、または最小の上限値を与えるクエリである。最大の下限値とは入力の値以下で、最大の値である。一方、最小の上限値とは、この入力の値以上で最小の列の値である。SQLでこれら2つの値を見つけようとするのは、あまりパフォーマンスが良くない。なぜなら、サブクエリを使うことになり、しかもかなり複雑なコードになるからだ。したがって、下限値と上限値の2つの列を持っておくほうが現実的な解になることが多い。参考までに、1 列だけでやった場合のアドレスブックから州コードを見つけるクエリを次に示そう。見てのとおり、かなり複雑だ。

```
SELECT name, street, city, state, zip
  FROM StateZip2, AddressBook
 WHERE state =
         (SELECT state
            FROM StateZip2
           WHERE high_zip =
                    (SELECT MIN(high_zip)
                       FROM StateZip2
                      WHERE Address.zip <= StateZip2.high_zip));
```

27.1.3　関数による分割

テーブルをパーティションに分割する方法には関数を使うものもある。関数の結果が非常に複雑で、SQLでは書くのが難しい場合、テーブルに関数の結果を保持する列を付け加えるケースを考えよう。

このテクニックの典型例の1 つがSoundex 関数だ（第12 章参照）。Soundex 関数は似た発音の名前に同じコードを割り当てる。この計算は非常に複雑なので、SQLではベンダー拡張を使わないと書くのが非常に難しい。

もともとSQLは計算を目的に作られた言語ではなかった。だが多くのベンダーが、他のプログラミング言語で使えるような関数のライブラリにクエリからアクセスできるようにしている。これは注意が必要だ。クエリからそうした関数を呼び出すのにかかる時間とコストには自覚的でなければならない。製品によっては他の言語のライブラリにパラメータを渡し、その結果を1行ごとに受け取るというスレッディドコード（threaded-code）の方式を取るものもあるが、これは非常に長い実行時間を要する。またある製品は他の言語のライブラリをコンパイルしてSQLにリンクさせられるものもある。

これはあくまで一般論だが、一番安全な方法は、パラメータの値を他の言語が読むことのできる標準フォーマットでファイルに書き出すことだ。そして、プログラムでそのファイルを読み、結果をINSERT文のパラメータに渡し、データベースにロードする。

27.1.4　数列による分割

人類は有史以来、シーケンシャルな順序を持ったパターンを探してきた。この順序は時間的なものもあれば、数列をナンバリングすることで得られるものもある。たとえば、支払い履歴から期日どおりに支払いが行われたか、延滞したかを分けてみよう。

```
CREATE TABLE PaymentHistory
(payment_nbr INTEGER NOT NULL PRIMARY KEY,
 paid_on_time CHAR(1) DEFAULT 'Y' NOT NULL
     CHECK(paid_on_time IN ('Y', 'N')));
```

PaymentHistory：支払い履歴テーブル　　payment_nbr：支払い番号

paid_on_time：延滞フラグ

```
INSERT INTO PaymentHistory
VALUES (1006, 'Y'),
       (1005, 'Y'),
       (1004, 'N'),
       (1003, 'Y'),
       (1002, 'Y'),
       (1001, 'Y'),
       (1000, 'N');
```

我々が欲しい結果は、次のように支払いが期日どおり行われたかどうかを基準に、同じタイプのコードをひとまとまりの番号でグルーピングしたものだ。つまり、`paid_on_time`列の値が変わるたびに、`grp`列がインクリメントされていく。

grp	payment_nbr	paid_on_time
1	1006	Y
1	1005	Y
2	1004	N
3	1003	Y
3	1002	Y
3	1001	Y

4	1000	N

次に示すのはフーゴ・コーネリスの考えた解だ。これは支払いに歯抜けのない連番が振られていることを前提としている。

```
SELECT (SELECT COUNT(*)
          FROM PaymentHistory AS H2,
               PaymentHistory AS H3
         WHERE H3.payment_nbr = H2.payment_nbr + 1
           AND H3.paid_on_time <> H2.paid_on_time
           AND H2.payment_nbr >= H1.payment_nbr)
             + 1 AS grp,payment_nbr, paid_on_time
  FROM PaymentHistory AS H1;
```

このクエリは、履歴からパターンを見つける場合にとても便利である。この問題のより複雑なバージョンとして、3 つ以上のカテゴリを扱うものがある。たとえば、製品の品質を管理するために、製品のリストと品質スコアのデータを持つ`ProductTests`というテーブルを考えよう。このテーブルから、同じ製品のシーケンシャルなグループに対する品質スコアの平均を求めたい。たとえば、製品A のバッチ番号1、2、3 に対する平均品質スコアは、(80 + 70 + 80) / 3 = 76.6666...といった具合だ。ここで注意が必要なのは、バッチ番号 8 番の製品Aのように、飛び石になった場合は別グループとして扱うことだ。

```
CREATE TABLE ProductTests
(batch_nbr INTEGER NOT NULL PRIMARY KEY,
 prod_code CHAR(1) NOT NULL,
 prod_quality DECIMAL(8,4) NOT NULL);
```

ProductTests：製品テスト　batch_nbr：バッチ番号　prod_code：製品コード
prod_quality：品質

```
INSERT INTO ProductTests (batch_nbr, prod_code, prod_quality)
VALUES (1, 'A', 80),
       (2, 'A', 70),
       (3, 'A', 80),
       (4, 'B', 60),
       (5, 'B', 90),
       (6, 'C', 80),
       (7, 'D', 80),
       (8, 'A', 50),
       (9, 'C', 70);
```

クエリは次のようになる。

```
SELECT X.prod_code, MIN(X.batch_nbr) AS start_batch_nbr, end_batch_nbr,
       AVG(B4.prod_quality) AS avg_prod_quality
  FROM (SELECT B1.prod_code, B1.batch_nbr,
               MAX(B2.batch_nbr) AS end_batch_nbr
          FROM ProductTests AS B1, ProductTests AS B2
         WHERE B1.batch_nbr <= B2.batch_nbr
           AND B1.prod_code = B2.prod_code
           AND B1.prod_code
                 = ALL (SELECT prod_code
                          FROM ProductTests AS B3
                         WHERE B3.batch_nbr BETWEEN B1.batch_nbr
                                                AND B2.batch_nbr)
         GROUP BY B1.prod_code, B1.batch_nbr) AS X
           INNER JOIN
             ProductTests AS B4
    ON B4.prod_code = X.prod_code AND B4.batch_nbr = X.batch_nbr
 GROUP BY X.prod_code, X.end_batch_nbr;
```

結果

```
 prod_code    start_batch_nbr    end_batch_nbr    avg_prod_quality
==========================================================================
 A            1                  3                76.6666666666666667
 B            4                  5                75.0000000000000000
 C            6                  6                80.0000000000000000
 D            7                  7                80.0000000000000000
 A            8                  8                50.0000000000000000
 C            9                  9                70.0000000000000000
```

27.1.5　ウィンドウによる分割

ティム・ミラーも同じような問題を抱えていた。ミラーは、取引を管理するTransactions テーブルから、各報告期間ごとのサマリを求めたいと考えていた。

```
CREATE TABLE Transactions
(report_period INTEGER NOT NULL,
 trans_nbr INTEGER NOT NULL PRIMARY KEY,
 trans_qty INTEGER NOT NULL);
```

Transactions：取引テーブル　report_period：報告期間　trans_nbr：取引番号
trans_qty：取引量

```
 report_period    trans_nbr    trans_qty
=========================================
 1                1            50
 1                2            100
 1                3            20
 2                4            10
 2                5            200
 3                6            40
```

以下は、プラマン・ラチェフが考えた方法だ。

```
SELECT report_period, SUM(trans_qty) AS trans_qty_tot,
       MAX(CASE WHEN rk1 = 1 THEN trans_nbr END) AS start_trans_nbr,
       MAX(CASE WHEN rk2 = 1 THEN trans_nbr END) AS end_trans_nbr
  FROM (SELECT report_period, trans_qty, trans_nbr,
               ROW_NUMBER() OVER(PARTITION BY report_period
                                 ORDER BY trans_nbr ASC) AS rk1,
               ROW_NUMBER() OVER(PARTITION BY report_period
                                 ORDER BY trans_nbr DESC) AS rk2
          FROM Transactions) AS T
 GROUP BY report_period;
```

結果

```
 report_period    trans_qty_tot    start_trans_nbr    end_trans_nbr
=================================================================
 1                170              1                  3
 2                210              4                  5
 3                40               6                  6
```

ROW_NUMBER 関数を2 回使って、昇順（rk1）と降順（rk2）の順序づけを行うことで、rk1またはrk2が1の行は、報告期間における最初と最後の行であることがわかる。

同じウィンドウ関数を使うなら、次のような方法もある。

```
SELECT report_period,
       SUM(trans_qty) OVER (PARTITION BY report_period) AS trans_qty_tot,
       MIN(trans_nbr) OVER (PARTITION BY report_period) AS start_trans_nbr,
       MAX(trans_nbr) OVER (PARTITION BY report_period) AS end_trans_nbr
  FROM Transactions;
```

結果

```
 report_period    trans_qty_tot    start_trans_nbr    end_trans_nbr
=================================================================
 1                170              1                  3
 1                170              1                  3
 1                170              1                  3
 2                210              4                  5
 2                210              4                  5
 3                40               6                  6
```

こちらのほうが、ウィンドウ関数のPARTITION BY 句のキーがすべて同じreport_periodであるため、オプティマイザが最適化しやすい。おそらくソートは1回で済むはずだ。

27.2

関係除算

関係除算は、コッドが定義した8つの基本的な関係代数の1つである【訳注2】。その考え方は、除テーブルで被除テーブルを分割し、商を得るというものだ。商テーブルは、被除テーブルの中で、除テーブルのレコードをすべて持つような部分集合から構成される。

関係除算は、実例を見たほうがわかりやすい。飛行機のパイロットと、彼らが操縦できる飛行機の待機状況を表すテーブルがあるとしよう。求めたいのは、格納庫に待機中の飛行機をすべて操縦できるパイロットの名前だ。この問題においては、`PilotSkills`テーブルが被除テーブル、`Hangar`テーブルが除テーブル、結果のパイロット名の集合が商となる。

```
CREATE TABLE PilotSkills
(pilot CHAR(15) NOT NULL,
 plane CHAR(15) NOT NULL,
     PRIMARY KEY (pilot, plane));
```

PilotSkills：パイロットスキルテーブル　　pilot：パイロット　　plane：飛行機

PilotSkills

```
 pilot     plane
=======================
 Celko     Piper Cub
 Higgins   B-52 Bomber
 Higgins   F-14 Fighter
 Higgins   Piper Cub
 Jones     B-52 Bomber
 Jones     F-14 Fighter
 Smith     B-1 Bomber
 Smith     B-52 Bomber
 Smith     F-14 Fighter
 Wilson    B-1 Bomber
 Wilson    B-52 Bomber
 Wilson    F-14 Fighter
 Wilson    F-17 Fighter
```

```
CREATE TABLE Hangar
(plane CHAR(15) NOT NULL PRIMARY KEY);
```

Hangar：格納庫テーブル　　plane：飛行機

Hangar

```
 plane
=============
 B-1 Bomber
 B-52 Bomber
 F-14 Fighter
```

PilotSkills÷Hangar

```
 pilot
============================
 Smith
 Wilson
```

　このサンプルデータでは、待機中のすべての飛行機を操縦できるのはSmithとWilsonだけだ。注意してほしいのは、HigginsとCelkoは、Piper Cubを操縦できるのだが、その機体はこのとき格納庫に存在しないということだ。

関係除算の重要な特性の1つは、除テーブルと商をクロス結合する（すなわち直積を得る）と被除テーブルの妥当な部分集合になることである。関係除算という名前は、クロス結合がちょうど掛け算のように作用することから付けられている。

27.2.1　剰余を持った除算

関係除算には2 種類ある。剰余を持った除算と厳密な除算である。前者は、被除テーブルは除テーブルよりも多くの値を持っていてもかまわない。これはコッドが最初に考えた定義だ。たとえば、たとえば、格納庫に待機中の飛行機以外も操縦できるパイロットであっても、商の中に含んでよい。これを得るためのクエリは次のとおりだ。

```
SELECT DISTINCT pilot
  FROM PilotSkills AS PS1
 WHERE NOT EXISTS
        (SELECT *
           FROM Hangar
          WHERE NOT EXISTS
                (SELECT *
                   FROM PilotSkills AS PS2
                  WHERE (PS1.pilot = PS2.pilot)
                    AND (PS2.plane = Hangar.plane)));
```

このクエリのやっていることは、よくある戦争映画のワンシーンを思い浮かべればわかる。不敵なパイロットが格納庫へ入ってきて、あたりをぐるっと見渡してこう言う。「オレが操縦できない機体なんかないね！」我々が見つけたいのは、まさにこういうパイロットだ。NOT EXISTS 述語でこの二重否定を表現しており、多くの

実装ではテーブル全体ではなくインデックスの値を見にいくので、パフォーマンスも良好だ。

関係除算を解くこのクエリは、クリス・デイトの教科書で取り上げられて有名になった。だが関係除算を実現する解はほかにもあるし、これが最速というわけでもない。こんなに深いサブクエリの入れ子を使わなくても解く方法がある。次は、私が考えた解ではないが、拙著で取り上げたことで広まったものだ。

```
SELECT PS1.pilot
  FROM PilotSkills AS PS1, Hangar AS H1
 WHERE PS1.plane = H1.plane
 GROUP BY PS1.pilot
HAVING COUNT(PS1.plane) = (SELECT COUNT(plane) FROM Hangar);
```

だが、この2 つの解には重大な違いがある。それは空集合の扱い、つまり格納庫が空だった場合――「ゼロによる関係除算」と言ってもよい――の結果である。デイトのクエリは、格納庫が空の場合、パイロット全員を返すが、COUNT関数を使ったクエリは空集合を返す。さて、操縦すべき飛行機が1 機もない場合、誰でも操縦できると考えるのと、そもそも操縦する対象がないのだから、操縦できる人間もいないと考えるのと、どちらが正しいのだろう【訳注3】。なお、デイトは自著『データベースシステム概論第6版』（丸善刊）で、COUNT版と同じ結果を返す除算の演算子（DIVIDEBY ..... PER）を定義している。

27.2.2　厳密な除算

関係除算の2 番目の種類が、「厳密な関係除算」である。この除算では、被除テーブルが除テーブルと過不足なく厳密に一致しなけれ

ばならない。

```
SELECT PS1.pilot
  FROM PilotSkills AS PS1
         LEFT OUTER JOIN
           Hangar AS H1
    ON PS1.plane = H1.plane
 GROUP BY PS1.pilot
HAVING COUNT(PS1.plane) = (SELECT COUNT(plane) FROM Hangar)
   AND COUNT(H1.plane)  = (SELECT COUNT(plane) FROM Hangar);
```

結果

```
 pilot
============================
 Smith
```

　このクエリの意味は、「パイロットが格納庫内の機体数と同数の操縦免許を持っており、かつ、すべての免許が格納庫内の機体と一致する（ほかの機体の免許ではない）必要がある」というものだ。「ほかの機体」は左外部結合によってNULLとして現れるが、COUNT(x)はNULLを数えない点がポイントだ。

　ちなみに、このクエリを簡略化しようとしてHAVING句を、

```
HAVING COUNT(PS1.plane) = COUNT(H1.plane)
```

と書くのは間違いなので注意してほしい。この条件は、格納庫にある機体数がnで、かつ、パイロットが持っている免許の数もnであると言っているが2つの集合が互いに等しいだけではまだ十分条件ではない。

27.2.3　パフォーマンスに関する注意

『DB2 Magazine』というオンラインマガジンの1996年冬号に、シェリル・ラーセンによる「強力なSQL：基礎を超えて」という記事が掲載された【訳注4】。DB2を使った彼女の測定によれば、除テーブルが被除テーブルの25%未満であれば、入れ子のEXISTS バージョンのほうが速く、25%以上であればCOUNTバージョンのほうが速いとのことだ。

反対に、マシュー・W・スポールディングは、SQL Server 2000で測定して逆の結果を得た。彼は、被除テーブルに200 万行、除テーブルに約1000 行を用意し、商が1000 行程度になる条件で検証したところ、COUNT バージョンは1 秒以下で実行されたが、NOT EXISTSのクエリはおよそ5秒程度かかったという。

この2つの結果から得られる教訓は、結局、読者は自分の使う環境でテストするしかないということだ。

27.2.4　トッドの除算

ステファン・トッドは2つのテーブルを結合し、結合キーを取り除いて残りの列が条件を満たすかどうかで除算を行う演算を定義した。

例としてJobParts(job, pno)とSupParts(sno, pno)という2 つのテーブルがあるとしよう。これは仕事に必要な部品とその業者が提供する部品のテーブルだ。仕事で必要になるすべての部品を供給する業者と仕事のペアを見つけよう。注意してほしいのだが、これは業者が提供するすべての部品を必要とする仕事と業者のペアを見つけるのとは違う。

読者は関係除算を使ってJobPartsテーブルをSupPartsテーブルで割ろうとするだろう。一般原則として剰余は被除テーブルからくるが、除テーブルのすべての値は結果（商）に保存される。

JobParts	
job	pno
j1	p1
j1	p2
j2	p2
j2	p4
j2	p5
j3	p2

SupParts	
sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s1	p5
s1	p6

s2	p1
s2	p2
s3	p2
s4	p2
s4	p4
s4	p5

ピエール・マリンはトッドの除算を行う方法として次のようなクエリを考えた。

```
SELECT DISTINCT JP1.job, SP1.sno
  FROM JobParts AS JP1, SupParts AS SP1
 WHERE NOT EXISTS
        (SELECT *
           FROM JobParts AS JP2
          WHERE JP2.job = JP1.job
            AND JP2.pno
                  NOT IN (SELECT SP2.pno
                            FROM SupParts AS SP2
                           WHERE SP2.sno = SP1.sno));
```

結果

```
 job   sno
===========
 j1    s1
 j1    s2
 j2    s1
 j2    s4
 j3    s1
 j3    s2
 j3    s3
 j3    s4
```

これは実際のところ、コッドの除算のクエリを少し修正したものだ。外側のSELECT文で両方のテーブルに結合を使うよう拡張している。2 番目のサブクエリのIN 述語はNOT EXISTS 述語で置き換えることもできる。そのほうが速い可能性もあるが、オプティマイザ次第だ。

これと少し似た問題に、同じ部品の組み合わせを売っている業者を見つける、という問題がある。ここでデータは次のようなものを使おう。

SupParts	
sno	pno
s1	p2
s3	p1
s4	p1
s5	p1

```
SELECT S1.sno, S2.sno
  FROM SupParts AS S1, SupParts AS S2
 WHERE S1.sno < S2.sno   -- 業者の組み合わせ
   AND S1.pno = S2.pno   -- 同じ部品を扱っている
 GROUP BY S1.sno, S2.sno
HAVING COUNT(*) = (SELECT COUNT (*) -- 取り扱い部品数が同じ
                     FROM SupParts AS S3
                    WHERE S3.sno = S1.sno)
   AND COUNT(*) = (SELECT COUNT (*)
                     FROM SupParts AS S4
                    WHERE S4.sno = S2.sno);
```

結果

```
 sno   sno
===========
 s3    s4
 s3    s5
 s4    s5
```

このクエリは少し変更するだけでトッドの除算になる。部品が共通の仕事に使われるという制限を付け加えればよいのだ。スティーブ・キャスは数値のコードを使った場合の特殊なバージョンを考えた。どのチームに選手が属しているかを管理するテーブルを考えよう。

```
CREATE TABLE Players
(player_id INTEGER NOT NULL);
```

Players：選手テーブル　　player_id：選手ID

```
CREATE TABLE TeamAssignments
(player_id INTEGER NOT NULL,
 team_id CHAR(5) NOT NULL,
     PRIMARY KEY (player_id, team_id));
```

TeamAssignments：チーム割り当てテーブル　　player_id：選手ID　　team_id：チームID

同じチームに属する選手のペアを求めるには、次のようなクエリを書く。

```
SELECT P1.player_id, P2.player_id
  FROM Players AS P1, Players AS P2
 WHERE P1.player_id < P2.player_id
 GROUP BY P1.player_id, P2.player_id
HAVING P1.player_id + P2.player_id
         = ALL (SELECT SUM(P3.player_id)
                  FROM TeamAssignments AS P3
                 WHERE P3.player_id IN (P1.player_id, P2.player_id)
                 GROUP BY P3.team_id);
```

27.2.5　結合を使った除算

複数の結合を使うことで関係除算を実施する方法もある。たとえば、Higginsと同じ飛行機で飛ぶことのできるパイロットを見つけるには、次のクエリを使う。

```
SELECT SP1.Pilot
  FROM (((SELECT plane FROM Hangar) AS H1
        INNER JOIN
          (SELECT pilot, plane FROM PilotSkills) AS SP1
            ON H1.plane = SP1.plane)
              INNER JOIN (SELECT *
                            FROM PilotSkills
                           WHERE pilot = 'Higgins') AS H2
                ON H2.plane = H1.plane)
 GROUP BY SP1.Pilot
HAVING COUNT(*) >= (SELECT COUNT(*)
                      FROM PilotSkills
                     WHERE pilot = 'Higgins');
```

最初の結合は、乗ることのできるパイロットがいる飛行機をすべて格納庫の中から見つける。2番目の結合はその中から「`SELECT * FROM PilotSkills WHERE pilot ='Higgins'`」という条件に合致するパイロットを見つける。もしHAVING句の演算子を>=から=に変えれば、このクエリは厳密な除算を行うものになる。あるいは、>=から<=または<に変えれば、Higginsの操縦することのできる飛行機を1つは操縦できるパイロットを見つけることができる。

このクエリを使うなら、除テーブル（この場合はHangar）をビューにしておくのが可読性の観点でいいかもしれない。それにそのほうが、オプティマイザにこれを一度計算すれば十分であることを知らせることができる。製品によっては、入れ子のクエリよりもこちらの結合を使ったクエリのほうが速いかもしれない。なぜなら、この結合は常に主キーを使うことができるため、結合演算は非常に高速に行われるからだ。

27.2.6　集合演算を使った除算

標準SQLには差集合の演算子EXCEPTがある。これを使うとコッドの関係除算を非常に簡単に書くことができる。EXCEPT演算子は被除テーブルから除テーブルを取り除いてくれるからだ。もし、結果が空ならばマッチする集合ということになる。逆に結果が空でなければ一致しなかったということだ。先ほどのパイロットと格納庫のテーブルを使えば次のように書ける。

```
SELECT DISTINCT Pilot
  FROM PilotSkills AS P1
 WHERE (SELECT plane FROM Hangar
        EXCEPT
        SELECT plane
          FROM PilotSkills AS P2
         WHERE P1.pilot = P2.pilot) IS NULL;
```

このクエリの意味はNOT EXISTSの場合と同じだ。各パイロットは格納庫に入ってきて1機ずつ飛行機を見て自分が乗れる機体に付箋を貼っていく。もしすべての機体に付箋を貼ったパイロットがいればそのパイロットは合格だ。このクエリでもう一つ面白いところは空集合が返ったかどうかを調べるためにIS NULL 述語を使っている

ところだ。サブクエリは結果が空の場合、NULLになるからだ。ただし、IS NULL述語が複数行を引数に取れるよう、標準SQLの拡張をサポートしている実装でなければこの解は動かない。その場合は、NOT EXISTS述語を使って次のように書けばよい。

```
SELECT DISTINCT Pilot
  FROM PilotSkills AS P1
 WHERE NOT EXISTS
       (SELECT plane FROM Hangar
        EXCEPT
        SELECT plane
          FROM PilotSkills AS P2
         WHERE P1.pilot = P2.pilot);
```

27.3

レムレーの除算

リチャード・レムレーは少し複雑な関係除算を考えた。もともと彼が扱っていた問題は次のような2 つのテーブルを使う。1 つはマネージャーと彼らが管理するプロジェクトのテーブル。もう1つは部署や割り当てられたプロジェクトを示す社員テーブル。各社員は必ず1つだけの部署に所属しており、一度に1 つだけのプロジェクトでしか働けない。ただし、1 つの部署には同時に複数のプロジェクトが割り当てられることがある。そのため、1 つのプロジェクトは複数の部署にまたがることがある。

```
CREATE TABLE MgrProjects
(mgr_name CHAR(10) NOT NULL,
 project_id CHAR(2) NOT NULL,
     PRIMARY KEY(mgr_name, project_id));
```

MgrProjects：マネージャープロジェクトテーブル　　mgr_name：マネージャー名
project_id：プロジェクトID

```
INSERT INTO MgrProjects
VALUES ('M1', 'P1'), ('M1', 'P3'),
       ('M2', 'P2'), ('M2', 'P3'),
       ('M3', 'P2'),
       ('M4', 'P1'), ('M4', 'P2'), ('M4', 'P3');
```

```
CREATE TABLE Personnel
(emp_id CHAR(10) NOT NULL,
 dept CHAR(2) NOT NULL,
 project_id CHAR(2) NOT NULL,
     UNIQUE (emp_id, project_id),
     UNIQUE (emp_id, dept),
     PRIMARY KEY (emp_id, dept, project_id));
```

Personnel：社員テーブル　emp_id：社員ID　dept：部署　project_id：プロジェクトID

```
--部署#1のデータ
INSERT INTO Personnel
VALUES ('Al', 'D1', 'P1'),
       ('Bob', 'D1', 'P1'),
       ('Carl', 'D1', 'P1'),
       ('Don', 'D1', 'P2'),
       ('Ed', 'D1', 'P2'),
       ('Frank', 'D1', 'P2'),
       ('George', 'D1', 'P2');

--部署#2のデータ
INSERT INTO Personnel
VALUES ('Harry', 'D2', 'P2'),
       ('Jack', 'D2', 'P2'),
       ('Larry', 'D2', 'P2'),
       ('Mike', 'D2', 'P2'),
       ('Nat', 'D2', 'P2');

--部署#3のデータ
INSERT INTO Personnel
VALUES ('Oscar', 'D3', 'P2'),
       ('Pat', 'D3', 'P2'),
       ('Rich', 'D3', 'P3');
```

問題は各部署の各マネージャーについて、彼または彼女が、各部署のどれだけの社員を管理するかを示すレポートを作ることだ。たとえば、その部署のすべての社員ではないが、1人以上の社員を管理するマネージャーを見つけるには一種の関係除算を行う。

```
SELECT M1.mgr_name, P1.dept
  FROM MgrProjects AS M1
         CROSS JOIN
           Personnel AS P1
 WHERE M1.project_id = P1.project_id
 GROUP BY M1.mgr_name, P1.dept
HAVING COUNT(*) <> (SELECT COUNT(emp_id)
                      FROM Personnel AS P2
                     WHERE P2.dept = P1.dept);
```

結果

```
 mgr_name    dept
==================
 M1          D1
 M1          D3
 M2          D1
 M3          D1
 M3          D3
```

このクエリはHAVING句で=の代わりに<>を使っていることを除けば、いたって単純な関係除算だ。これはつまり、マネージャーと社員の関連が直接明示されていないため、「マネージャー – プロジェクト – 部署 – 社員」というようにつながりをたどっているのである。各プロジェクト下の社員の数を、各マネージャー下の社員の数と比較しているわけだ。

レムレーは私の解に変更を加え、集約関数の中で特性関数を使う方法を考えついた。

```
SELECT DISTINCT M1.mgr_name, P1.dept
  FROM (MgrProjects AS M1
          INNER JOIN
            Personnel AS P1
              ON M1.project_id = P1.project_id)
                INNER JOIN
                  Personnel AS P2
                    ON P1.dept = P2.dept
 GROUP BY M1.mgr_name, P1.dept, P2.project_id
HAVING MAX (CASE WHEN M1.project_id = P2.project_id
                 THEN 1 ELSE 0 END) = 0;
```

このクエリでトリッキーなのは、「GROUP BY M1.mgr_name, P1.dept_name, P2.project_id」と「SELECT DISTINCT M1.mgr_name, P1.dept_name」を組み合わせて使っていること

だ。これは、(mgr_name, dept_name, project_id)で表される(x, y, z)座標の三次元空間から、同じproject_id を持つ社員が同じ部署にいるかどうかを調べることで、(mgr_name, dept)で表される2次元へ還元している。

　このことを理解すると、クエリを次のように変更できることがわかる。マネージャーと部署の各組み合わせを表すテーブルを作り、マネージャーのプロジェクトに対する権限（authority）レベルを調べる。これが次のクエリの導出テーブルT1の意味することである。(authority =1)はプロジェクトを管理していないマネージャーを意味し、(authority=2)は当該のプロジェクトを管理しているマネージャーを指す。ビットフラグを立てているようなイメージを持つとわかりやすい。

```
SELECT T1.mgr_name, T1.dept,
       CASE SUM(T1.authority)
       WHEN 1 THEN 'None'
       WHEN 2 THEN 'All'
       WHEN 3 THEN 'Some'
       ELSE NULL END AS power
  FROM (SELECT DISTINCT M1.mgr_name, P1.dept,
                        MAX (CASE WHEN M1.project_id = P1.project_id
                                  THEN 2 ELSE 1 END) AS authority
          FROM MgrProjects AS M1
                 CROSS JOIN
                   Personnel AS P1
         GROUP BY M1.mgr_name, P1.dept, P1.project_id) AS T1
 GROUP BY T1.mgr_name, T1.dept;
```

結果

```
 mgr_name    dept    power
==========================
 M1          D1      Some
 M1          D2      None
 M1          D3      Some
 M2          D1      Some
 M2          D2      All
 M2          D3      All
 M3          D1      Some
 M3          D2      All
 M3          D3      Some
 M4          D1      All
 M4          D2      All
 M4          D3      All
```

このクエリは、ある部署内のすべてのプロジェクトに対してマネージャーが持つ権限レベルを合計する。これによってマネージャーが部署全体でどの程度の権限を持っているかを知ることができる。もしあるマネージャーの権限レベルの合計が1なら、その人はその部署のいかなるプロジェクトにもまったく権限を持っていない。もし権限レベルの合計が2なら、そのマネージャーは部署内のすべてのプロジェクトに対して権限を持っている。権限レベルの合計が3なら、権限レベルが1と2の複数のプロジェクトを持っていることを意味する。

ここで注意が必要なのは、マネージャーは決して1つのプロジェクトで1つの部署をカバーしなくても、複数のプロジェクトでカバーできれば「`All`」になることだ。たとえば、`M2` は、`P2`と`P3`を管理しており、`D3`はこの2つのプロジェクトでカバーされるため、結果的に`M2`は`D3`を「`All`」で管理することになる。

このクエリを格納庫とパイロットのサンプルに当てはめると次のようなクエリになる。

```
SELECT PS1.pilot,
       CASE WHEN COUNT(PS1.plane) > (SELECT COUNT(plane) FROM Hangar)
             AND COUNT(H1.plane)  = (SELECT COUNT(plane)FROM Hangar)
              THEN 'more than all'
            WHEN COUNT(PS1.plane) = (SELECT COUNT(plane) FROM Hangar)
             AND COUNT(H1.plane)  = (SELECT COUNT(plane) FROM Hangar)
              THEN 'exactly all'
            WHEN MIN(H1.plane) IS NULL
              THEN 'none '
            ELSE 'some ' END AS skill_level
  FROM PilotSkills AS PS1
         LEFT OUTER JOIN Hangar AS H1
    ON PS1.plane = H1.plane
 GROUP BY PS1.pilot;
```

結果

```
 pilot           skill_level
================================
 Higgins         some
 Smith           exactly all
 Wilson          more than all
 Jones           some
 Celko           none
```

27.4

RDBMS におけるブール式

もう一度「格納庫とパイロット」のサンプルを使おう。「Piper CubとF-14、またはPiper Cub とF-17を操縦できるパイロットを見つける」ブール式を含むクエリを作り、これを保存することを考える。トリックは、ブール式を選言標準形の形に変換することだ。選言標準形とは、ANDでつないだ述語のかたまりをORでつないだ形のことだ【訳注5】。

ちなみに、あらゆるブール式は標準形で表すことができる。標準形とは、各ブール変数が各項に一度しか現れない式である。すべての変数が各項に現れなくてもよい場合は、選言標準形と呼ばれる。任意のブール式を選言標準形に変換するアルゴリズムは少し複雑だが、回路設計に関するまともな解説書ならば載っている。我々の簡単な例は次のように変換される。

```
('Piper Cub' AND 'F-14 Fighter') OR ('Piper Cub' AND 'F-17 Fighter')
```

次に、これを以下のテーブルにロードする。

```
CREATE TABLE BooleanExpressions
(and_grp INTEGER NOT NULL,
 skill CHAR(16) NOT NULL,
     PRIMARY KEY (and_grp, skill));
```

BooleanExpressions：ブール式テーブル　and_grp：ANDグループ　skill：技術

```
INSERT INTO BooleanExpressions
VALUES (1, 'Piper Cub'), (1, 'F-14 Fighter'),
       (2, 'Piper Cub'), (2, 'F-17 Fighter');
```

これに対し、候補となるパイロットのテーブルは以下のような形をしている。

```
CREATE TABLE Candidates
(candidate_name CHAR(16) NOT NULL,
 skill CHAR(16) NOT NULL,
     PRIMARY KEY (candidate_name, skill));
```

Candidates：パイロット候補テーブル　　candidate_name：候補者名　　skill：技術

```
INSERT INTO Candidates
VALUES ('John', 'Piper Cub'),     --失格
       ('John', 'B-52 Bomber'),
       ('Mary', 'Piper Cub'),     --合格
       ('Mary', 'F-17 Fighter'),
       ('Larry', 'F-14 Fighter'), --失格
       ('Larry', 'F-17 Fighter'),
       ('Moe', 'F-14 Fighter'),   --合格
       ('Moe', 'F-17 Fighter'),
       ('Moe', 'Piper Cub'),
       ('Celko', 'Piper Cub'),    -- 失格
       ('Celko', 'Blimp'),
       ('smith', 'Kite'),         -- 失格
       ('smith', 'Blimp');
```

ここまで用意すれば、条件を満たすパイロットを探すクエリはシンプルになる。

```
SELECT DISTINCT C1.candidate_name
  FROM Candidates AS C1, BooleanExpressions AS Q1
 WHERE C1.skill = Q1.skill
 GROUP BY Q1.and_grp, C1.candidate_name
HAVING COUNT(C1.skill)
         = (SELECT COUNT(*)
              FROM BooleanExpressions AS Q2
             WHERE Q1.and_grp = Q2.and_grp);
```

結果

```
 candidate_name
================
 Moe
 Mary
```

SELECT 句にCOUNT(*)も含めておけば、候補者をランキングするのにも使える。たとえば、Moeは両方の資格を満たすが、他の候補者は1 つしか満たさない。こういう違いが、COUNT 列に現れる。

27.5

FIFOとLIFO の部分集合

在庫管理システムの経験のない読者には、実例を見せたほうがわかりやすいだろう。我々がある製品の在庫を管理していて、1日に1回、次のようなテーブルを使って在庫の入庫をすると考えてほしい。

```
CREATE TABLE InventoryReceipts
(receipt_nbr INTEGER PRIMARY KEY,
 purchase_date DATE NOT NULL,
 qty_on_hand INTEGER NOT NULL
     CHECK (qty_on_hand >= 0),
 unit_price DECIMAL (12,4) NOT NULL);
```

InventoryReceipts：在庫テーブル　receipt_nbr：受入番号　purchase_date：仕入日　qty_on_hand：入庫数　unit_price：仕入単価

InventoryReceipts

receipt_nbr	purchase_date	qty_on_hand	unit_price
1	2015-01-01	15	10.00
2	2015-01-02	25	12.00
3	2015-01-03	40	13.00
4	2015-01-04	35	12.00
5	2015-01-05	45	10.00

仮に、この製品が2015 年1 月5日に100 単位売れたとする。このとき、売れた部品の仕入原価をどうすれば計算できるだろうか？ 考え

られるやり方を以下に示す（あくまでも一例だ。やり方はほかにもある）。

方法1

最初は、2015年1月5日時点の取替原価を使う方法だ【訳注6】。この方法だと、現在は仕入単価が落ち込んでいるので10ドル/個で済む。部品の仕入原価の合計は1000ドルだ。

方法2

現時点での平均原価を用いる。現在は160個の在庫があり、その仕入れにトータルで1840 ドル支払ってきた。すると、部品の平均仕入単価は11.50ドルになり、100 個分で1150ドルになる。

方法3

LIFO、つまり「後入れ先出し（Last In, First Out）」方式を用いる。直近の仕入れから順に、時系列をさかのぼって計算する。

2015年1月5日仕入れ分：45個×10ドル＝450ドル（出荷数45個）

2015年1月4日仕入れ分：35個×12ドル＝420ドル（出荷数80個）

2015年1月3日仕入れ分：20個×13ドル＝260ドル（出荷数100個、余り20個）

したがって、売れた部品の仕入原価は1130ドルとなる。

方法4

FIFO、つまり「先入れ先出し（First In, First Out）」方式を用いる。一番古い仕入れから順に、時系列に沿って計算する。

2015年1月1日仕入れ分：15個×10ドル＝150ドル（出荷数15個）

2015年1月2日仕入れ分：25個×12ドル＝300ドル（出荷数40個）

2015年1月3日仕入れ分：40個×13ドル＝520ドル（出荷数80個）

2015年1月4日仕入れ分：20個×12ドル＝240ドル（出荷数100個、余り15個）

したがって、売れた部品の仕入原価は1210ドルとなる。

これら4 つの選択肢のうち、最初の2 つをプログラミングするのは簡単だ。LIFOとFIFO のケースはもっと手ごわい。なぜなら、毎日の入庫を先入れもしくは後入れの順番で検索し、要求された出庫数を満たす日付を探す必要があるからだ。

まずはLIFOのケースから考えてみよう。最初に、次のようなビューを作成する。

```
CREATE VIEW LIFO (stock_date, unit_price, tot_qty_on_hand, tot_cost)
AS
SELECT R1.purchase_date, R1.unit_price,
       SUM(R2.qty_on_hand), SUM(R2.qty_on_hand * R2.unit_price)
  FROM InventoryReceipts AS R1,
       InventoryReceipts AS R2
 WHERE R2.purchase_date >= R1.purchase_date
 GROUP BY R1.purchase_date, R1.unit_price;
```

LIFO：後入れ先出しビュー　stock_date：仕入日　unit_price：その日の仕入単価
tot_qty_on_hand：その日の在庫累計　tot_cost：その日までに支払った仕入金額合計

このビューの各行は、仕入日とその日の仕入単価、在庫の累計とその日までに支払った仕入金額の合計を表している。このビューを使うと、LIFO方式のコストは次のクエリで求められる。使っている計算は直接的だし、ロジックも簡単だ。

```
SELECT (tot_cost - ((tot_qty_on_hand - :order_qty_on_hand) * unit_price)) AS cost
  FROM LIFO AS L1
 WHERE stock_date
         = (SELECT MAX(stock_date)
              FROM LIFO AS L2
             WHERE tot_qty_on_hand >= :order_qty_on_hand);
```

ここでは、要求数（:order_qty_on_hand）を満たす（またはそれ以上の）在庫がある直近の日付を見つけ出さなければならない。偶然にも、ちょうど要求された数と同じ在庫を持つ日を見つけたら、その日までに支払った金額を売れた部品の仕入原価として返す。もし、在庫数が要求された数に届かなければ、何も返さない。

もし要求数以上の在庫数がある場合は、要求数を満たすだけの在庫数に達する日を現在から数えて調べ、その日における在庫残数（tot_qty_on_hand-:order_qty_on_hand）と仕入単価（unit_price）を掛けて、最後にその日までの仕入金額合計（tot_cost）から引き算する。

ビューの代わりに、導出テーブルとCASE式を使うこともできる。次に示すクエリでは、CASE 式により、在庫の累計が:order_qty_on_handを下回る日はその日の入庫数で仕入原価を計算し、最後に:order_qty_on_handを上回る日だけ必要な数を求

めてから仕入原価を計算している。外側のクエリは、`SUM(R3.v)`でこれらの計算結果を合計している。

```
SELECT SUM(R3.v) AS cost
  FROM (SELECT R1.unit_price *
                CASE WHEN SUM(R2.qty_on_hand) <= :order_qty_on_hand
                     THEN R1.qty_on_hand
                     ELSE :order_qty_on_hand - (SUM(R2.qty_on_hand) - R1.qty_on_hand)
                 END
          FROM InventoryReceipts AS R1, InventoryReceipts AS R2
         WHERE R1.purchase_date <= R2.purchase_date
         GROUP BY R1.purchase_date, R1.qty_on_hand, R1.unit_price
        HAVING (SUM(R2.qty_on_hand) - R1.qty_on_hand) <= :order_qty_on_hand) AS R3(v);
```

FIFOについても、類似の方法で計算することができる。

```
CREATE VIEW FIFO (stock_date, unit_price, tot_qty_on_hand,tot_cost)
AS
SELECT R1.purchase_date, R1.unit_price,
       SUM(R2.qty_on_hand), SUM(R2.qty_on_hand * R2.unit_price)
  FROM InventoryReceipts AS R1,
       InventoryReceipts AS R2
 WHERE R2.purchase_date <= R1.purchase_date
 GROUP BY R1.purchase_date, R1.unit_price;
```

クエリのロジックもほぼ同じである。

```
SELECT (tot_cost - ((tot_qty_on_hand - :order_qty_on_hand) * unit_price)) AS cost
  FROM FIFO AS F1
 WHERE stock_date = (SELECT MIN (stock_date)
                       FROM FIFO AS F2
                      WHERE tot_qty_on_hand >= :order_qty_on_hand);
```

【訳注1】　もう少し日本人にわかりやすい喩えとしては、被覆は「所属クラブ」、パーティションは「出身都道府県」などをイメージするとよい。所属クラブは一人がスキ

ーやテニスなど複数をかけもちすることがあるので重複するが、出身都道府県は重複なく分かれる。

【訳注2】　コッドの定義した8つの関係演算は次のとおり。制限（restrict）、射影（project）、積（product）、和（union）、交差（intersect）、差（difference）、結合（join）、除算（divide）。

【訳注3】　この問題は、第23章で見た「存在仮定」に似たところがある。2 機の飛行機を操縦できる飛行士にとって、0 機の飛行機を操縦することは、できて当然の仕事ではないだろうか？

【訳注4】　「Powerful SQL:Beyond the Basics」 http://gertjans.home.xs4all.nl/usenet/BeyondTheBasics.htm

【訳注5】　選言標準形は、式の形が規則的になるため、機械的な処理を行うのに向いている。

【訳注6】　取替原価とは、現時点で同じ資産を調達するのにかかるコスト。

28

グルーピング演算子

Grouping Operations

　グループを作る演算子とパーティションを作る演算子は似ているが、本章の議論にあたってはその違いを区別しておきたい。グループ演算子は、複数の行を1 つにまとめたうえで、各グループについて1行を返す。一方、パーティション演算子は、単純にもとのテーブルを複数の部分集合に切り分けるだけだ。SQLでは、グループを作る演算子は`GROUP BY`、パーティションを作る演算子は`PARTITION BY`である。

28.1

GROUP BY 句

GROUP BY句は単純なパーティションを基礎にしている。ある集合のパーティションは次のような性質を持つ部分集合のことである。

性質1　部分集合の和が元の集合になる。
性質2　部分集合同士の間に共通の要素は存在しない。

パーティションをイメージするには、丸いピザ生地を人数分に切り分けることを想像してほしい。SQL-99 ではOLAP 機能の拡張が行われ、GROUP BY 句にROLLUP やCUBE といった演算子を付け加えることもできるようになった。この拡張を使うと、単純なパーティション以外の集合も作ることができるが、だが、あくまで基本はパーティションである。

GROUP BY句はFROM句とWHERE句の結果を入力にとり、GROUP BY句に記述された列のリストを基準に同じ値を持つ行を同じグループに振り分けていく。その結果テーブルは集約テーブルと呼ばれる。GROUP BY句が集約を行うと、その後のすべての演算はオリジナルのテーブルの行ではなく、ここで定義されたグループに対して行われるようになる。

標準SQLの規約では、NULLは1つのグループを形成する。GROUP BY 句における列の並び順は重要ではないが、いくつかの列はSELECT句にも現れることが普通なので、GROUP BY 句とSELECT句の列の並びは揃えておくほうが読みやすいだろう。注意してほしいのは、

SELECT句に含まれる列はGROUP BY句の列の部分集合でありうるが、その逆は成り立たないことだ。

次のような簡単なテーブルを使って、GROUP BY句の仕組みを解説していこう。

```
CREATE TABLE Villes
(state_code CHAR(2) NOT NULL, -- usps codes
 city_name CHAR(25) NOT NULL,
     PRIMARY KEY (city_name, state_code));
```

Villes：villeで終わる都市テーブル　state_code：州コード　city_name：都市名

このテーブルに最後が「-ville」で終わる都市の名前を登録していく。最初の問題は州コードを使ってそのような名前を持つ都市の数を見つけることだ。すぐに思いつくクエリは次のようなものだろう。

```
SELECT state_code, city_name, COUNT(*)
  FROM Villes
 GROUP BY state_code;
```

テネシー州のグループには('TN', 'Nashville')や('TN', 'Knoxville')といった行が含まれるだろう。このクエリにおいて、州コードは集約キーであり、1 つのグループ内では不変である。COUNT(*)はグループ内の要素の数を数えるもので、これも明らかだ。問題は都市名である。このテーブルは州で集約されているために最大でも50 程度の行しか含まない。

COUNT(*)は明らかに単一の値であり、1つのグループに対して一意に決まる。ところが、各グループにおける都市名はどのような単

一の値を使えばよいのだろうか。その州に典型的な都市名を1つピックアップするべきだろうか。すべての都市が同じ名前を持っていたならそれも簡単なのだが。それともいっそのこと、NULLにすべきだろうか。おそらく最悪の選択肢はCOUNT(*)の数だけ行を出力することだろう。だがこれではそもそもグループではなくなってしまう。したがってこのクエリはエラーになる【訳注1】。

結果の各行が表すのは1 つのグループである。それゆえ、その行に含まれるものはグループの特性を表すものでなくてはならない。グループに含まれるある特定の行の性質を表すわけではないのだ。これが、SELECT 句に含めることができるのは集約キーと集約関数（および定数）だけというルールが存在する理由である。

28.1.1　NULL とグループ

現在、SQLはNULLを1つのグループとして扱う。このとき、NULL はあたかも互いに等しい値であるかのように見える。NULL の扱い方としてもう1 つ考えられるのは、1 つのNULL で1 つのグループを形成する方法だ。これは、実際に初期のSQL 製品のいくつかで採用された方法だ。これは思ったほどあり得ない選択肢ではない。だが、この2つの選択肢のどちらを選ぶかの判断を正しく行うには、モデル化しようとしているデータのセマンティクスを知る必要があるだろう。SQLはシンタックスに基づいた言語であって、セマンティクスには基づいていない。たとえば、もしNULLが医療記録における失われた診断のために使われていたら、そのNULLの値が明らかになったときには、各患者はそれぞれ異なる病気を持っていることが判明する

かもしれない。もし読者が未処理の診断報告書を1 つのグループとしてまとめたいと思うなら、NULLを1 つのグループにまとめるのも意味がある。一方、まだ未処理の診断報告書を関連する一群の病気の治療のためのアクションプランとして使いたい場合は、各NULLを個別のグループとして扱いたいと思うだろう。

もう1つ別の例を出そう。交通違反切符を格納するデータベースを考えよう。ここでNULLは失われた車両のタグを意味する。このデータベースにタグを持たない車が複数存在することは明らかだ。もしNULLを区別した場合、このデータベースに対するクエリはズバリ次のようになる。

```
SELECT x, ...
  FROM Table1
 WHERE x IS NOT NULL
 GROUP BY x
UNION ALL
SELECT x, ...
  FROM Table1
 WHERE x IS NULL;
```

また、集約キーがNULL になるケースもあるだろう。たとえば、タグを持たない車についてVIN（車両識別番号）を集約キーに使うような場合だ。

28.2

GROUP BY 句とHAVING 句

GROUP BY 句を使うときの最大の問題は、WHERE 句とHAVING 句の使い分け方だ。たとえば、プログラマの数が5人未満の部署をすべて見つけるクエリを考えよう。

```
CREATE TABLE Personnel
(emp_nbr   INTEGER NOT NULL,
 dept_nbr  INTEGER NOT NULL,
 job_title CHAR(16));
```

Personnel：社員テーブル　emp_nbr：社員番号　dept_nbr：部署番号
job_title：職種

Personnel

emp_nbr	dept_nbr	job_title
100	1	Programmer
101	1	Programmer
102	1	Programmer
103	1	Sales
104	1	Sales
105	1	Sales
106	1	Sales
107	2	Sales
108	2	Sales

108	2	Engineer
108	2	Engineer
108	2	Engineer
108	2	Engineer
108	2	Engineer

```
SELECT dept_nbr
  FROM Personnel
 WHERE job_title = 'Programmer'
 GROUP BY dept_nbr
HAVING COUNT(*) < 5;
```

結果

```
 dept_nbr
==========
        1
```

　このクエリの結果にはプログラマが1人もいない部署#2の行は含まれない。まずWHERE句が最初に実行されるので、プログラマでない社員の行はGROUP BY句に渡されないからだ。これは望んだ結果とは違うだろう。

　一方、次のクエリを見てほしい。これはプログラマが1人もいない部署も結果に含める。なぜなら、COUNT(DISTINCT x)は空集合に対しては0を返すからだ。

```
SELECT DISTINCT dept_nbr
  FROM Personnel AS P1
 WHERE 5 > (SELECT COUNT(DISTINCT P2.emp_nbr)
              FROM Personnel AS P2
             WHERE P1.dept_nbr = P2. dept_nbr
               AND P2.job_title = 'Programmer');
```

結果

```
 dept_nbr
==========
         1
         2
```

読者の中にはサブクエリよりはHAVING 句を使うことを考えた人もいるかもしれない。だがこのアプローチはなかなか難しい。HAVING 句は集約されたテーブルの行のみに適用されるため、集約キーと集約関数だけを参照する。これが次のクエリが失敗する理由だ。

```
SELECT dept_nbr --エラーになる
  FROM Personnel
 GROUP BY dept_nbr
HAVING COUNT(*) < 5
   AND job_title = 'Programmer';
```

このクエリにおいて、HAVING 句が実行されるとき、job_titleは集約されたテーブルの中には列としては含まれていない。言い換えれば、job_titleは行の性質であってグループの性質ではないのだ。次のクエリも同じ理由で失敗する。

```
SELECT dept_nbr --エラーになる
  FROM Personnel
 WHERE COUNT(*) < 5
   AND job_title = 'Programmer'
 GROUP BY dept_nbr;
```

COUNT(*)は部署のグループが作られるまでは存在していない。そのグループを作るのはGROUP BY句でありWHERE句でCOUNT(*)を参照することは、実行順序から言って不可能なのだ。HAVING句を使うときの正解はこうだ。

```
SELECT dept_nbr
  FROM Personnel
 GROUP BY dept_nbr
HAVING SUM(CASE WHEN job_title = 'Programmer'
                THEN 1 ELSE 0 END) < 5;
```

結果

```
 dept_nbr
==========
         1
         2
```

SUM関数はグループの性質であるため、問題なくHAVING句で参照することができる。その中でCASE式による特性関数を使えば、職種がプログラマの社員数をカウントできる。

28.2.1　グループの特性とHAVING 句

上でも見たように、集約関数とHAVING句を使うとGROUP BY句で作られたグループの特性を知ることができる。

HAVING 句を使うことで、以下に示すようなグループの性質を調べることができる。トム・モローはこの点について多くの示唆を与えてくれた。

- HAVING COUNT (DISTINCT col_x) = COUNT (col_x)

  col_xは一意である
- HAVING COUNT(*) = COUNT(col_x)

  col_xにはNULLが含まれていない
- HAVING MIN(col_x - <const>) = -MAX(col_x - <const>)

  col_xの最大値と最小値が指定した定数から同じ幅の距離にある
- HAVING MIN(col_x)* MAX(col_x)<0

  col_xの最大値と最小値のどちらかが負数であり、もう一方が正数である
- HAVING MIN(col_x)* MAX(col_x)> 0

  col_xはすべて正数かすべて負数のどちらかである
- HAVING MIN(SIGN(col_x)) = MAX(SIGN(col_x))

  col_xはすべて正数かすべて負数かすべて0である
- HAVING MIN(ABS(col_x)) = 0

  col_xは少なくとも1つの0を含む
- HAVING MIN(ABS(col_x)) = MIN(col_x)

  col_xは0以上である（この条件はWHERE句で書いたほうがよい）

- HAVING MIN(col_x) = -MAX(col_x)

  col_xの最大値と最小値が0から同じ幅の距離にある
- HAVING MIN(col_x) * MAX(col_x) = 0

  col_xの最大値または最小値のどちらかまたは両方が0である
- HAVING MIN(col_x) < MAX(col_x)

  col_xは一意ではない（この条件はcount(*)> 1と同じだがこちらのほうが速いかもしれない）
- HAVING MIN(col_x) = MAX(col_x)

  col_xは一意であるかすべてNULLである
- HAVING (MAX(seq) - MIN(seq) +1) = COUNT(seq)

  seqは歯抜けのない連番である

なお、GROUP BY 句がなければHAVING 句はテーブル全体を1 つのグループとして扱う。SQL の昔の実装ではGROUP BY句なしでHAVING句を使うことはできなかった。現在ではこの制約はないが、いまでもそのような前提で書かれている古いコードを見る機会があるかもしれない。

このことから、テーブル全体の性質をHAVING 句で調べようと思ったら、SELECT 句には集約関数と定数しか書けないということになる。これはサンプルを見て理解するほうが早い。以下のような1 から始まる一意な連番の列を持ったテーブルがあるとする。ここに新しい行を挿入する。ただし、テーブルに既存の数は使ってはならない。もし連番に歯抜けがあれば、その歯抜けを埋めなければならな

い。歯抜けがない場合は、単純に連番の最大値よりも1 大きい数を挿入する。

```
CREATE TABLE Batting_Lineup
(batting_seq INTEGER NOT NULL PRIMARY KEY
     CHECK (batting_seq > 0),
 player_name CHAR(5) NOT NULL);
```

Batting_Lineup：打順テーブル　　batting_seq：打順　　player_name：選手名

```
INSERT INTO Batting_Lineup VALUES (1, 'Tom'), (2, 'Dick'), (4, 'Harry'), (5, 'Moe');
```

　このテーブルから、どうすれば歯抜けを見つけられるだろうか？それには、次のようにHAVING句を応用すればいい。このクエリは歯抜けがある場合は'gap'という定数を返す。なければ1行も返さない。

```
SELECT 'gap'
  FROM Batting_Lineup
HAVING COUNT(*) <> MAX(batting_seq);
```

　このサブクエリ内のSELECT 句で'SELECT batting_seq'のように列を含めることはできない。これは、GROUP BY句を使っていないため、テーブルが1つのグループに集約されるからだ。もちろん、'SELECT *'も使えない。*は単に列名に展開されるだけだ。

　このトリックを使うと、歯抜けを埋めるINSERT文を作ることができる【訳注2】。このINSERT文は、歯抜けがない場合は最大値＋1を、歯抜けがある場合は、一番小さい歯抜けを挿入する。この場合は3だ。

```
INSERT INTO Batting_Lineup (batting_seq, player_name) VALUES
(CASE WHEN EXISTS -- 歯抜けがない場合
           (SELECT 'no gaps'
              FROM Batting_Lineup
             HAVING COUNT(*) = MAX(batting_seq))
      THEN (SELECT MAX(batting_seq) FROM Batting_Lineup) + 1
      ELSE (SELECT MIN(batting_seq + 1) -- 歯抜けがある場合
              FROM Batting_Lineup
             WHERE (batting_seq + 1) NOT IN (SELECT batting_seq FROM Batting_Lineup)
               AND batting_seq > 0)
  END, 'Celko');
```

このクエリのトリックは、ELSE句で巨大なスカラサブクエリを計算していることだけだ。なお、SELECT句でMIN(batting_seq + 1)やMIN(batting_seq)+ 1と書くと、インデックスを使えない製品もあるかもしれない。

28.3

集約の階層

SQLでは、こんなふうに集約関数を入れ子にすることはできないルールになっている。

```
SELECT depart_name, MIN(COUNT(stapler_nbr)) -- 違法な構文！
  FROM Companies
 GROUP BY depart_name;
```

このように集約関数を入れ子にするときの主な目的は、集約の階層を上げたい場合である。

この例の場合は、各部署にあるホッチキスの数の最小値を求めようとしているのだろう。だがこのクエリは意味をなさない。なぜなら、部署ごとにグループ化しており、部署のホッチキスの数、最大値、最小値、平均といった統計的性質はグループにつき1つの値しか持たない。このクエリは、グループの性質を1つのスカラ値に縮約してしまっており、記述統計の本義に反する。

28.3.1　グループ化したビューによる集約の階層化

業務的なレポートでは、集約を階層化した構造を持つものが少なくない。この種のレポートは非常によく使われるので、こうしたレポートを作るのに特化したツールも存在する。たとえば、売上げはまずセールスマン個人で集約される。次に、セールスマンの属する部署で集約され、さらにその部署の属する地域で集約される。最後には、会社全体のレベルで集約されるだろう。各レベルは、1つ上位のレベルのパーティションになっている。

現在のSQLはCUBEとROLLUPオプションを持っており、このレポートをSQLで実現することも可能だ。このオプションについては第30章で詳しく見ていくが、率直に言って、SQL のこうした機能を使って階層化するよりも、レポート作成ツールでやるほうが速いし高機能である。

もしあえてSQLでやるというなら、1つのトリックは、GROUP BY句を持ったビューを使ってレポーティングのレベルを定義することだ。先ほどの売上げレポートの例であれば、次のUNIONを多用したクエリで各レベルのレポートを作成できる。最も詳細な下位レベルはセールスマンであり、最も上位レベルが会社である。

```
CREATE TABLE Sales
(region_nbr   CHAR(16),
 district_nbr CHAR(16),
 salesman     CHAR(16),
 sales_amt    INTEGER);
```

Sales：売上げテーブル　region_nbr：地域番号　district_nbr：地区番号
salesman：セールスマン　sales_amt：売上げ額

```
SELECT region_nbr, district_nbr, salesman, SUM(sales_amt)
  FROM Sales
 GROUP BY region_nbr, district_nbr, salesman
UNION
SELECT region_nbr, district_nbr, '{SALESMEN}', SUM(sales_amt)
  FROM Sales
 GROUP BY region_nbr, district_nbr
UNION
SELECT region_nbr, '{OFFICE}', '{SALESMEN}', SUM(sales_amt)
  FROM Sales
GROUP BY region_nbr
UNION
SELECT '{REGION}', '{OFFICE}', '{SALESMEN}', SUM(sales_amt)
  FROM Sales;
```

{}で囲まれた定数は、ASCIIでは任意のアルファベットよりも下にソートされるので、グループ内で常に最後に位置する。これがSQLで階層レポートを出力する方法であるが、あまり良い方法ではない。以下、この問題をSQLで解くことのさまざまな難点を解説しよう。

28.3.2　サブクエリによる集約の階層化

標準SQLでは、FROM句でサブクエリをテーブルとして利用できるし、式を記述できる場所ならどこでもスカラサブクエリを利用できる。両者を応用すると、1つのクエリで多階層の集約を記述できる。たとえば、各地域の売上げの合計に対して、各セールスマンがどの程度売り上げたかを調べるには次のように書く。

```
SELECT salesman, region_nbr, district_nbr, SUM(sales_amt) AS salesman_tot,
      (SELECT SUM(sales_amt)
         FROM Sales AS S1
        WHERE S1.region_nbr = S2.region_nbr
          AND S1.district_nbr = S2.district_nbr) AS district_nbr_tot
  FROM Sales AS S2
 GROUP BY salesman, region_nbr, district_nbr, district_nbr_tot;
```

このクエリが正しく動作するのは、サブクエリが各グループに対して定数となるからである。やろうと思えば、このサブクエリを式の入力に取ることで、地域内で各セールスマンの売り上げた割合を計算することもできる。

ここでちょっとトリッキーな問題を考えよう。それは、集約の集約を求めることだ。たとえば、地域ごとの売上げの平均を求める、といったような。SQL 初心者はこういうとき、深く考えず次のようなクエリを書いて失敗する。

```
SELECT region_nbr, AVG(SUM(sales_amt)) AS region_nbr_average -- 正しくない
  FROM Sales
 GROUP BY district_nbr, region_nbr;
```

このクエリを見ると、パーサはAVG(SUM(sales_amt))のところでつまずいて、「集約関数を入れ子にすることはできない」というエラーメッセージを返す。標準SQLで望む結果を得るには、もう少し込み入った書き方をしないといけない。region_nbr内におけるdistrict_nbrごとの売上げ合計を計算するにはサブクエリを使う必要がある。その結果に対して、region_nbr ごとに平均を計算するのだ。答えはこうである。

```
SELECT T1.region_nbr, AVG(T1.district_nbr_total) AS region_nbr_average
  FROM (SELECT region_nbr, district_nbr, SUM(sales_amt)
          FROM Sales
         GROUP BY region_nbr, district_nbr)
           AS T1 (region_nbr, district_nbr, district_nbr_total)
 GROUP BY T1.region_nbr;
```

おそらくサブクエリT1は一度実行されて実体化された後、それがSELECT 文で利用されるだろう。なお、標準SQLでは次のような書き方はできないので注意してほしい。

```
--正しくないSQL
SELECT region_nbr, AVG(SELECT SUM(sales_amt)
                         FROM Sales AS S1
                        WHERE S1.region_nbr = S2.region_nbr
                        GROUP BY district_nbr) AS region_nbr_average
  FROM Sales AS S2
 GROUP BY region_nbr;
```

これが良くない理由は、集約関数の結果を他の集約関数やサブクエリの入力に取ることができないからだ。これが禁止されている理

由は、サブクエリが一般に複数の行、複数の列を返すことがあるからである（このケースでは、たまたまスカラ値を返すが）。

28.3.3　CASE 式による集約の階層化

集約関数を入れ子にする別のトリックを紹介しよう。映画の名前とレビュアーによる点数を記録するテーブルを考える。このテーブルを使って次のような問いに答えたい。

Q1　各映画の最低点は何点か。また最低点を付けた評論家は何人か。

Q1　各映画の最高点は何点か。また最高点を付けた評論家は何人か。

サンプルデータは次のとおりだ。最高点が5、最低点が1としよう。

```
CREATE TABLE Reviews
(movie_name    CHAR(16),
 reviewer_name CHAR(16),
 rating        INTEGER,
     PRIMARY KEY(movie_name, reviewer_name));
```

Reviews：レビューテーブル　movie_name：映画タイトル　reviewer_name：評論家　rating：評価

Reviews

movie_name	reviewer_name	rating
001	A	1
001	B	1

001	C	3
002	A	2
002	B	4
002	C	4
002	D	2
003	A	4

これを求めるクエリはこうだ。

```
SELECT movie_name,
       MIN(rating) AS worst_rating,
       COUNT(CASE WHEN NOT EXISTS
                           (SELECT *
                              FROM Reviews AS R2
                             WHERE R2.movie_name = R1.movie_name
                               AND R2.rating < R1.rating)
                  THEN 1 ELSE NULL END) AS worst_tally,
       MAX(rating) AS best_rating,
       COUNT(CASE WHEN NOT EXISTS
                           (SELECT *
                              FROM Reviews AS R3
                             WHERE R3.movie_name = R1.movie_name
                               AND R3.rating > R1.rating)
                  THEN 1 ELSE NULL END) AS best_tally
  FROM Reviews AS R1
 GROUP BY movie_name;
```

結果

```
 movie_name  worst_rating  worst_tally  best_rating  best_tally
============================================================
 001                    1            2            3           1
 002                    2            2            4           2
 003                    4            1            4           1
```

2つのNOT EXISTS述語内のサブクエリは、それぞれ映画ごとの最低点と最高点を求めている。その結果は複数行ありうるので、外側のCOUNT 関数で数えている。こうすることで、集約関数を入れ子にすることを防止している。

28.4

計算列によるグルーピング

SQL-99 では計算列によるグループ化が可能になった。たとえば、月別の売上げを計算するにはこう書く。

```
SELECT EXTRACT(MONTH FROM sale_date) AS rpt_month,
       SUM(sales_amt)
  FROM Sales
 GROUP BY rpt_month;
```

これは、列名だけしか集約キーとして認めていなかった古い標準とはかなり異なる【訳注3】。もともとのモデルでは、SELECT 文は SELECT 句を最後に実行するため、そこで計算されて作られた列を、他の句で参照することはできなかった（存在しないものを参照することはできない道理だ）。

もっとも、SQL-99 以前のSQL においても、サブクエリをFROM 句で使うことで、同じようなことを擬似的に実現できた。

```
SELECT salesmonth, SUM(sales_amt)
  FROM (SELECT EXTRACT(MONTH FROM sale_date) AS salesmonth, sales_amt
          FROM Sales) AS MonthSales
         GROUP BY salesmonth;
```

あるいは、相関サブクエリをSELECT句で使うという方法もある。

```
SELECT DISTINCT EXTRACT(MONTH FROM S1.sale_date),
      (SELECT SUM(S2.sales_amt)
         FROM Sales AS S2
        WHERE EXTRACT(MONTH FROM S2.sale_date) = EXTRACT(MONTH FROM S1.sale_date))
  FROM Sales AS S1;
```

両者を比較すると、おそらく前者のほうが計算量は少なく速度的には有利だろう。だがもちろん、単に計算列を使う方法に比べれば

どちらも高コストだ。

28.5

GROUP BY 句でペアを作る

これは言葉で説明するよりサンプルを見たほうが早い。人名と性別のテーブルがあり、そこから夕食会に招待する男性と女性のペアを作りたい。つまり、適当なルールで男女の1対1対応を作るということだ。

```
CREATE TABLE People
(person_name VARCHAR(15) NOT NULL,
 sex_code INTEGER DEFAULT 1 NOT NULL -- ISO標準の性別コード
     CHECK (sex_code IN (1, 2)));
```

People：人名テーブル　　person_name：人名　　sex_code：性別コード

```
--男性
INSERT INTO People VALUES ('Bob', 1), ('Ed', 1), ('Joe', 1), ('Dave', 1);

--女性
INSERT INTO People VALUES ('Sue', 2), ('Joan', 2), ('Kate', 2), ('Mary', 2),
('Petra', 2), ('Nancy', 2);
```

次は、スティーブ・キャスからの解答だ。`rank` 列が同じ男女がカップリングされる。女性のほうが2人多いので、PetraとSueはあぶれることになる。

```
SELECT A.person_name, A.sex_code, COUNT(*) AS rank
  FROM People AS A, People AS B
 WHERE B.person_name <= A.person_name
   AND B.sex_code = A.sex_code
 GROUP BY A.person_name, A.sex_code;
```

結果

```
 person_name  sex_code  rank
============================
 Bob                 1     1
 Joan                2     1
 Dave                1     2
 Kate                2     2
 Ed                  1     3
 Mary                2     3
 Joe                 1     4
 Nancy               2     4
 Petra               2     5
 Sue                 2     6
```

それぞれの人に対して、`COUNT(*)`は同じ性別の名前だけを数えてアルファベット順の「順位」をつける。自己結合によって、同じ性別で自分よりもアルファベット順で「前」の人とだけペアリングされる。その次に、結果をこの「順位」と性別コードの順で順序づけし、マッチするランクの名前を順序づける。

これは正しい結果を返すが、惜しいのは一緒に夕食会に行くカップルが異なる行に表示されることだ。これだと、1行を見ただけで「今日はデートなの？」という質問に答えることはできない。基本的なアイデアは正しいので、カップルを1行にまとめる方法を考えよう。

```
SELECT M1.person_name AS male, F1.person_name AS female,
       COALESCE (M1.rank, F1.rank) AS rank
  FROM (SELECT P1.person_name, COUNT(P2.person_name)
          FROM People AS P1, People AS P2
         WHERE P2.person_name <= P1.person_name
           AND P1.sex_code = 1
           AND P2.sex_code = 1
         GROUP BY P1.person_name) AS M1(person_name, rank)
       FULL OUTER JOIN
        (SELECT P1.person_name, COUNT(P2.person_name)
           FROM People AS P1, People AS P2
          WHERE P2.person_name <= P1.person_name
            AND P1.sex_code = 2
            AND P2.sex_code = 2
          GROUP BY P1.person_name) AS F1(person_name, rank)
    ON M1.rank = F1.rank;
```

結果

```
 male  female  rank
====================
 Bob   Joan       1
 Dave  Kate       2
 Ed    Mary       3
 Joe   Nancy      4
       Petra      5
       Sue        6
```

この解は、2つの部分集合をアルファベット順に並べることでペアができることを前提としている。また、男性と女性の数が違う場合は、NULLで埋めてやらなければならない。

もう少し現代風にウィンドウ関数を使って書き直すとこうなる。

```
SELECT M1.person_name AS male, F1.person_name AS female,
       COALESCE (M1.rank, F1.rank)
  FROM (SELECT P1.person_name,
               ROW_NUMBER() OVER(ORDER BY P1.person_name) AS rank
          FROM People AS P1
         WHERE P1.sex_code = 1) AS M1
       FULL OUTER JOIN
          (SELECT P2.person_name,
                  ROW_NUMBER() OVER(ORDER BY P2.person_name) AS rank
             FROM People AS P2
            WHERE P2.sex_code = 2) AS F1
         ON M1.rank = F1.rank;
```

こうすると、ペアリングで自己結合を使う必要がなくなる。おそらくこちらのほうがパフォーマンスは良いだろう。

28.6

GROUP BY句とソート

標準SQLでは要求されていないが、大半の実装では集約されたクエリの結果は自動的にソートされている。これは集約を行う際、内部的にはテーブルに対して集約キーでソートが行われるからである。NULLグループは先頭または末尾に位置付けられる（ベンダーによって異なる）。したがって、GROUP BY 句のキーではないキーがORDER BY 句に存在すると、オプティマイザは再度ソートを実行する必要があるため、コストが高くなる可能性がある。

ORDER BY句では、集約関数や計算列をキーに使うことも可能である。たとえば、総売上順に地区を表示するなら、次のように書けばよい。

```
SELECT region_nbr, district_nbr, SUM(sales_amt) AS district_sales_amt
  FROM Sales
GROUP BY region_nbr, district_nbr
ORDER BY district_sales_amt, region_nbr;
```

2つの地域が同じ売上げであることもありうるので、その場合でも順序を一意に固定するためにも、region_nbr列もORDER BY句のキーに含めておくのがよいだろう。

ソートには安定ソートと非安定ソートがあるが、SQL の実装では非安定ソートが使われることが多い。安定ソートは、同じ値に関して、ソート前の行の順序がソート後も維持されているソートであり、非安定ソートはその逆だ。たとえば、トランプのカードを数字と絵柄でソートするとしよう。最初に数字でソートすれば、エースが最大になるので、まずすべての絵柄の2、その次にすべての絵柄の

3……、以下すべての絵柄のエースまで続く。このとき、同じ数字のグループにおいて、絵柄の順番は任意である。

次に、この結果を絵柄に基づいて降順でソートすれば（ブリッジのソート順を想定している）、クラブ、ダイヤ、ハート、スペードの順にソートされるだろう。このとき、もし2番目のソートが非安定ソートであれば、数字の順序を破壊する可能性がある。一方もし安定ソートならば、数字の順序も保存される。

安定ソートはほぼ必ず非安定ソートより遅い。したがって、多くのデータベースは非安定ソートを好む。だが、賢いオプティマイザならば、中間のワークテーブルに存在する順序を見つけて、それを使って非安定ソートではなく安定ソートを行うこともある。そうすれば、余計な計算をスキップできる。オプティマイザはまた、クラスタ化インデックスなど、データに存在する他の既存の順序を利用することもある。とはいえ、特定の実装のデフォルトの順序に依存するべきではない。移植性がないし、いつ変更されるかわからないからだ。

もし結果の表示される順序が重要ならば、（たとえGROUP BY 句を使っていても）明示的にORDER BY 句を使用するべきである。なお、一般的なベンダー拡張として、SELECT 句の列順序を表す整数をORDER BY 句で使える機能があるが、これはSELECT 句の列順が変わった場合にORDER BY 句を変更し忘れる危険があるし、標準SQLで削除予定の機能とされているため、今後は使うべきではない。

【訳注1】　このクエリはMySQLでは動作する。

【訳注2】　このINSERT 文は、すべての場合に対応するにはまだ不完全である。たとえば、1が欠番の場合にそれを挿入することができない。この欠番を補うトリックについてより詳細を知りたい読者は、拙著『達人に学ぶ SQL 徹底指南書』の「1-3　3値論理とNULL」および「1-10 帰ってきたHAVING句」を参照してほしい。

【訳注3】　この書き方が許されている実装は、現在のところPostgreSQL、MySQLなど一部である。

29

単純な集約関数

Simple Aggregate Functions

単純な集約関数は、SQL が誕生したときから存在する機能であり、これを持っていない製品は存在しない。多くの場合、オプティマイザは集約関数のためのデータを利用して実行パスを考えるし、インデックスも利用できることが多いため、実行速度も速い。

単純な集約関数は、まずパラメータによって与えられた値の列を作る。パラメータは普通、単一の列名だが、スカラ関数や計算を含む式を与えることもできる。パラメータに与えることができないのは、他の集約関数（たとえば、`SUM(AVG(x))`は違法だ）とサブクエリ（たとえば、`AVG(SELECT col1 FROM SomeTable WHERE ...)`も違法）だけである。サブクエリが許されないのは、非スカラ値を返す可能性があり、その場合は集約関数が列の中で列を作ることになってしまうからである。

この作業用の列が作られたら、次にすべての`NULL`が除外され、計算が実行される。ここは非常に重要なポイントだ。集約関数が計算の前に`NULL`を除外するという点は、是非覚えておいてほしい。

なお、集約関数には、`ALL`と`DISTINCT`というオプションがある。`ALL`はデフォルトの動作で、パラメータリストの値に重複があってもすべて保持する。デフォルトなのでわざわざ指定することは、現実

にはない。DISTINCTは、計算の前に重複値を除外してパラメータリストを一意にする。

　では、どんな集約関数があるか、具体的に見ていこう。大きく3つのカテゴリに分類される。

1. **数学的演算**：平均と合計。これらはすべての数値型に対して適用できる。
2. **極値関数**：最小と最大。これらは順序を持つすべてのデータ型に対して適用できる。
3. **カウント**：テーブルの行数を数える。

29.1

COUNT 関数

COUNT関数には2種類ある。カーディナリティと式のカウントである。COUNT(*)は、テーブルの行数（これを関係モデルの用語で「カーディナリティ」と呼ぶ）を返す。ちなみにこれは、標準SQLの関数の中で唯一アスタリスクを入力に取る関数である。これは非常に便利であるだけでなく、テーブルサイズについてのシステム情報を利用できるため非常に高速である。重要な点は、この関数はNULLも含めて数えることである。なぜなら、この関数はあくまで行全体を対象にするのであって、特定の列値だけを数えているわけではないからだ。

空のテーブルに対するCOUNT(*)は0を返す。これ自体は決しておかしいことではないが、他のすべての集約関数は、空のテーブルに対しては空集合を返す。ラテン語の格言「無からは無しか生じない」のとおりである。いまさらSQLの構文を変更するのは手遅れだが、カーディナリティを求める場合の入力には（EXISTS述語でやるように）テーブル式を使う構文があったほうが、便利だったかもしれない。

読者は、COUNT(*)の使い方は簡単だと思っているかもしれない。だがこの関数にはたくさんの精妙なトリックが存在する。いくつかサンプルを使って示そう。まず、米国大統領の情報を管理するテーブルを考える。姓、名、ミドルネーム（イニシャル）、政党といった列を持っている。また、彼らの属する政党のテーブルもあるとする。

```
CREATE TABLE Presidencies
(first_name CHAR(11) NOT NULL,
 initial VARCHAR(4) DEFAULT ' ' NOT NULL, -- デフォルトはスペース１文字
 last_name CHAR(11) NOT NULL,
 party_code CHAR(2) NOT NULL
     REFERENCES Parties(party_code),
 start_term_year INTEGER NOT NULL
     CHECK (start_term_year >= 1789),
 end_term_year INTEGER); -- NULLは現職を意味する
```

Presidencies：大統領テーブル　first_name：名　initial：イニシャル
last_name：姓　party_code：政党コード　start_term_year：任期開始年
end_term_year：任期終了年

```
CREATE TABLE Parties
(party_code CHAR(2) NOT NULL PRIMARY KEY,
 party_name VARCHAR(25) NOT NULL);
```

Parties：政党テーブル　party_code：政党コード　party_name：政党名

```
INSERT INTO Parties
VALUES ('D', 'Democratic'),
       ('DR', 'Democratic Republican'),
       ('R', 'Republican'),
       ('F', 'Federalist'),
       ('W', 'Whig'),
       ('L', 'Libertarian');
```

Presidencies

```
first_name      initial  last_name      party  start_term_year  end_term_year
=============================================================================
'George'        ''       'Washington'   'F'    1789             1797
'John'          ''       'Adams'        'F'    1797             1801
'Thomas'        ''       'Jefferson'    'DR'   1801             1809
'James'         ''       'Madison'      'DR'   1809             1817
'James'         ''       'Monroe'       'DR'   1817             1825
'John'          'Q.'     'Adams'        'DR'   1825             1829
'Andrew'        ''       'Jackson'      'D'    1829             1837
'Martin'        ''       'Van Buren'    'D'    1837             1841
'William'       'H.'     'Harrison'     'W'    1841             1841
'John'          ''       'Tyler'        'W'    1841             1845
'James'         'K.'     'Polk'         'D'    1845             1849
'Zachary'       ''       'Taylor'       'W'    1849             1850
'Millard'       ''       'Fillmore'     'W'    1850             1853
'Franklin'      ''       'Pierce'       'D'    1853             1857
'James'         ''       'Buchanan'     'D'    1857             1861
'Abraham'       ''       'Lincoln'      'R'    1861             1865
'Andrew'        ''       'Johnson'      'R'    1865             1869
'Ulysses'       'S.'     'Grant'        'R'    1869             1877
'Rutherford'    'B.'     'Hayes'        'R'    1877             1881
'James'         'A.'     'Garfield'     'R'    1881             1881
'Chester'       'A.'     'Arthur'       'R'    1881             1885
'Grover'        ''       'Cleveland'    'D'    1885             1889
'Benjamin'      ''       'Harrison'     'R'    1889             1893
'Grover'        ''       'Cleveland'    'D'    1893             1897
'William'       ''       'McKinley'     'R'    1897             1901
'Theodore'      ''       'Roosevelt'    'R'    1901             1909
'William'       'H.'     'Taft'         'R'    1909             1913
'Woodrow'       ''       'Wilson'       'D'    1913             1921
'Warren'        'G.'     'Harding'      'R'    1921             1923
'Calvin'        ''       'Coolidge'     'R'    1923             1929
'Herbert'       'C.'     'Hoover'       'R'    1929             1933
'Franklin'      'D.'     'Roosevelt'    'D'    1933             1945
'Harry'         'S.'     'Truman'       'D'    1945             1953
'Dwight'        'D.'     'Eisenhower'   'R'    1953             1961
'John'          'F.'     'Kennedy'      'D'    1961             1963
'Lyndon'        'B.'     'Johnson'      'D'    1963             1969
'Richard'       'M.'     'Nixon'        'R'    1969             1974
'Gerald'        'R.'     'Ford'         'R'    1974             1977
'James'         'E.'     'Carter'       'D'    1977             1981
'Ronald'        'W.'     'Reagan'       'R'    1981             1989
'George'        'H.W.'   'Bush'         'R'    1989             1993
'William'       'J.'     'Clinton'      'D'    1993             2001
'George'        'W.'     'Bush'         'R'    2001             2009
'Obama'         'H.'     'Barack'       'D'    2009             NULL
```

さて、社会の先生からいままでに米国大統領は何人いたか、という質問を受けたとする。読者は、「`SELECT COUNT(*) FROM`

Presidencies;」のようなクエリを書くかもしれないが、これは間違いだ。グローバー・クリーブランドが複数行登場しているからだ。彼は連続ではない2期を務めた唯一の大統領だ。つまり、このテーブルは1行が1人に対応していないのだ。このような場合は単純にCOUNT(*)を使うのではなく、一意な複数列の組み合わせを求めなければならない。

ポイントは、COUNT関数の入力に与えている式が本当に数えたい対象かどうか、ということだ。COUNT([ALL] <value expression>)は、<value expression>集合の要素の数を返す。NULL は計算の前に除外されるのでカウントされない。そのため、空集合に対しては0が返る。この関数を自然言語（ここでは日本語）で読み下すと、「この式の判明している値の数を数えろ」が正しい。この例では、大統領の数を数えるには、COUNT(first_name || ' ' || initial || ' ' || last_name)としなければならない。だが、これだけでは、まだクリーブランドの重複行を除外できていない。

この重複に対処するには、COUNT(DISTINCT <value expression>)関数を使う必要がある。これは、<value expression>集合の一意な要素の数を返す。やはりNULL は計算の前に除外され、その後で冗長な行が除外される（つまり、1つの値にまとめられる）。この場合においても、空集合に対しては0が返されるのは同じである。

当然のことだが、COUNT(DISTINCT <value expression>)は主キーまたは一意な列に対して適用すると、COUNT(*)を使ったのと

同じ結果になる。COUNT 関数におけるALLとDISTINCT の使い方は、SELECT句における[ALL | DISTINCT]オプションと同じパターンに従う。これは、SQLの共通的なパターンだ。

29.1.1　DISTINCT 付きの集約関数を最適化する

このトリックは、イツァク・ベン-ガンが考えた。うまくいくかどうかは、読者の使う製品のオプティマイザに依存するところもある。

まず、次のようなDISTINCT 付きのCOUNT 関数を使った普通のSELECT 文を考える。このパフォーマンスを最適化することを考える。

```
CREATE TABLE Sales
(cust_id    INTEGER,
 order_date DATE,
 order_amt  INTEGER,
     PRIMARY KEY(cust_id, order_date));
```

Sales：売上げテーブル　cust_id：顧客ID　order_date：注文日　order_amt：注文金額

```
SELECT cust_id, COUNT(*) AS order_cnt,
       COUNT(DISTINCT order_date) AS order_day_cnt,
       SUM(order_amt) AS order_amt_tot
  FROM Sales
 GROUP BY cust_id;
```

このクエリは、cust_idでグルーピングし、顧客ごとの注文数の合計、一意な注文日の数、注文金額の合計を返す。オプティマイザの中には、COUNT(*)とCOUNT(DISTINCT order_date)を計算するのに別々にテーブルをスキャンするものがある【訳注1】。これを1

つのスキャンだけで行うようにするには、次のように共通表式を使う方法がある。顧客と注文日でグルーピングし、それをより高次の集約で利用するのだ。こうすると、DISTINCTを使う必要がない。

```
WITH CustomerDailySales (cust_id, order_date, daily_order_cnt, order_amt_tot)
AS
(SELECT cust_id, order_date, COUNT(*), SUM(order_amt)
   FROM Sales
  GROUP BY cust_id, order_date)
SELECT cust_id,
       SUM(daily_order_cnt) AS daily_order_cnt,
       COUNT(order_date) AS order_day_distinct_cnt,
       SUM(order_amt_tot) AS order_amt_tot
  FROM CustomerDailySales
 GROUP BY cust_id;
```

　このクエリにおいては、オプティマイザは基底テーブルを一度しかスキャンしない。ただし、集約を2回行うことになるため、最初のクエリより本当に速いかどうかはやってみないとわからない。

29.2

SUM 関数

SUM関数は数値型に対してのみ使用できる。厳密な数値型と近似的な数値型に対して利用したときの結果の精度は実装依存であるため、これについては製品のマニュアルを見てほしい。

SUM([ALL] <value expression>)は、判明しているすべての値の合計を返す。NULLは計算の前に除外される。空集合に対しては、0ではなく空集合を返す。これはCOUNT関数と違うところだ。もしSELECT句に他の列が存在する場合は、空集合はNULLに変換される。

SUM(DISTINCT <value expression>)は、判明しているすべての一意な値の合計を返す。

NULLおよび冗長な値は計算の前に除外される。空集合に対しては、0ではなく空集合を返す。

この空集合に対するルールは、なかなか理解しがたいものである。特に、SELECT句に他の列が存在する場合に空集合がNULLに変換されるというルールはわかりにくい。だが、これは他の標準的な集約関数においてもそうなのだ。

```
-- 結果は1行も返さない
SELECT SUM(x)
  FROM EmptyTable;

--結果は(0, NULL)という1行だけ
SELECT COUNT(*), SUM(x)
  FROM EmptyTable;
```

数値の和は極めて単純に見えるが、精度に関して注意が必要だ。正数と負数からなる近似的な数値型の集合があるとしよう。このま

ったく同じ集合を2 つのテーブルに投入する。ただし、片方はランダムな順序で、もう一方は絶対値でソートする。この場合、合計の結果は後者のテーブルのほうがより正確になる。理由は単純で、絶対値が同じ大きさを持つ正数と負数は互いに打ち消しあって0になるので、誤差が生じないからだ。また、ソートしたテーブルのほうがオーバーフローやアンダーフローのエラーを起こす確率も低い。もっとも、オープン系であれメインフレームであれ、多くの実装はSUMを実行するたびにソートを行うので、この問題に悩む必要はない。

厳密な数値型を使う場合、厳密な数にせよ近似的な数にせよ、保持したい数値が型の許容する精度に収まらないことがある。このときデータベースエンジンは、丸めまたは切り捨て（どちらを行うかは実装依存だ）を行ったあとに、もとの数字を有効桁数まで保存するよう近似する。このことは普段は意識されないが、実装間でデータ移行を行ったり、ホスト言語からSQLのテーブルにデータを移すときに驚くことがある。これを防止するため、テーブルの数値列を必要とされるよりも1つ多い小数桁で定義しておくのが安全だろう。

SQL では、切り捨ては0 へ近づくように切り捨てると定義されている。つまり、1.5 は1 に、21.5は21になる。だが、すべてのプログラミング言語がこのルールに従っているわけではない。正数については異論ある人はいないだろうが、負数については、0から遠ざかるように切り捨てるケースもある。たとえば、–1.5は–2になる。SQL は、丸めについては仕様を定めておらず、完全に実装に任せている。10.2.1 項で論じたように、丸めには主に科学的丸めと商業的丸めの2種類がある点にも注意してほしい。

29.3

AVG 関数

AVG([ALL] <value expression>) は<value expression>集合の平均を返す。空集合に対しては空集合を返す。すべてNULL の集合は空集合に変換される。AVG(x) は一般に、(SUM(x)/COUNT(*))と同値ではない。SUM(x)関数はNULLを除外するが、COUNT(*)はそうではないからだ。

AVG(DISTINCT <value expression>)は、<value expression>集合の中の判明している一意な値の平均を返す。空集合に対しては空集合を返す。主キーや一意な列に適用すると、AVG(<value expression>)と同じ結果になる。

AVG(DISTINCT x)は一般に、AVG(x)や(SUM(DISTINCT x)/COUNT(*))と同値ではない。SUM(DISTINCT x)は重複値とNULLを除外するが、COUNT(*)はそうではないからだ。

SQLエンジンは、AVGにおける合計を求めるコードに、内部的にSUM関数を使っているはずである。ということは、丸めと切り捨てでSUMと同じ問題に直面するということである。念のため、読者の使う環境で軽くテストをしてもらいたい。

だが、AVG関数において本当にやっかいなのは、平均そのものの性質にある。つまり、これは集合の中央の傾向を測定する手段としては不完全で、非常にミスリーディングなのだ。特に問題なのが、データの偏りに弱いことだ。ダレル・ハフのロングセラー『統計でウソをつく法』に載っているサンプルを見てみよう。以下は、25 人の社員が働く企業における社員の給与額とその人数の表だ。

社員の人数	給与（ドル）	統計指標
12	2, 000	最頻値、最小値
1	3, 000	中央値
4	3, 700	
3	5, 000	
1	5, 700	平均
2	10, 000	
1	15, 000	
1	45, 000	最大値

給与の平均（正確には算術平均）は5700ドルである。管理職が組合にいい顔をしようとするときは、この数字を使いたいだろう。一方、組合が待遇改善を求める場合は、最頻値の2000ドルを使うだろう。これは最も出現回数の多い値だ。これが最小値と同じということは、大多数の社員は低賃金で搾取されていることを意味する。

このケースにおいて一番良い指標は中央値である。これは後ほど詳しく見ていくが、その値よりも給与の大きい社員と少ない社員の数が同じになる値であり、この場合は3000ドルだ。中央値を計算する際、母集合の要素の個数が偶数だと中央値が1 つに決まらない。この場合、真ん中の2つの値の平均を中央値とするか、2つの値のどちらかを中央値とする方法がある。前者は指標としてはより正確だが、現実にその値を持つ要素は存在しなくなる。後者は、現実に存

在する値ではあるが、上を取るか下を取るかで値が大きく異なる可能性がある（{0,0,100,100}という集合の場合、下を取れば0、上を取れば100だ）。

最頻値にも問題がある。これは、値の分散によっては1 つに決まらない可能性がある。非常に貧しい人々と同じ数だけの金持ちな人々が暮らす国を考えてほしい。この国の富の分布は二峰性分布を示すだろう。さらに所得階層が細かく分類できる国ならば、多峰性分布を示すはずだ。

いくつかの実装は、中央値や最頻値を求める関数を持っている。だがこれらは標準SQLの一部ではない。これらをピュアSQLで求める手段については、第31章で見る。

29.3.1　データがないグループを含む平均

ときとして、平均を計算するときに空集合のグループが含まれるケースがある。例を使って説明しよう。ある漁場の管理人が魚のサンプリング調査を行っている。魚の生息数をいくつかのグループごとに集計したい。ただし、魚の各サンプルは1つ以上のグループに分類される（「泥水を好む」とか「清流にしか住まない」とか）。管理人は、その漁場に「いない」魚の平均数を計算しようと苦労していた。これだけ聞くと「何を言ってるんだ？」と思うだろうが、実はそれほど妙な要求ではない。そして、それほど簡単な要求でもない。

管理人は次のようなテーブルを使って、魚に関するデータを収集していた。

```
CREATE TABLE Samples
(sample_id INTEGER NOT NULL,
 fish_name CHAR(20) NOT NULL,
 found_cnt INTEGER NOT NULL,
     PRIMARY KEY (sample_id, fish_name));
```

Samples：魚のサンプル調査テーブル　　sample_id：サンプルID　　fish_name：魚名　　found_cnt：発見数

同じサンプルIDの魚は、同じグループに分類される。ただし上述のように、1つのサンプルは、複数のグループに分類されることがある点に注意してほしい。

```
CREATE TABLE SampleGroups
(group_id INTEGER NOT NULL,
 sample_id INTEGER NOT NULL,
     PRIMARY KEY (group_id, sample_id));
```

SampleGroups：サンプルグループテーブル　　group_id：グループID　　sample_id：サンプルID

テーブルのデータは次のようになっている。

Samples

sample_id	fish_name	found_cnt
1	Seabass	14
1	Minnow	18
2	Seabass	19

SampleGroups

group_id	sample_id

1	1
1	2
2	2

管理人は、同じグループ内における魚の平均数をその種類ごとに求めようとしてた。たとえば、グループ1にはサンプル1と2の魚が含まれている。グループ1に含まれる'Minnow'（ヒメハヤ）の平均数は、:my_fish_name='Minnow'、:my_group=1というパラメータを使って求められそうだ。この場合、クエリは次のように書ける。

```
SELECT fish_name, AVG(found_cnt)
  FROM Samples
 WHERE sample_id IN (SELECT sample_id
                       FROM SampleGroups
                      WHERE group_id = :my_group)
   AND fish_name = :my_fish_name
 GROUP BY fish_name;
```

このクエリは'Minnow'の平均数として18を返すが、これは間違いだ。グループ1内のサンプル2には'Minnow'が1匹もいない。1回目のサンプリングでは18匹捕まえたが、2回目のサンプリングでは1匹も捕まえられなかったということだから、正しい平均数は((18＋0)/ 2)＝9である。

正答を得るためには、別のアプローチをとる必要がある。まずは、いくつかのステップに分割しよう。最初に、SELECT文を実行して必要なサンプルの数を求める。次に別のSELECT文で合計を計算し、最後に手で平均を計算するのだ。これらを1つのSELECT文で書く方法はあるだろうか。

明快な方法は、sample_idごとに存在しないfish_nameを見つけ出し、それらに対して数がゼロの行を新たに作ってやることである。こうすれば、最初のクエリが使えるようになる。存在しない行は、次のINSERT文で作れる。

```
INSERT INTO Samples
SELECT DISTINCT M1.sample_id, M2.fish_name, 0
  FROM Samples AS M1, Samples AS M2
 WHERE NOT EXISTS
        (SELECT *
           FROM Samples AS M3
          WHERE M1.sample_id = M3.sample_id
            AND M2.fish_name = M3.fish_name);
```

だが困ったことに、魚の種類は10 万以上、サンプルの数も1 万以上あることが判明した。先の方法では、管理人が持っているディスクの容量をオーバーしてしまう。1 つのSQL 文で求めるには、いくつかのトリックを使う必要がある。

```
SELECT fish_name,
       SUM(found_cnt)/
         (SELECT COUNT(sample_id)
            FROM SampleGroups
           WHERE group_id = :my_group) AS avg_smpl
  FROM Samples
 WHERE fish_name = :my_fish_name
 GROUP BY fish_name;
```

結果

```
 fish_name    avg_smpl
=====================
 Minnow              9
```

SELECT 句のスカラサブクエリには、「平均とは値の総計（被除数）を値の個数（除数）で割った数である」という、平均を求めるルールをそのまま適用している。これと同じことをやるもう1 つの方法は、外部結合を使ってすべてのグループIDを保存することだが、これをやると、サンプルグループに存在しない魚についてNULLが生じる。そのため、NULLを0に変換するといった対処が必要になる。

29.3.2　複数列を対象にした平均

複数列に対する集計は、COALESCE関数を使うと簡単にNULLを0にできて便利だ。

```
CREATE TABLE Foobar
(c1 INTEGER,
 c2 INTEGER,
 c3 INTEGER);

INSERT INTO Foobar
VALUES (0, 0, 0), (1, 0, 0), (NULL, NULL, NULL),
       (NULL, 1, 2), (NULL, NULL, 0), (4, 5, NULL);

SELECT (COALESCE(c1, 0.0)
      + COALESCE(c2, 0.0)
      + COALESCE(c3, 0.0)) AS c_total
  FROM Foobar;
```

同じように、複数列に対する最大と最小は、CASE式、またはGREATEST関数とLEAST関数を使うことでできた。平均の場合も考え方は同じだ。ただし、NULLを除外するのが少し難しい。まず、NULLの数をカウントするトリックは次のようなものだ。

```
SELECT (COALESCE(c1-c1, 1)
      + COALESCE(c2-c2, 1)
      + COALESCE(c3-c3, 1)) AS null_cnt
  FROM Foobar;
```

もし列がNULLなら、引き算の結果はNULLに、非NULLなら0になる。このクエリを組み込んで、NULLの行を計算から除外する解がこれだ。全列がNULLだった場合はNULLを返すようにしている。

```
SELECT CASE WHEN COALESCE(c1, c2, c3) IS NULL
            THEN NULL
            ELSE (COALESCE(c1, 0.0)
                + COALESCE(c2, 0.0)
                + COALESCE(c3, 0.0)) / (3 - (COALESCE(c1-c1, 1)
                                           + COALESCE(c2-c2, 1)
                                           + COALESCE(c3-c3, 1)))
        END AS horizontal_avg
  FROM Foobar;
```

29.4

極値関数

MINとMAX は、数学における極値関数として知られている。この関数は、集合の要素が順序を持っていることを前提している。そうでなければ、要素の値に基づいて最初とか最後の値を選択することはできない。SQL が持っているのは2 つの簡単な極値関数だが、これを一般化してn個の要素を返すようなクエリを書くことも可能だ。

29.4.1　簡単な極値関数

MAX([ALL | DISTINCT] <value expression>)は、<value expression>集合内の最大の要素を返す。この関数は、数値型だけでなく文字列型と時間型に対しても適用できる。空集合に対しては空集合を返す。技術的には、MAX(DISTINCT <value expression>) も書けるが、結果はMAX(ALL <value expression>)やMAX(<value expression>)とまったく同じなので、使う意味がない。

MIN([ALL | DISTINCT] <value expression>)は、<value expression>集合内の最小の要素を返す。この関数は、数値型だけでなく文字列型と時間型に対しても適用できる。空集合に対しては空集合を返す。技術的には、MIN(DISTINCT <value expression>)も書けるが、結果はMIN(ALL <value expression>)やMIN(<value expression>)とまったく同じなので、使う意味がない。

MAX 関数は数値型に使った場合は最大値を返す。時間型に使った場合は、'9999-12-31'（ISO 8601における最後の日付）に最も近

い値を返す。文字列型に使った場合は、昇順に並べて最後の値を返す（照合順序を知っておく必要がある）。

MIN 関数の場合も同様である。数値型に使った場合は最小値を、時間型に使った場合は、'9999-12-31'から最も遠い値を、文字列型に使った場合は、昇順に並べて最初の値を返す。

だが中には、この時間型に対するMAXやMINの動作に違和感を感じる人もいる。中には、MAX は現在時刻より前で、最も現在時刻に近い値を返すべきだと考える人もいるかもしれない。同様に、MIN関数は、現在時刻よりあとで、最も現在時刻に近い値を返すべきだということになる。人間の心理として、時間に関する推論は現在時刻を起点にしたくなるものらしい。

この問題を考えるために、次のような条件の例を考えよう。

```
billing_date < (CURRENT_DATE - INTERVAL '90' DAY)
```

ほとんどの人は、この条件の意味を「現在から90日以前」の請求日を探すものと理解するだろう。MINとMAXにも同じ考え方を適用すると、上のような解釈が生まれることになるのだろう。

極値関数を使うにあたり、もう1つ考えねばならないのが、列に使われる照合順序である。列は、照合順序で定義された順序でソートされるからだ。同じデータであっても、テーブルが異なれば異なる照合順序が割り当てられていることがあるため、最大値や最小値が異なる可能性がある。もっとも、このように異なる照合順序を混在させるのは良くないスキーマ設計なのだが。

極値関数をサブクエリの中で利用するときは、いくつかパフォーマンス向上のトリックがある。うまくいくかは実装によっても違うが、1つ紹介しよう。

いま、給与の履歴を持つテーブルがあるとする。ここから、現在の社員の給与水準を検索するクエリを考える。素直に考えればこうなるだろう。

```
SELECT emp_status, ...
  FROM SalaryHistory AS S0
 WHERE S0.change_date
         = (SELECT MAX(S1.change_date)
              FROM SalaryHistory AS S1
             WHERE S0.emp_id = S1.emp_id);
```

だが、このクエリは次のように書くこともできる。

```
SELECT emp_status, ...
  FROM SalaryHistory AS S0
 WHERE NOT EXISTS
        (SELECT *
           FROM SalaryHistory AS S1
          WHERE S0.emp_id = S1.emp_id
            AND S0.change_date < S1.change_date);
```

MAX関数を使った相関サブクエリは、まずサブクエリでワークテーブルを作り、それをemp_id でグルーピングする。そして各グループに対して最大値を探し、それを最終結果のために保存する。他方、NOT EXISTSバージョンは、条件を満たす最初の行を見つけた時点でTRUEを返すので、スキャン量が少ない。したがって、NOT EXISTSのほうが速く動くことが期待できる。

29.4.2　極値関数の一般化

これは上位n 個または下位n 個の値を求める問題として知られている。この問題を考えたのはジム・ワンコウスキーである。社員の給与を格納するテーブルがあるとする。ここから最も高い給与を得ている上位3人の社員を選択するクエリを考えてほしい。1人だけを見つけるならば「`SELECT MAX(salary_amt) FROM Personnel;`」という単純なSQL を書けばよい。だが、SQLはある列について複数の大きな値を返す関数を持っていない。このクエリの面倒なところは、いくつか仕様にあいまいなところがあるからである。たとえば次のような点だ。

1. 給与をどのように定義するのか。基本給のことか？ それとも歩合給も含むのか？ 簡単にするため、以後給与は各社員の`salary_amt`列を使うものとしよう。
2. この会社のテーブルに3人以下しか社員がいなかった場合はどうするのか？ 社員全員を表示すればよいのだろうか？ あるいは、`NULL` や空集合またはエラーメッセージを返すべきだろうか？ ちょうど誰も応募者がいなかったコンテストに合格者がいないように。
3. 同じ給与の社員が2人いた場合はどうするべきか？ 3行以上になるのを承知のうえで全員表示するべきか？ それともランダムに誰か1人を選んで残りを落とすべきだろうか？ あるいはこの場合も`NULL`や空集合またはエラーメッセージを表示すべきだろうか？

この問題を具体的に考えるために、次のテーブルを考えよう。Able、Bill、Charlesがこのテーブルにおいて最も高い給与を得ている3人だ。しかし、BillとCharlesが同額を得ているので純粋に金額上位3つを抜き出してみると1000ドル、900ドル、800ドルになる。この3つの金額に該当する社員はAble、Bill、Charlesに加え、Deltaも含まれる。

Personnel

emp_id	salary_amt
Able	1000
Bill	900
Charles	900
Delta	800
Eddy	700
Fred	700
George	700

SQL の初心者がこの問題を解こうとすると、まず`ORDER BY` 句で結果をソートして、カーソルでその結果を先頭から順に読もうとする。標準SQLではたしかにカーソルは`ORDER BY` 句を持っているが、特定の行数を返す機能はない（大半の製品は、そのような機能を実装依存の構文でサポートしているが）。だが、この方法はテーブル

全体をソートしてさらに全行を結果セットに含む必要があるため、テーブルが大きいと時間もかかるし、リソースも多く消費する。

　手続き型言語でこの問題を解く最良のアルゴリズムはC.A.R.ホーアによるパーティションアルゴリズムだ。これは、値の集合を3 つのパーティションに分割するクイックソートを使うものだ。実行時間の期待値はわずか2n回の操作である。もしファイルがすでにソート済みであれば、このトリックは1回の操作で答えにたどり着く。以下にPascalで書いたコードを示す。

```
CONST
 list_length = { some large number };
 ...
TYPE
 LIST = ARRAY [1..list_length] OF REAL;
 ...
PROCEDURE FindTopK (Kth INTEGER, records : LIST);
VAR pivot, left, right, start, finish: INTEGER;
BEGIN
start := 1;
finish := list_length;
WHILE start < finish
DO BEGIN
 pivot := records[Kth];
 left := start;
 right := finish;
 REPEAT
   WHILE (records[left] > pivot) DO left := left + 1;
   WHILE (records[right] < pivot) DO right := right - 1;
   IF (left >= right)
   THEN BEGIN { swap right and left elements }
     Swap (records[left], records[right]);
     left := left + 1;
     right := right - 1;
     END;
   UNTIL (left < right);
   IF (right < Kth) THEN start := left;
   IF (left > Kth) THEN finish := right;
   END;

{ 最初のｋ個は、１からｋ番目までの位置を示す。最も大きい数がｋ番目に位置することを除けば、
いかなる特定の順序も持たない。}
```

ワンコウスキーが最初にこの問題を提示したとき、他のいくつかの解法も見せてくれた。1つのやり方は、入れ子のサブクエリを使ったUNION 演算である。最初の結果テーブルでテーブル全体の最大値を求め、2番目の結果テーブルで最大値よりも小さい中で最大値を求め、.....（以下同じ）。このパターンはn個に拡張可能である。クエリは次のようになる。

```
SELECT MAX(salary_amt)
  FROM Personnel
UNION
SELECT MAX(salary_amt)
  FROM Personnel
 WHERE salary_amt < (SELECT MAX(salary_amt)
                       FROM Personnel)
UNION
SELECT MAX(salary_amt)
  FROM Personnel
 WHERE salary_amt < (SELECT MAX(salary_amt)
                       FROM Personnel
                      WHERE salary_amt
                           < (SELECT MAX(salary_amt)
                                FROM Personnel));
```

結果

```
 MAX(salary_amt)
=================
 1000
  900
  800
```

この解の欠点は、パフォーマンスが非常に悪いことだ。サブクエリのネストもUNIONも非常にコストが高い。UNION のたびに重複行を削除するためのソートが実行される。これはsalary_amt列が一意ではないため、やむをえない。

UNIONを使ったアプローチを改善するにはサブクエリを使って3番目に高い給与を見つけ、それ以上の給与をすべて選択することだ。この方法は同じ給与の社員が複数いた場合も対応できる。

```
SELECT DISTINCT salary_amt
  FROM Personnel
 WHERE salary_amt
        >= (SELECT MAX(salary_amt)
              FROM Personnel
             WHERE salary_amt
                    < (SELECT MAX(salary_amt)
                         FROM Personnel
                        WHERE salary_amt
                               < (SELECT MAX(salary_amt)
                                    FROM Personnel)));
```

だが今日ではこうした問題を解く一番良い方法は、ウィンドウ関数を使うことだ。サブクエリが不要だし、ソートもウィンドウ関数で1回しか実行されないため、上に挙げた解よりも高速に動く可能性が高い。だが、常にそうであるかは断言が難しいので、各自の環境でテストしてほしい。

```
SELECT DISTINCT X.salary_amt
  FROM (SELECT salary_amt,
               DENSE_RANK()
                 OVER (ORDER BY salary_amt DESC)
          FROM Personnel) AS X(salary_amt, sal_rank)
         WHERE X.sal_rank <= 3;
```

もう1つの別解は、相関名と単一行の結果テーブルを使うものだ。このパターンはより大きなグループに拡張するのが簡単だ。また、ORDER BY 句を使わなくともソートされた結果を得られる。欠点は、結果が1行で返されるため、その後の編集が不便になるかもしれ

ないことと、結果を他のクエリと結合するのが難しくなることである。

```
SELECT MAX(P1.salary_amt),
       MAX(P2.salary_amt),
       MAX(P3.salary_amt)
  FROM Personnel AS P1,
       Personnel AS P2,
       Personnel AS P3
 WHERE P1.salary_amt > P2.salary_amt
   AND P2.salary_amt > P3.salary_amt;
```

結果

```
 MAX(P1.salary_amt)  MAX(P2.salary_amt)  MAX(P3.salary_amt)
=============================================================
 1000                900                 800
```

　このサンプルは、社員が最低3人いれば、上位3人の給与を返す。そうでない場合は、結果が空になる。このときも結果を返すようにするには、外部結合を使おう

```
SELECT MAX(P1.salary_amt),
       MAX(P2.salary_amt),
       MAX(P3.salary_amt)
  FROM Personnel AS P1
         LEFT OUTER JOIN
           Personnel AS P2
             ON P1.salary_amt > P2.salary_amt
               LEFT OUTER JOIN
                 Personnel AS P3
                   ON P2.salary_amt > P3.salary_amt;
```

　このクエリは給与の値が2つ以下の場合はNULLを返す。

　以下に示すのは、同じ給与の社員を複数行で表示するバージョンだ。この解はピエール・ボーキンが考えたものだ。

```
SELECT P1.emp_id, P1.salary_amt
  FROM Personnel AS P1, Personnel AS P2
 WHERE P1.salary_amt >= P2.salary_amt
 GROUP BY P1.emp_id, P1.salary_amt
HAVING (SELECT COUNT(*) FROM Personnel) - COUNT(*) + 1 <= :n;
```

結果

```
 emp_id   salary_amt
======================
 Able            1000
 Bill             900
 Charles          900
```

もし新しい社員を900ドルで雇えば、その社員も結果に含まれる。だが、800ドル以下で雇った社員は結果には現れない。多くの場合においてこれが最も望ましい結果であろう。

同じ結果を得る異なったクエリを、あと2つ紹介しよう。

```
SELECT P1.emp_id, P1.salary_amt
  FROM Personnel AS P1, Personnel AS P2
 GROUP BY P1.emp_id, P1.salary_amt
HAVING COUNT(CASE WHEN P1.salary_amt < P2.salary_amt
                  THEN 1
                  ELSE NULL END) + 1 <= :n;
```

```
SELECT P1.emp_id, P1.salary_amt
  FROM Personnel AS P1
       LEFT OUTER JOIN
            Personnel AS P2
         ON P1.salary_amt < P2.salary_amt
 GROUP BY P1.emp_id, P1.salary_amt
HAVING COUNT(P2.salary_amt) + 1 <= :n;
```

この2つの解からもわかるように、この問題を解くのにサブクエリは不要である。

ところでこれは余談だが、大学の集合論の講義を覚えていれば、この問題がジョン・フォン・ノイマンによる入れ子集合を使った順序数の定義と同じであることに気づくであろう。集合論の定理からは自己結合に関する多くのアイデアが得られる。ジョン・フォン・ノイマンは20 世紀の最も偉大な数学者の1 人であり、プログラム内蔵式コンピュータとゲーム理論を生み出した。

この問題において、3という数字が、任意の数に拡張可能であるべきなのは明らかだ。1つの注意点は「P1.salary_amt<=P2.salary_amt」という条件は境界値を含むので、社員が3 人以下しかいない場合でも結果が得られることだ。もし、定数に達しない場合、結果を無効にしたいときは「P1.salary_amt<P2.salary_amt」という条件に変えればよい。このクエリを表現するもう1つの方法は、次のように書くことだ。

```
SELECT Elements.emp_id, Elements.salary_amt
  FROM Personnel AS Elements
 WHERE (SELECT COUNT(*)
          FROM Personnel AS Boundary
         WHERE Elements.salary_amt < Boundary.salary_amt) < 3;
```

ここでもやはり、スカラサブクエリにおけるCOUNT(*)と比較の条件を変えると、微妙に異なる結果を得るよう調節できる。読者の使う製品でどのバージョンが最速かは、実際に試してみてもらいたい。

もし社員数ではなく給与の金額で上位3 つを選びたければ、COUNT(*)の代わりにCOUNT(DISTINCT)を使えばよい。

```
SELECT Elements.emp_id, Elements.salary_amt
  FROM Personnel AS Elements, Personnel AS Boundary
 WHERE Elements.salary_amt <= Boundary.salary_amt
 GROUP BY Elements.emp_id, Elements.salary_amt
HAVING COUNT(DISTINCT Boundary.salary_amt) <= 3;
```

結果

```
emp_id    salary_amt
====================
Able             1000
Bill              900
Charles           900
Delta             800
```

これは社員の数ではなく、給与の数を数えている。そのため、複数の社員が同じ給与の場合、全員を結果に含めることになる。つまりそのような社員がいた場合、結果が3行以上になるということだ。

ケン・ヘンダーソンは、導出テーブルとスカラサブクエリを使った解を考えた。

```
SELECT P2.salary_amt
  FROM (SELECT (SELECT COUNT(DISTINCT P1.salary_amt)
                  FROM Personnel AS P1
                 WHERE P3.salary_amt <= P1.salary_amt) AS ranking,
               P3.salary_amt
          FROM Personnel AS P3) AS P2
 WHERE P2.ranking <= 3;
```

結果

```
  salary_amt
============
        1000
         900
         900
         800
```

このクエリにIN 述語を使うことで、他の集約関数を作ることもできる。たとえば、給与の履歴を記録するテーブルがあるとしよう。そこから、社員ごとに直近3 回の給与の平均額を求めたい。もう1 つ前提として、(まだ入社して日が浅いなどの理由で)給与の変更が3 回以下の社員については、テーブルにあるだけの行を使って平均を求めるとする。

```
SELECT S0.emp_id, AVG(S0.last_salary_amt)
  FROM SalaryHistory AS S0
 WHERE S0.change_date
          IN (SELECT P1.change_date
                FROM SalaryHistory AS P1,
                     SalaryHistory AS P2
               WHERE P1.change_date <= P2.change_date
               GROUP BY P1.change_date
              HAVING COUNT(*) <= 3)
 GROUP BY S0.emp_id;
```

29.4.3　複数の基準の極値関数

一般化した極値関数の基本的なアイデアは、データをソートすることだ。ということは、複数の列を使うよう一般化できるということである。これは、WHERE句の検索条件を変えることで実現できる。たとえば、バスケットボールチームから、身長と体重をソートキーにしてトップn位の選手を求めるには、次のように書く。

```
SELECT P1.emp_id
  FROM Personnel AS P1, Personnel AS P2
 WHERE P2.height >= P1.height     -- 第1ソートキー
    OR (P2.height = P1.height AND -- 第2ソートキー
        P2.weight >= P1.weight)
 GROUP BY P1.emp_id
HAVING COUNT(*) <= :n;
```

手続き型に馴れているプログラマのほうがこのコードを理解しやすいかもしれない。彼らはファイルシステムで、まさに同じように複数のフィールドを使ったソートを行うからだ。キーが増えるとWHERE句の条件が複雑になるので、入れ子の各レベルにおいて比較述語が正しく使われているかを確認するのが大事だ。条件の順番もまた大事である。同じ身長の中で体重順にするのと、同じ体重の中で身長順にするのとでは大違いだ。

行比較を使うと、このコードをもう少し簡略化できる。

```
SELECT P1.emp_id
  FROM Personnel AS P1, Personnel AS P2
 WHERE (P2.height, P2.weight) <= (P1.height, P1.weight)
 GROUP BY P1.emp_id
HAVING COUNT(*) <= 4;
```

ただし、この書き方の欠点は、昇順と降順を混在させるのが難しいことだ。

複数の基準による集約問題の別バージョンとして、グループ内での一般化極値関数がある。たとえば、部署別の上位3 つの給与を求めたい。これはサブクエリにグループを絞る条件を追加すれば答えが得られる。

```
SELECT dept_nbr, salary_amt
  FROM Personnel AS P1
 WHERE (SELECT COUNT(*)
          FROM Personnel AS P2
         WHERE P2.dept_nbr = P1.dept_nbr
           AND P2.salary_amt < P1.salary_amt) < :n;
```

あるいは次のように書いてもいい。

```
SELECT P2.dept_nbr, MIN(P1.salary_amt)
  FROM Personnel AS P1, Personnel AS P2
 WHERE P1.dept_nbr = P2.dept_nbr
   AND P1.salary_amt >= P2.salary_amt
 GROUP BY P2.dept_nbr, P2.salary_amt
HAVING COUNT(DISTINCT P1.salary_amt) <= 3;
```

29.5

LIST 関数

LIST([DISTINCT]<string expression>)は、もともとSybaseのSQL Anywhere（昔のWatcom SQL）の機能だった。これは、文字列型に対して使用できる唯一の集約関数だ。この関数は、複数の文字列を受け取り、NULLを除外してから、カンマでつなげた1つの長大な文字列を出力する。これは文字列結合の一般化であり、その点でSUM関数が加算の一般化なのと同じだ。DISTINCTオプションは、通常の結合と同じように、演算の前に重複値を除外する。

MySQL 4.1はこの関数を拡張してGROUP_CONCATという関数を実装した。機能は基本的に同じだが、ORDER BYとSEPARATORオプションを指定できる。これは特に、SQLを書くためのSQLを使うときに便利だ。たとえば簡単な例を挙げると、スキーマテーブルに対してこの関数を使えば、あるテーブルのすべての列を得て、SELECT *を列名に展開するのにそのリストを使うことができる。

29.5.1　再帰共通表式とLIST 関数

まず最初にしなければならないことは、リストに表示される項目をソートするためのシーケンス列を追加することである。これはROW_NUMBER関数を使えば動的にも可能だが、いまはすでにテーブルの列として存在すると仮定しよう。

```
CREATE TABLE Make_String
(list_seq INTEGER NOT NULL PRIMARY KEY,
 cat_string VARCHAR NOT NULL);
```

Make_String：文字列作成テーブル　　list_seq：リストシーケンス　　cat_string：結合する文字列

```
INSERT INTO Make_String
VALUES (1, 'abc'), (2, 'bcd'), (3, 'cde'),
       (4, 'def'), (5, 'efg'), (6, 'fgh'), (7, 'ghi');
```

ここで欲しい結果は次のようなものだ。

```
'abc, bcd, cde, def, efg, fgh, ghi'
```

これをSQLで行うには、再帰共通表式を使えばよい。

```
WITH RECURSIVE
String_Tail(list_seq, cat_string)
AS
(SELECT list_seq, cat_string
   FROM Make_String
  WHERE list_seq > 1),
String_Head(list, max_list_seq, list_seq)
AS
(SELECT cat_string, 1, list_seq
   FROM Make_String
UNION ALL
SELECT String_Head.list || ', ' || String_Tail.cat_string,
       String_Head.max_list_seq + 1,
       String_Head.list_seq
  FROM String_Tail, String_Head
 WHERE String_Tail.list_seq = String_Head.max_list_seq + 1)
SELECT list
  FROM String_Head
 WHERE list_seq = 1
   AND max_list_seq = (SELECT MAX(list_seq)
                         FROM Make_String);
```

このクエリの面白い副作用は、文字列がきちんと左から右へ結合されることだ。

29.5.2　クロステーブルとLIST 関数の併用

カール・フェダールは、これと同じような結果を得るのに次のような方法を用いた。衣服の商品を持つテーブルから、客が試着した商品の名前の組み合わせを1つの列に連結する。

```
CREATE TABLE Clothes
(id INTEGER NOT NULL,
 seq INTEGER NOT NULL,
      PRIMARY KEY (id, seq),
 clothing_item VARCHAR(10) NOT NULL,
 worn_flg CHAR(1) NOT NULL
  CHECK (worn_flg IN ('Y', 'N')));
```

Clothes：衣服テーブル　id：ID　seq：連番　clothing_item：商品　worn_flg：試着フラグ

```
INSERT INTO Clothes
VALUES (1, 1, 'Hat', 'Y'), (1, 2, 'Coat', 'N'), (1, 3, 'Glove', 'Y'),
       (2, 1, 'Hat', 'Y'), (2, 2, 'Coat', 'Y'),
       (3, 1, 'Shoes', 'N'),
       (4, 1, 'Pants', 'N'), (4, 2, 'Sock', 'Y');
```

```
CREATE TABLE Crosstabs
(seq_nbr INTEGER NOT NULL PRIMARY KEY,
 seq_nbr_1 INTEGER NOT NULL,
 seq_nbr_2 INTEGER NOT NULL,
 seq_nbr_3 INTEGER NOT NULL,
 seq_nbr_4 INTEGER NOT NULL,
 seq_nbr_5 INTEGER NOT NULL);
```

Crosstabs：クロステーブル　seq_nbr：番号　seq_nbr_1：番号1　seq_nbr_2：番号2　seq_nbr_3：番号3　seq_nbr_4：番号4　seq_nbr_5：番号5

```
INSERT INTO Crosstabs
VALUES (1, 1, 0, 0, 0, 0),
       (2, 0, 1, 0, 0, 0),
       (3, 0, 0, 1, 0, 0),
       (4, 0, 0, 0, 1, 0),
       (5, 0, 0, 0, 0, 1);
```

```
SELECT Clothes.id,
       TRIM (MAX(SUBSTRING(clothing_item FROM 1 FOR seq_nbr_1 * 10))
         || ' ' || MAX(SUBSTRING(clothing_item FROM 1 FOR seq_nbr_2 * 10))
         || ' ' || MAX(SUBSTRING(clothing_item FROM 1 FOR seq_nbr_3 * 10))
         || ' ' || MAX(SUBSTRING(clothing_item FROM 1 FOR seq_nbr_4 * 10))
         || ' ' || MAX(SUBSTRING(clothing_item FROM 1 FOR seq_nbr_5 * 10)))
       AS cat_item
  FROM Clothes, Crosstabs
 WHERE Clothes.seq = Crosstabs.seq_nbr
   AND Clothes.worn_flg = 'Y'
 GROUP BY Clothes.id;
```

結果

```
 id   cat_item
=================
  1   Hat  Glove
  2   Hat Coat
  4   Sock
```

29.6

PRD 関数

ボブ・マクガワンはある問題についてコンピュサーブに質問を投稿した。彼のクライアントは、ある投資会社で、過去の投資パフォーマンスを調べるため、次のようなテーブルを使っていた。

```
CREATE TABLE Performance
(portfolio_id CHAR(7) NOT NULL,
 execute_date DATE NOT NULL,
 rate_of_return DECIMAL(13,7) NOT NULL);
```

Performance：パフォーマンステーブル　portfolio_id：ポートフォリオID
execute_date：投資実行日　rate_of_return：利回り

```
INSERT INTO Performance
VALUES ('001', '2006-01-06', 0.5),
       ('001', '2006-01-07', 0.3),
       ('001', '2006-01-08', 0.6),
       ('001', '2006-01-09', 0.8),
       ('001', '2006-01-10', 0.1);
```

ある期間の利回りを計算するためには、次の式を使う。

```
  (1 + rate_of_return [day_1])
* (1 + rate_of_return [day_2])
* (1 + rate_of_return [day_3])
* (1 + rate_of_return [day_4])
...
* (1 + rate_of_return [day_N])
```

1つのポートフォリオについて、ある期間におけるリターンを1行で得るにはどのようなクエリを作ればいいだろうか？ マクガワン氏が望んでいたのは、ちょうどSUM関数が列方向の合計を得るように、列方向の積を得る集約関数である。もし読者が数学専攻だったな

ら、和を求める大文字のシグマ（Σ）と、積を求める大文字のパイ（∏）で表記する関数を思い出しただろう。もし後者の関数がSQLにあれば、次のような構文になるはずだ。

```
PRD ([DISTINCT] <expression>)
```

この関数でDISTINCT オプションに意味があるかどうかは疑問だが、この集約関数が使えれば、彼の問題は非常に簡単なSQLで解決するだろう。イメージとしてはこんな感じだ。

```
SELECT portfolio_id, PRD(1.00 + rate_of_return)
  FROM Performance
 WHERE execute_date BETWEEN start_date AND end_date
 GROUP BY portfolio_id;
```

29.6.1　式によるPRD 関数

残念ながらSQLに前項の最後で見たような関数はない。したがって、この関数を表現するには、2つ目のテーブルが必要になる。このテーブルは5日からなる期間を保持する。

```
CREATE TABLE BigPi
(execute_date DATE NOT NULL,
 day_1 INTEGER NOT NULL,
 day_2 INTEGER NOT NULL,
 day_3 INTEGER NOT NULL,
 day_4 INTEGER NOT NULL,
 day_5 INTEGER NOT NULL);
```

BigPi：大文字のパイテーブル　execute_date：投資実行日　day_1：日1
day_2：日2　day_3：日3　day_4：日4　day_5：日5

たとえば、1月6日から10日までの5日間について投資パフォーマンスを求めよう。テーブルに入れるデータは次のようなものだ。

```
INSERT INTO BigPi
VALUES ('2006-01-06', 1, 0, 0, 0, 0),
       ('2006-01-07', 0, 1, 0, 0, 0),
       ('2006-01-08', 0, 0, 1, 0, 0),
       ('2006-01-09', 0, 0, 0, 1, 0),
       ('2006-01-10', 0, 0, 0, 0, 1);
```

考え方は、`BigPi.execute_date`がn番目の日付と等しい1列だけにフラグ1を立て、残りの日付は0にする。すると、この問題を解くクエリはこうなる。

```
SELECT portfolio_id,
       (SUM((1.00 + P1.rate_of_return) * M1.day_1) *
        SUM((1.00 + P1.rate_of_return) * M1.day_2) *
        SUM((1.00 + P1.rate_of_return) * M1.day_3) *
        SUM((1.00 + P1.rate_of_return) * M1.day_4) *
        SUM((1.00 + P1.rate_of_return) * M1.day_5)) AS product
  FROM Performance AS P1, BigPi AS M1
 WHERE M1.execute_date = P1.execute_date
   AND P1.execute_date BETWEEN '2006-01-06' AND '2006-01-10'
 GROUP BY portfolio_id;
```

結果

```
 portfolio_id   product
=======================
 001            6.1776
```

もし範囲内の日付の`rate_of_return` が1 つでも不明の場合は、積は0 になる。これは必ずしも悪い結果ではないが、本当にパフォーマンスが0の場合と区別がつかない。失われたデータがあるときは`NULL`を返したいなら、各`SUM`関数の`CASE`式を次のように変える。

```
CASE WHEN SUM((1.00 + P1.rate_of_return) * M1.day_N) = 0.00
     THEN CAST (NULL AS DECIMAL(13, 7))
     ELSE SUM((1.00 + P1.rate_of_return) * M1.day_N)
 END
```

あるいは、次のように書いてもいい。

```
COALESCE (SUM((1.00 + P1.rate_of_return) * M1.day_N), 0.00)
```

29.6.2　対数によるPRD 関数

SQLグルの1 人であるロイ・ハーヴェイは、この問題に複数の別解を見つけた。もし読者が計算尺を使ったことのある年齢で、高校の数学を覚えていれば、対数の和によって積を計算できることを知っているだろう。この解のいいところは、SUM 関数の中でDISTINCTオプションを使えることだ。

だが、このアプローチには注意すべき点も多い。標準SQL では、自然対数はLN 関数で計算するが、底が10の対数を求めるには、LOG10関数を使う。一般化された形で示すと、LOG(<par ameter>, <base>)を使う。底の入力に0 以下が与えられた場合の対数は定義されていないため、標準では例外を返すよう定義されている。だが、古い製品の中には、0またはNULLを返すものもあるかもしれない。同様に、標準では指数をEXP 関数として定義している。対数と指数の列から積を求める式は次のようになる。

```
CREATE TABLE NumberTable
(nbr DECIMAL(13, 7));

INSERT INTO NumberTable
VALUES (2), (3), (4), (5), (6);

SELECT ((EXP (SUM (LN (CASE WHEN nbr = 0.00
                            THEN CAST (NULL AS FLOAT)
                            ELSE ABS(nbr) END))))
                       * (CASE WHEN MIN (ABS (nbr)) = 0.00
                               THEN 0.00
                               ELSE 1.00 END)
                           * (CASE WHEN MOD (SUM (CASE WHEN SIGN(nbr) = -1
                                                       THEN 1
                                                       ELSE 0 END), 2) = 1
                                   THEN - 1.00
                                   ELSE 1.00 END)) AS big_pi
  FROM NumberTable;
```

結果

```
 big_pi
=======
    720
```

この解の優れたところは、PRD (DISTINCT <expression>)と同値の結果をSUM (DISTINCT<expression>)オプションによって得られることだ。

なお、このクエリで使うデータ型には注意が必要だ。整数型の0と1 を使うと、CASE 式においてDECIMAL型の0.00と1.00が使われているので、不適切である。

3つのCASE式のうち、最初のものは、すべての0と負数を0 以上の数またはNULLに変換している。これは、SQL が例外を発生させたときのためである。2 番目のCASE 式は、選択された任意の行のnbr 列の中に0 があった場合には0 を返す。「MIN(ABS(nbr))=0.00」

は、与えられた数値のリストの中に0 があるかどうかを調べる便利なトリックだ。3 番目のCASE 式は、nbr 列の中に負の奇数があった場合に–1を返す。一番内側のCASE 式はSIGN 関数を使っている。この関数は負数ならば–1、正数ならば＋1、0 ならば0 を返す。SUM 関数がこの結果を合計して、MOD関数で偶数か奇数かを判断している。

最初にロジックを考えるときは上のクエリのようになるが、もっと少ない計算で同じことをできる。

```
SELECT CASE MIN (SIGN (nbr))
       WHEN 1 THEN EXP (SUM (LN (nbr))) -- すべて正数
       WHEN 0 THEN 0.00 -- 1つ以上の0を含む
       WHEN -1 -- 1つ以上の負数を含む
       THEN (EXP (SUM (LN (ABS(nbr)))) *
         (CASE WHEN MOD (SUM (ABS (SIGN(nbr)-1/ 2)), 2) = 1
               THEN -1.00 ELSE 1.00 END))
               ELSE CAST (NULL AS FLOAT) END AS big_pi
  FROM NumberTable;
```

考え方は、3つの特別なケースを想定することだ。

1. すべて正数
2. 1つ以上の0を含む
3. 1つ以上の負数を含む

入力値がどのケースに相当するかは、集合の要素の最小値にSIGN 関数を使えばすぐにわかる。なお、負数を含む場合はさらに2つのサブケースに分類される。

3.1 負数が偶数の場合

3.2 負数が奇数の場合

それから簡単な計算をすれば最終結果の符号が決まる。もし、計算対象の列に正数しか入っていないことが確実なら、単純に次のように書くことができる。

```
PRD(<exp>) = EXP(SUM(LN (<exp>)))
```

余談だが、『Bypasses: A Simple Approach to Complexity』（Melzak, 1983）は、共役性についての一般的な原理を解説したコンパクトな本だ。

共役性は、ここでは対数と指数を変換する際に利用しているが、計算の複雑さを減らすために式を変換したり、反転させたりするためにも使われる。

29.7

ビット単位の集約関数

いまから紹介する方法は、推奨できる方法ではない。というのも、個々の構成要素が独自の意味を持つベクトルで1つの列を上書きしてしまうことによって、第1正規形を破壊することになるからだ。だが頻繁に見るやり方なので、ここで言及しておきたい。下手なプログラマは、データモデルにおける個々の属性に列を与える代わりに、INTEGERやSMALLINTといったバイナリ表現における1つ1つのビットに意味を与えようとする。

こういうやり方をすると、1または2の補数で計算を行うハードウェアでは大きな問題を引き起こす。また、NULLの扱い方も非常に難しい。ユーザーからデータ型の物理モデルを隠蔽していない製品では、実装依存のビット演算子であるBIT_AND やBIT_ORという演算子を持っている【訳注2】。だが大半の製品はビット単位の集約関数は持っていない。

集約関数の一般的な動作ルールをもう一度おさらいしておくと、次のようになる。

ルール1　すべてのNULLを除外する。

ルール2　パラメータリストにDISTINCTがあれば、すべての重複行を除外する。

ルール3　パラメータリストにALLがあるか、何の指定もなければ、すべての重複行を残す。

ルール4　残った値に対して要求された計算を行う。

ルール5　空集合またはすべてNULLの集合（ルール1より、これは空集合になる）に対してはNULLを返す。

ルール2とルール3はビット演算子には関係ないことに注意してほしい。というのも(a BIT_OR a)＝aであり、(a BIT_AND a)＝aだからだ。つまり、重複値は一切結果に影響しないのである。

29.7.1　ビット単位のOR 集約関数

整数型のビット列を持つ簡単なテーブルを作ろう。CHECK制約は、負数を除外するとともに、異なる長さのビット文字列を防止している。

```
CREATE Table Foobar
(bits INTEGER -- テスト用なのでNULLを許可する
     CHECK(bits BETWEEN 0 AND 15));
```

Foobar：サンプルテーブル　　bits：ビット文字列

```
INSERT INTO Foobar VALUES (1), (2), (3), (4);
```

ここで欲しいのは、bits列に対するビット単位のORである。bit列のそれぞれの整数をビットに直してOR演算を行うと、次のようになる。

整数	ビット表現
1	001
2	010

3	011
4	100
…	…
7	111

```
SELECT  MAX (CASE WHEN MOD (bits/1, 2) = 1
                   AND bits IS NOT NULL
                  THEN 1 ELSE 0 END)
      + MAX (CASE WHEN MOD (bits/2, 2) = 1
                   AND bits IS NOT NULL
                  THEN 2 ELSE 0 END)
      + MAX (CASE WHEN MOD (bits/4, 2) = 1
                   AND bits IS NOT NULL
                  THEN 4 ELSE 0 END)
      + MAX (CASE WHEN MOD (bits/8, 2) = 1
                   AND bits IS NOT NULL
                 THEN 8 ELSE 0 END) AS bitwise_or
  FROM Foobar;
```

結果

```
 bitwise_or
============
          7
```

`bits/1`は演算の必要はないと思うかもしれないが、他の除算と同じパターンであることを示すためにこう表記している。希望は、賢いオプティマイザなら`MAX`関数内部の`CASE`を利用することだ。この`CASE` 式は、取り得る結果の集合が制限されている（この場合は{0, 2^n}）。このため、いったん任意の行が可能な最大値を返せば、すべて計算しなくてもそこで評価をやめることができる。この式の問題点は、`bits`列が全行`NULL`だと0を返すことだ。これは

「SIGN(MAX (bits)) * (<bitwise OR expression>)」とすることで回避できる。なぜなら、もしFoobarテーブルが空か、またはbits 列がすべてNULL だったなら、SIGN 関数はMAX(bits)に対してNULL を返し、NULLが伝播するからだ。もしbits列にNOT NULL制約が付与されていれば、この可能性は除外できる。

29.7.2　ビット単位のAND 集約関数

以上の議論を踏まえると、次のビット単位のAND演算のロジックも明らかだ。MAX関数はここではMIN関数になる。なぜなら、1つでも0があれば、集約結果におけるビットを0にセットできるからだ。SIGN関数を使ったトリックも同様に利用できる。

```
SELECT SIGN(MAX(bits)) *
         MIN (CASE WHEN MOD (bits/1, 2) = 1
                    AND bits IS NOT NULL
                   THEN 1 ELSE 0 END)
       + MIN (CASE WHEN MOD (bits/2, 2) = 1
                    AND bits IS NOT NULL
                   THEN 2 ELSE 0 END)
       + MIN (CASE WHEN MOD (bits/4, 2) = 1
                    AND bits IS NOT NULL
                   THEN 4 ELSE 0 END)
       + MIN (CASE WHEN MOD (bits/8, 2) = 1
                    AND bits IS NOT NULL
                   THEN 8 ELSE 0 END) AS bitwise_and
  FROM Foobar;
```

結果

```
 bitwise_or
============
          0
```

【訳注1】　最近のDBMSのオプティマイザならば一度のスキャンで済ませるだろう。

【訳注2】　この演算子はたとえばPostgreSQL、MySQLでは動作する。

30

高度な集約、ウィンドウ関数、OLAP

Advanced Grouping, Windowed Aggregation and OLAP in SQL

ほとんどのSQL プログラマはOLTP まわりの仕事をしており、あまりOLAP やDWHといったデータを集計しレポートを作成することに特化したシステムの経験はないと思う。こうしたシステムでは、スキーマデザインと典型的な処理パターンは、通常のOLTPのクエリとは大きく異なる。簡単に言うと、読者がOLTPの常識と思っていることは、DWHでは完全にひっくり返る。

違い1

OLTPでは、データは短く頻繁なトランザクションによって更新される。DWHでは、大量の静的なデータを定期的にロードする。そのあとは、ほとんど変更を受けない。

違い2

OLTP データベースは、現在の作業に必要なだけのデータしか保持しない。DWH では、持ちうる限りの履歴データを保持する。Teradata 社は2008 年時点において、ペタバイト以上のDWHを運用する顧客が少なくとも5社あると公表している。eBay（5ペタバイト）、ウォルマート（2.5ペタバイト）、バンク・オブ・アメリカ（1.5ペタバイト）、Dell（1ペタバイト）、および名前は

非公開の銀行である。ペタバイトは、2^50＝1, 125, 899, 906, 842, 624 バイト＝1, 024テラバイトである。

違い3

OLTPのクエリは、シンプルなものが多い。DWHのクエリは集約を行うため複雑になる。たとえば、OLTPクエリが「ジョー・セルコの買ったチョコレートの値段は？」と訊くのに対して、DWH のクエリは「チョコレートの購買と地理および服装の相関関係は？」と訊く。

違い4

OLTPは可能な限り速いことが求められる。DWHは計算の精度に関心があり、複雑なクエリを実行するためなら少々時間がかかってもよいと考える。

違い5

適切に設計されたOLTPデータベースは正規化されている。通常、DWHはスタースキーマまたはスノーフレークスキーマを採用しており、これはかなり非正規化されている。スタースキーマという言葉を考え出したのはラルフ・キンボールである。この概念の詳細については、同氏の書籍や論文を参照してほしい。

30.1

スタースキーマ

スタースキーマは、端的に言って基本的な正規化ルールに対して違反している。スタースキーマは、巨大なセントラル・ファクトテーブルを持つ。ファクトテーブルには、たとえば売上といったレポート出力の対象となるあらゆる事実が格納される。OLTPでは、在庫で一テーブル、売上で1テーブル、顧客で1テーブル、といった調子で複数のファクトテーブルを持っていた（それが正規化というものだ）。DWHでは、すべてが巨大な1つのテーブルに放り込まれる。

ファクトテーブルの値に対するディメンションは、比較的小さなテーブル群に保持され、レポーティングに必要なスケールの単位を提供する。たとえば、売上ファクトに対する時間ディメンションの集約階層としては、年、月、週といったレベルが考えられる。あるいは、重量ディメンションならば、パウンド、キログラム、あるいは箱の大きさといったレベルがあるだろうし、カテゴリディメンションであれば、商品を部署ごとに分類するだろう。

こうしたディメンションを用意しておくことで、好きな単位の粒度でファクトを集約したり、また特定のディメンションを変更することも可能になる。標準SQLにOLAP機能が追加されたのは最近のことなので、昔はOLAPをやるには、Microsoft社のMDXのようなOLAPに特化した言語を使わなければならなかった【訳注1】。これがOLAP の敷居を高くしていた理由の1 つだ。OLAP の本質は、ベンダーのデモで見るようなウィザードやGUI の中にあるのではない。そうした見栄えのする外見は、多くの場合、事情をよく知らない素人を惑わすためのエサなのだ。

いまでは、OLAPの多くの要素がリレーショナルデータベースのエンジン自体に統合されている。この技術統合によって、RDBMS とOLAP の違いはあいまいになってきている。従来、RDBMSはOLAPにおいて、ディメンションのデータを保持するだけの受動的な役割しか持っていなかったが、いまやOLAPのエンジンそのものになりつつある。RDBMSは、今後ますます伝統的なリレーショナルデータと、OLAP向けのデータの両方のニーズを満たすようになるだろう。それによって、OLAP専用の技術に対する投資を減らすことができ、RDBMSの技術、スキル、リソースに対する投資からの収益は増えていくだろう。ただし、これはあくまで集計エンジンとしての話であって、SQLができることと、レポーティングツールや統計パッケージソフトの仕事を混同してはならない。

30.2

GROUPING 演算子

SQL-99では、GROUP BY 句にROLLUPとCUBEオプションの拡張が行われた。これによって作られるグループは、しばしば超集合（super-group）と呼ばれる。これは、古いSQL でも、GROUP BYとUNIONを使って書くことができるが、かなりSQLが大きくなり面倒だ。

ご想像のとおり、NULL は従来のGROUP BY と同様、1 つのグループを形成する。ただし、GROUPING(<column reference>)関数を使うことで、もともと集約キーに存在していたNULL なのか、CUBE、ROLLUP、GROUPING SETSによって生成された超集合行のNULLなのかを識別することが可能だ。この関数は、後者のNULLに対しては1、それ以外のNULLには0を返す。

SQL:2003は、NULLのチェックを行う複数列バージョンを追加した。その再帰的定義は以下のとおりである。

```
GROUPING (<column ref 1>, ..., <column ref n-1>, <columnref n>)
```

これは以下と同値だ。

```
2 *GROUPING(<column ref 1>, ...<column ref n-1>) + GROUPING(<column ref n>))
```

30.2.1 GROUP BY GROUPING SETS

GROUPING SET(<column list>)はUNION で集約クエリをつなげる書き方の簡略形である。これはレポート作成でよく利用する。

たとえば、部署別の合計と職種別の合計をつなげて出力するには、従来はUNIONを使ってこう書いていた。

```
CREATE TABLE Personnel
(dept_name CHAR(8),
 job_title CHAR(16));

INSERT INTO Personnel
VALUES ('001', 'Sales'), ('001', 'Development'), ('001', 'Research'),
       ('002', 'Sales'), ('002', 'Development'),
       ('003', 'Sales');

SELECT dept_name, NULL AS job_title,
       COUNT(*)
  FROM Personnel
 GROUP BY dept_name
UNION ALL
SELECT NULL AS dept_name, job_title,
       COUNT(*)
  FROM Personnel
 GROUP BY job_title;
```

結果

```
DEPT_NAME  JOB_TITLE     COUNT(*)
==================================
001                           3
002                           2
003                           1
           Development        2
           Sales              3
           Research           1
```

いまでは、もっと簡単に書ける。

```
SELECT dept_name, job_title, COUNT(*)
  FROM Personnel
 GROUP BY GROUPING SETS (dept_name, job_title);
```

OLAPのグルーピング関数には共通の問題がある。小計レベルのディメンションにNULLを生成するのだ。この生成されたNULLと本物の

NULLを見分ける手段が、先述のGROUPING関数である。これを使ってみよう。

```
SELECT CASE GROUPING(dept_name)
         WHEN 1 THEN 'department total'
         ELSE dept_name END AS dept_name,
       CASE GROUPING(job_title)
         WHEN 1 THEN 'job total'
         ELSE job_title END AS job_title,
       COUNT(*)
  FROM Personnel
 GROUP BY GROUPING SETS (dept_name, job_title);
```

結果

```
DEPT_NAME         JOB_TITLE    COUNT(*)
=========================================
002               job total           2
003               job total           1
001               job total           3
department total  Development         2
department total  Sales               3
department total  Research            1
```

他のグルーピング関数はすべて、GROUPING SETS関数を使って定義することができる。

30.2.2 ROLLUP

ROLLUPは古典的な合計と小計のレポート形式をSQLで表現したもので、「通常の」集約された行に加えて小計の行も結果に追加する。小計行は、超集合行であり、すべての階層について計算を行う。ROLLUPはGROUPING SETSを次のように繰り返すことで定義される。

```
GROUP BY ROLLUP (a, b, c) = GROUP BY GROUPING SETS (a, b, c)
                                                   (a, b)
                                                   (a)
                                                   ()
```

n 個の要素を持つROLLUPは、n＋1 個のGROUPING SETSに変換できる。もう1 つ覚えておいてほしいのは、ROLLUPにおいて入力に取るリストの順序が非常に重要なことだ。これは次に見るCUBEとの大きな違いである。

30.2.3 CUBE

CUBEは、ROLLUPで生成された超集合行に加え、いわゆる「クロス集計表」の行を生成する。これらもまた超集合行だ。これらは名前のとおり、スプレッドシートにおける行列をクロスさせた合計のことだ。CUBEもやはりGROUPING SETSで表すことができる。CUBEの場合、リストのすべての順列について小計を求めるので、n 個の要素を持つCUBE は2^n 個のGROUPING SETSに展開される。

```
GROUP BY CUBE (a, b, c) =  GROUP BY GROUPING SETS (a, b, c)
                                                  (a, b)
                                                  (a, c)
                                                  (b, c)
                                                  (a)
                                                  (b)
                                                  (c)
                                                  ()
```

この例では、3 個のCUBE の要素が8 個のGROUPING SETS に展開された。CUBE はROLLUP と違って、リストの順序はおかまいなしに可能なすべての組み合わせの小計を求める。したがって、CUBE (a, b) の結果は、CUBE (b, a) と同じだ。その点で、CUBE の結果は同

じリストに対するROLLUPの結果を包含するので、CUBEはROLLUPの拡張だとも言える。

30.2.4　SQL によるOLAP の例

次に、高度なOLAP関数を伝統的なSQLと組み合わせて使用する例を示す。レストランの売上げテーブルから、地域と都市別の売上げを求めるものだ。

```
SELECT B.region_nbr, S.city_id, SUM(S.sales_amt) AS total_sales
  FROM SalesFacts AS S, MarketLookup AS M
 WHERE EXTRACT (YEAR FROM trans_date) = 2011
   AND S.city_id = B.city_id
   AND B.region_nbr = 6
 GROUP BY ROLLUP(B.region_nbr, S.city_id);
```

レストラン集合は、「地域＝6」と「年＝1999」という条件から明示的に制限される。結果イメージは次のようになる。これは、2 つのグループ(region_nbr, city_id)に対してROLLUP を行い、以下の3つのレベルの集計結果を返す。

- 合計
- 地域別の小計
- 地域別都市別の小計

結果イメージ

```
region_nbr   city_id   total_sales
=================================
6            1         81655   ⬅ 地域別都市別の小計
6            2         131512
6            3         58384
...          ...       ...
6            30        1733
6            31        5058
6            NULL      1190902   ⬅ 地域＝6の小計
NULL         NULL      1190902   ⬅ 合計
```

30.3

ウィンドウ句

ウィンドウ句は非公式にOVER句とも呼ばれる。考え方としては、まずテーブルをPARTITION BY句でパーティションに分割し、パーティション内の行をORDER BY句でソートする。次に、ウィンドウ関数が呼び出されたカレント行に仮想的なカーソルがセットされ、カレント行の前後数行からなる部分集合を作る。その際、もし<window frame exclusion>オプションが指定されていれば、特定の行を部分集合から除外する。最後に、その部分集合が集約関数または順序関数に渡され、スカラ値を出力する。ウィンドウ関数は、れっきとした関数であり、通常の関数が従うルールに従う。ただし、その構文は他の関数とは大きく異なっている。ウィンドウの部分は、<window name>または<window specification>で指定される。<window specification>は、OVER句の中でウィンドウの詳細を規定する。

ウィンドウは、使われた構文の違いに影響されることなく、同じように動作する。BNFは次のとおりだ。

```
<window function>::= <window function type> OVER <window name or specification>
<window function type>::=
<rank function type> | ROW_NUMBER ()| <aggregate function>
<rank function type>::= RANK() | DENSE_RANK() | PERCENT_RANK() | CUME_DIST()
<window name or specification>::= <window name> | <in-line window specification>
<in-line window specification>::= <window specification>
```

ウィンドウ句は3 つの下位の句を持っている。それが、パーティショニング、順序づけ、ウィンドウフレームである。以下1つずつ説明しよう。

30.3.1 PARTITION BY 句

PARTITION BY 句では、列名の集合を使ってパーティションを定義する。これは、先行するFROM 句、WHERE 句、GROUP BY 句、HAVING 句によって作られた行集合に対して作用する。もし何のパーティション定義も指定されない場合は、行集合全体が1 つのパーティションとみなされ、一度に行全体に対して集約関数が適用される。PARTITION BYはGROUP BYと似ているが、同じではない。GROUP BYは行を1つのパーティションに割り当て、1つのパーティションを1 行に集約する。一方、PARTITION BYは単に行をパーティションに分割するだけで、集約は行わない。したがって、入力と出力の行数は同じである。

30.3.2 ORDER BY 句

ウィンドウ句における順序づけは、カーソルにおけるORDER BY 句と似ている。ソートキーを指定し、昇順か降順かをオプションで指定できる。カーソルにおけるORDER BYとの違いは、各パーティション内部でソートされることだ。他の句はいずれもオプション扱いだが、ORDER BY は指定しないと意味をなさない【訳注2】。

```
<sort specification list>::= <sort specification> [{,<sort specification>}...]
<sort specification>::= <sort key> [<ordering specification>] [<null ordering>]
<sort key>::= <value expression>
<ordering specification>::= ASC | DESC
<null ordering>::= NULLS FIRST | NULLS LAST
```

なお、ORDER BY 句の仕様は、最近ではより一般的なものに変更されている。そのルールは以下のとおりだ。

ルール1

ソートキーに単純なキーだけでなく、式を指定できる。ただし、結果セットのソート順序を確認するためにも、列名だけを指定するほうがよい。

ルール2

<sort specification>は、指定されたキーに対して昇順・降順を指定する。DESCが指定されていない場合は、デフォルトで昇順（ASC）とみなされる。

ルール3

<null ordering>が指定されていない場合は、暗黙に実装依存の<null ordering>が使用される。これは古い標準SQLにおいては、ベンダーごとにNULLの扱いが異なるため大問題だった。なお、NULL同士は互いに等しいとみなされる。

ルール4

<sort specification>で順序に差異がない2つの値は、同値とみなされる。この同値なペア内部での順序は実装依存となる。

30.3.3　ウィンドウフレーム句

最もトリッキーなオプションがウィンドウフレームである。以下にBNFを示すが、これを理解するにはコードを見てみる必要がある。

```
<window frame clause>::= <window frame units> <window frame extent>
  [<window frame exclusion>]
<window frame units>::= ROWS | RANGE
```

RANGEは数値、日付、期間のいずれかのデータ型の単一のソートキーとあわせて使う。これは、行数ではきっちり指定できないあいまいなデータを値の範囲で指定するものだ。ROWSが指定されていれば、ソートキーは整数で指定される。

```
<window frame extent>::= <window frame start> | <window frame between>
<window frame start>::= UNBOUNDED PRECEDING | <window frame preceding> | CURRENT ROW
<window frame preceding>::= <unsigned value specification> PRECEDING
```

もしウィンドウがUNBOUNDED PRECEDINGで開始された場合、下限は必ずパーティションの最初の行になる。CURRENT ROWは読んで字のごとくである。<window frame preceding>は、実際の先行する行数である。

```
<window frame bound>::= <window frame start> | UNBOUNDED FOLLOWING
  | <window frame following>
<window frame following>::= <unsigned value specification> FOLLOWING
```

もしウィンドウがUNBOUNDED FOLLOWINGで開始された場合、下限は必ずパーティションの最後の行になる。CURRENT ROWは上と同じだ。<window frame following>は、実際の先行する行数である。

```
<window frame between>::= BETWEEN <window frame bound 1> AND <window frame bound 2>
<window frame bound 1>::= <window frame bound>
<window frame bound 2>::= <window frame bound>
```

カレント行とそのウィンドウフレームは、パーティションの範囲内に収まらなければならない。PRECEDINGとFOLLOWINGの限界は、実質的にフレームの終端に合わせるよう調整される。

```
<window frame exclusion>::= EXCLUDE CURRENT ROW | EXCLUDE GROUP | EXCLUDE TIES
  | EXCLUDE NO OTHERS
```

<window frame exclusion>はあまり使われないし、サポートしている実装も少ない。これは特定の条件に合致する行をウィンドウから削除する。

1. EXCLUDE CURRENTROWは、ウィンドウからカレント行を削除する。
2. EXCLUDE GROUPは、カレント行とカレント行と同じ値を持つ行を削除する。
3. EXCLUDE TIESは、カレント行とカレント行と同じ値を持つ行以外の行を削除する。
4. EXCLUDE NOOTHERSは、いかなる行も削除されないことを保証する。

30.4

ウィンドウ集約関数

通常の集約関数もウィンドウ句を持つことができる。

```
<aggregate function> OVER([PARTITION BY <column list>] [ORDER BY <sort column list>]
  [<window frame>])

<aggregate function>::=
     MIN([DISTINCT | ALL] <value exp>) | MAX([DISTINCT | ALL] <value exp>) |
     SUM([DISTINCT | ALL] <value exp>) | AVG([DISTINCT | ALL] <value exp>) |
     COUNT([DISTINCT | ALL] [<value exp> | *])
```

構文に特に違和感はないだろう。生成されたウィンドウは、ちょうど集約関数が適用されるときのグループのような扱いを受けるのだ。違うのは、この場合の集約関数はウィンドウ関数として機能するため、集約は行われないことだ。

30.5

順序関数

順序関数はウィンドウ句を使用するが、その中でORDER BYを使わないと意味をなさない。この関数は、パーティションまたはウィンドウフレーム内における行を相対的に順序づける。この関数は何の入力パラメータも取らない。

30.5.1 ROW_NUMBER

ROW_NUMBER関数は、ウィンドウ内において行を一意に識別する連番を割り当てる。その順序はORDER BY 句で定義される。最初の行は1から始まり、ウィンドウ内の最後の行まで1ずつ加算していく。もしORDER BY がない場合は、適当な順序が割り当てられるが、ORDER BY がないとエラーになる実装もある。

30.5.2 RANK 関数とDENSE_RANK 関数

RANK関数は、ウィンドウ内の行に連番の順位を割り振る。同位タイの行に対しては同じ順位を割り振るが、その際、タイの下の順位には欠番が生じる。たとえば、3位が2行ある場合は、1、2、3、3、5……となる。

DENSE_RANKも基本的には同じだが、同位タイの下の順位にも欠番を生じさせない。3 位が2行ある場合でも、1、2、3、3、4……となる。

30.5.3 PERCENT_RANK 関数とCUME_DIST 関数

これらはSQL:2003で追加された。PERCENT_RANKは百分率を使った相対順位を0～1の範囲で返す。CUME_DIST は累積分布という意味で、PERCENT_RANKと似ているが、こちらはパーセンタイル値をやはり0～1の範囲で返す。

どちらの関数も、定義はすでに見てきたウィンドウ関数を使って行うことができる。いま、<approximate numeric type>を実装依存の精度を持つ近似的な数値型としよう。PERCENT_RANK() OVER <window specification>は次と同値である。

```
CASE
WHEN COUNT(*)
   OVER(<window specification>
     RANGE BETWEEN UNBOUNDED PRECEDING
                AND UNBOUNDED FOLLOWING) = 1
THEN CAST (0 AS <approximate numeric type>)
ELSE (CAST (RANK ()
              OVER(<window specification>) AS <approximate numeric type>) - 1)
           / (COUNT (*)
                OVER(<window specification>
                     RANGE BETWEEN UNBOUNDED PRECEDING
                               AND UNBOUNDED FOLLOWING) - 1)
END
```

同様に、累積分布は実装依存の精度で近似的な数値型で定義される。CUME_DIST() OVER <window specification>は次と同値である。

```
(CAST (COUNT (*)
         OVER(<window specification>
              RANGE UNBOUNDED PRECEDING) AS <approximate numeric type>)
        / COUNT(*)
            OVER(<window specification>1
                 RANGE BETWEEN UNBOUNDED PRECEDING
                           AND UNBOUNDED FOLLOWING))
```

実は、他のウィンドウ関数も、互いを使って定義することができる。だがこれはちょっと面白いというだけで、特に実用上のメリットはない。たとえば、RANK() OVER <window specification>は次と同値である。

```
(COUNT (*) OVER(<window specification> RANGE UNBOUNDED PRECEDING)
  -COUNT (*) OVER(<window specification> RANGE CURRENT ROW) + 1)
```

DENSE_RANK() OVER(<window specification>)は次と同値である。

```
COUNT (DISTINCT ROW (<value exp 1>, ...,<value exp n>))
   OVER(<window specification> RANGE UNBOUNDED PRECEDING)
```

ここで<value exp i>はテーブルのソートキーを表している。

ROW_NUMBER() OVER (<window specification>)は次と同値である。

```
COUNT(*) OVER(<window specification> ROWS UNBOUNDED PRECEDING)
```

30.5.4　ウィンドウ関数の使用例

<aggregation grouping>は、集約関数が計算を行うパーティション内の行集合を定義する。たとえば、次の例では、各月ごとにカレントの月と直近2か月を1つの集合として集約関数の計算対象とする。これはいわゆる移動平均や累計を求めるクエリだ。

```
CREATE TABLE SalesHistory
(region_nbr  INTEGER,
 sales_month INTEGER,
 sales_amt   INTEGER,
     PRIMARY KEY(region_nbr, sales_month));
```

SalesHistory：売上げ履歴テーブル　region_nbr：地域番号　sales_month：売上げ月　sales_amt：売上げ額

```
region_nbr  sales_month  sales_amt
==================================
1                     1        100
1                     2        200
1                     3        500
1                     4        100
2                     1        700
2                     2        800
2                     3        900
2                     4       1000
```

```
SELECT SH.region_nbr, SH.sales_month, SH.sales_amt,
       AVG(SH.sales_amt)
          OVER(PARTITION BY SH.region_nbr
                   ORDER BY SH.sales_month ASC
                    ROWS 2 PRECEDING) AS moving_average,
       SUM(SH.sales_amt)
          OVER(PARTITION BY SH.region_nbr
                   ORDER BY SH.sales_month ASC
                    ROWS BETWEEN UNBOUNDED PRECEDING
                             AND CURRENT ROW) AS moving_total
  FROM SalesHistory AS SH;
```

結果

```
region_nbr  sales_month  sales_amt  moving_average  moving_total
============================================================
         1            1        100  100.00           100
         1            2        200  150.00           300
         1            3        500  266.67           800
         1            4        100  266.67           900
         2            1        700  700.00           700
         2            2        800  750.00          1500
         2            3        900  800.00          2400
         2            4       1000  900.00          3400
```

`AVG(SH.sales_amt) OVER(PARTITION BY...)`において、`OVER`句の中の構築子は、`AVG`関数が適用されるデータのウィンドウを定義している。ウィンドウ句はパーティションに分割された行集合を定義する。この集合が集約関数の計算対象となる。この例におけるウィンドウ句は、次のような一連の演算を行っている。

1. `SalesHistory`テーブルを地域ごとに分割する。
2. 各地域内のデータを月でソートする。
3. 同じ地域内の先行する2行をカレント行と同じ集合にグルーピングする。
4. 各グループに対して平均を計算する。

データベースエンジンが実際に計算を行うときは、必ずしもこの順序に従ってステップを踏む必要はない。だが、結果はこのステップを踏んだ場合と同じでなければならない。クエリ内の2番目のウィンドウ関数は、地域内の売上げについて、月順の累計を求めている。計算ステップは移動平均と同じだ。

集約関数の計算対象となるグループには2 種類ある。物理的グループと論理的グループである。前者は、カレント行の前後の行を行数

によって指定してグルーピングする。上のクエリは、AVGもSUMもROWSを使っているので物理的グルーピングである。一方、論理的グルーピングでは、行数ではなくデータの値に基づいた期間や範囲でグルーピング対象の行を決める。たとえば、カレント行にプラスして前2か月のデータを含むと定義するようなケースだ。

物理的グループは、データが連続的で歯抜けがないとわかっている場合には問題ない。また、値に基づく計算が不要なので論理的グループよりも多様なデータに対して適用できる。論理的グループは、データに歯抜けがあったり不規則なパターンを示す場合でも使えるのが便利だ。ただし、データの値に算術演算が適用できなければならないので、対象が数値や日付に限られる。

ウィンドウ関数の結果に名前を付けて、それをさらに計算したり加工するという使い方もある。ウィンドウ関数の結果を導出テーブルに含めて、外側のクエリで参照すれば簡単にできる。また、この導出テーブルをビューにするという使い方も当然考えられる。

```
SELECT X.*
  FROM (SELECT <window function 1> AS W1,
               <window function 2> AS W2,
               ...
               <window function n> AS Wn
          FROM ...
        [WHERE ...] )AS X
 [WHERE...]
 [GROUP BY ...];
```

30.6

ベンダー拡張

ベンダーは独自のウィンドウ関数を実装している。ベンダーの今後の拡張方針を占うことはできないが、ここでは2 種類のよくある拡張タイプについて解説しよう。プログラミングの練習として、これらの関数を標準SQLと読者の使っている製品の方言で書き換えてみてほしい。

30.6.1 LEAD 関数とLAG 関数

LEAD関数とLAG関数は、OracleやPostgreSQLなどで利用できる非標準の機能だ。これらは、集約値を求めるのではなく、カレント行の前後に移動して、その値を式の中で利用するためのものだ。この関数は3つの入力値とOVER句を使用する。一般的な構文は以下のとおりである。

```
[LEAD | LAG] (<expr>, <offset>, <default>) OVER(<window specification>)
```

1つ目の入力値<expr>は、前後の行を計算するための式である。

2つ目の入力値<offset>は、カレント行から相対的にどれだけの距離を前後に移動するか指定する。これは正の整数である必要がある。デフォルトは1である。

3つ目の入力値<default>は、<offset>がパーティションの範囲外を指定した場合の戻り値である。以下にこの関数のサンプルを示す。

```
CREATE TABLE Personnel
(dept_nbr  INTEGER,
 emp_id    INTEGER,
 sal_amt   INTEGER,
     PRIMARY KEY(emp_id));
```

Personnel：社員テーブル　dept_nbr：部署番号　emp_id：社員ID　sal_amt：給与額

```
dept_nbr  emp_id  sal_amt
========================
      10    7839     5000
      10    7782     2450
      10    7934     1300
      20    7788     3000
      20    7902     3000
      20    7566     2975
      20    7876     1100
      20    7369      800
```

```
SELECT dept_nbr, emp_id, sal_amt,
       LEAD(sal_amt, 1, 0)
          OVER(PARTITION BY dept_nbr
                   ORDER BY sal_amt DESC NULLS LAST)AS lead_sal_amt,
       LAG (sal_amt, 1, 0)
          OVER(PARTITION BY dept_nbr
                   ORDER BY sal_amt DESC NULLS LAST) AS lag_sal_amt
  FROM Personnel;
```

結果

```
dept_nbr  emp_id  sal_amt  lead_sal_amt  lag_sal_amt
=================================================
10        7839       5000          2450            0
10        7782       2450          1300         5000
10        7934       1300             0         2450
20        7788       3000          3000            0
20        7902       3000          2975         3000
20        7566       2975          1100         3000
20        7876       1100           800         2975
20        7369       800               0         1100
```

社員7782に注目してほしい。この社員の現在の給与は2450ドルである。給与に着目すると、同じ部署内でこの社員より高く最も近い給与は5000ドル、この社員より高く最も近い給与は1300ドルである。次に、社員7934 を見よう。この社員の現在の給与は1300ドルである。これは部署内の最低賃金だ。したがって彼よりも低い給与はデフォルトの0が出力される。

30.6.2 FIRST_VALUE 関数とLAST_VALUE 関数

FIRST_VALUE関数とLAST_VALUE関数は、非標準だか多くの実装で何らかの形でサポートされている。この2 つは、データを集約するというより、パーティション内でデータをソートし、最初または最後の行を返すための関数だ。

結果の行に含まれる式は、普通はソートキーとは何の関係もない。これは、イギリス海軍の提督が軍艦をアルファベット順に整列させたというジョークを彷彿とさせる。一般的な構文は次のとおりである。

```
[FIRST_VALUE | LAST_VALUE](<expr>) OVER(<window specification>)
```

次のサンプルを使えば、次のようになる。

```
CREATE TABLE Personnel
(emp_id    INTEGER,
 dept_nbr  INTEGER,
 hire_date DATE,
     PRIMARY KEY(emp_id));
```

Personnel：社員テーブル　emp_id：社員ID　dept_nbr：部署番号　hire_date：入社日

```
emp_id  dept_nbr  hire_date
============================
  7369        20  2011-01-01
  7566        20  2011-01-02
  7902        20  2011-01-02
  7788        20  2011-01-04
  7876        20  2011-01-07
  7499        30  2011-01-27
  7521        30  2011-01-09
  7844        30  2011-01-17
  7654        30  2011-01-18
  7900        30  2011-01-20
```

```
SELECT emp_id, dept_nbr, hire_date,
       FIRST_VALUE(hire_date)
          OVER(PARTITION BY dept_nbr
                   ORDER BY emp_id) AS first_hire_by_dept
  FROM Personnel;
```

結果は、部署別で最も小さい社員番号を持つ社員の入社日を返す。ここで、社員番号と入社日はまったく論理的関係がないことに注意してほしい。

emp_id	dept_nbr	hire_date	first_hire_by_dept	
7369	20	2011-01-01	2011-01-01	◀ 部署20の最初のemp_id
7566	20	2011-01-02	2011-01-01	
7902	20	2011-01-02	2011-01-01	
7788	20	2011-01-04	2011-01-01	
7876	20	2011-01-07	2011-01-01	◀ 部署20の最後のemp_id
7499	30	2011-01-27	2011-01-27	◀ 部署30の最初のemp_id
7521	30	2011-01-09	2011-01-27	
7844	30	2011-01-17	2011-01-27	
7654	30	2011-01-18	2011-01-27	
7900	30	2011-01-20	2011-01-27	◀ 部署30の最後のemp_id

FIRST_VALUE関数、LAST_VALUE関数と同じことを標準SQLで行うには、少し複雑なクエリを書く必要がある。FIRST_VALUE関数を標準SQLで表現するクエリは以下のようになる。

```
WITH FirstLastQuery
AS
(SELECT emp_id, dept_nbr, hire_date,
        ROW_NUMBER()
          OVER(PARTITION BY dept_nbr
                  ORDER BY emp_id ASC) AS asc_order
   FROM Personnel)
SELECT P.emp_id, P.dept_nbr, P.hire_date, FQ.hire_date
  FROM Personnel AS P, FirstLastQuery AS FQ
 WHERE P.dept_nbr = FQ.dept_nbr
   AND FQ.asc_order = 1;
```

LAST_VALUE関数もORDER BYを降順に変えるだけで作ることができる。

30.7

ウィンドウ関数小史

1999 年初頭、IBMとOracleは共同でウィンドウ関数を標準SQLに取り入れる拡張を提案した。そしてANSI のいつにない（というのは褒め言葉だ）素早い対応によって、これらはSQL-99 に取り入れられた。IBM はこの仕様の一部をDB2 UDB 6.2 に実装した。これは1999年中頃にはある程度商用利用が可能になった。1999後半にリリースされたOracle 8i R2とDB2 UDB 7.1は、どちらもウィンドウ関数のサポートをより強化した。

ウィンドウ関数の発展には、他のベンダーの貢献もあった。BI ツールを開発しているBrio、MicroStrategy、Cognos、それにデータベースベンダーのInformixだ。ハミッド・ピラヘシュをリーダーとするIBMアルマデン研究所は特に重要な役割を果たした。ピラヘシュのチームは、このテーマについて1 年ほど研究したのち、SQL のOLAP 機能の拡張アプローチをまとめ、Oracleに連絡した。すると、ちょうどOracleでも独立に同じテーマについての重要な仕事がなされていたことがわかった。Oracleで重要な役割を果たしたのは、アンディ・ウィコウスキだった。そこで両社は2か月かけて共同で標準への提案を作り上げた。

初めてOLAP 機能を実装した製品はRed Brickである。これは標準化より前のことだ。だが完全とは言いがたかった。こうしたOLAP機能の経緯については、以下のANSIのドキュメントから知ることができる。

「Introduction to OLAP functions」Fred Zemke, Krishna Kulkarni, Andy Witkowski, Bob Lyle著, 1999（ISO/IEC JTC1/SC32 WG3:YGJ-

068, ANSI NCITS H2-99-154r2）

【訳注1】　MDX（Multi Dimensional Expressions）とは、名前のとおり多次元データに対する問い合わせ言語のこと。MOLAPにおいては、事実上の標準言語として使われている。

【訳注2】　OracleやPostgreSQLなどでは、ウィンドウ関数で`ORDER BY` 句を指定しないと構文エラーになる。DB2にはこのような制限はないが、本文でも触れられているとおり、`ORDER BY` 句を指定しないとウィンドウ関数を使う意味はまずない。

31

SQLにおける記述統計

Descriptive Statistics in SQL

SQLは統計解析用のプログラミング言語ではないが、簡単な記述統計を行うためのトリックがいくつかある。またベンダーの多くも標準以外の統計機能を用意している。以下では、どのDBMSでも利用できる汎用的なコードに限定して示すが、これらを使うよりも製品固有の機能を使うほうが便利なこともあるだろう。ビルトイン関数のほうが速度も速いので、移植性の要件が重要でない場合は、そうした関数を使うほうがよい。

最もよく使われるベンダー拡張の関数は、中央値、最頻値、標準偏差、分位数、分散あたりだろう。より詳細な統計分析を行う必要がある場合は、データをSASやSPSSといった統計ソフトに投入する。だが、たいていの記述統計はSQLでもカバーできる。

31.1

最頻値

最頻値は、集合の中で最も登場回数の多い値だ。最頻値が2 つある集合もあり、そのような分布を二峰性分布と呼ぶ。3つあれば、三峰性分布である。

ほとんどのSQL実装は最頻値を求める関数は持っていない。というのは、これを計算するのは簡単だからだ。最頻値を計算するSQLは何種類かあるが、次に示すのはシェパード・トウィンドが考えたもので、複数の最頻値があるケースにも対応している。

```
CREATE TABLE Personnel
(emp_id     INTEGER PRIMARY KEY,
 salary_amt INTEGER );
```

Personnel：社員テーブル　　emp_id：社員ID　　salary_amt：給与

```
INSERT INTO Personnel
VALUES (1, 300), (2, 400), (3, 500),
       (4, 300), (5, 800), (6, 500),
       (7, 300), (8, 400), (9, 900);

SELECT salary_amt, COUNT(*) AS salary_mode
  FROM Personnel
 GROUP BY salary_amt
HAVING COUNT(*)
         >= ALL (SELECT COUNT(*)
                   FROM Personnel
                  GROUP BY salary_amt);
```

結果

```
  salary_amt  salary_mode
=========================
         300            3
```

ウィンドウ関数を使って書くとこうなる。

```
WITH
Freq (salary_amt, salary_amt_cnt)
AS
(SELECT salary_amt,
        COUNT(*) OVER(PARTITION BY salary_amt)
   FROM Personnel)
SELECT F1. salary_amt
  FROM Freq AS F1
 WHERE F1.salary_amt_cnt
         = (SELECT MAX (salary_amt_cnt)
              FROM Freq AS F2);
```

結果

```
  salary_amt
=============
         300
         300
         300
```

最頻値は、記述統計としては脆弱だ。というのも、ほんの少量のデータを追加するだけで値が変わってしまうからだ。たとえば、`part_color`という変数に、赤が100, 000 個、青が99, 999 個あった場合、最頻値は赤である。だが、青がもう2つ追加されただけで、最頻値は青に変わってしまう。この弱点を補うアイデアの1つは、変数の中である程度の分散kを許すことだ。一般にk を計算する最良の方法は、値の出現率を使うことだ。だがもちろん、適切なk の値は状況によって変わるだろう。

```
SELECT AVG(salary_amt) AS mode
  FROM Personnel
 GROUP BY salary_amt
HAVING COUNT(*)
        >= ALL (SELECT COUNT(*) * 0.95 -- パーセンテージは任意の値
                  FROM Personnel
                 GROUP BY salary_amt);
```

31.2

AVG 関数

数学でいう単純平均は、SQLではAVG関数として組み込まれている。この関数の問題の1 つは、SQLはなるべくデータ型を保存しようとするため、入力値xが整数型だと、結果も整数型になる可能性があることだ。また、たとえ結果が小数になったとしても、製品のデフォルトのデータ型が望ましいものになる保証はない。この問題を避けるには、`AVG(1.0 * x)`と書くか、次のように明示的な型変換を行うことだ。

```
AVG(CAST (x AS FLOAT)) または AVG(CAST (x AS DECIMAL (s, p)))
```

このあたりの動作は実装依存なので、製品の仕様を確認してほしい。

経験の浅いプログラマは、集約関数が計算の前にNULLを除外することを忘れがちだ。これはつまり、`(SUM(x)/COUNT(*))`と`AVG(x)`は同値ではないということだ。`(x * 1.0)/COUNT(*)`と`AVG(COALESCE(x * 1.0, 0.0))`の結果に、NULL がどのような違いを与えるか考えてほしい（29.3節も参照）。

31.3

中央値

中央値（median）は、上にも下にもちょうど同数の値が存在するような値として定義される。もしデータ集合の中にそういう値が見つかれば、その値は統計的中央値と呼ばれることもある。見つからなければ、普通は真ん中の2 つの数の平均をとる。こちらは会計的中央値とも呼ばれる。通常、中央値というとこの会計的中央値を指すので、以後もこちらの定義に従おう。サンプルとして、デイトの有名な部品テーブルを利用しよう(Date, 1983, 1995a)。これは、部品の重さなどの属性を管理するテーブルだ。

```
CREATE TABLE Parts
(part_nbr   CHAR(2) PRIMARY KEY,
 part_name  CHAR(8),
 part_color CHAR(8),
 part_wgt   INTEGER,
 city_name  CHAR(8));
```

Parts：部品テーブル　part_nbr：部品番号　part_name：部品名　part_color：部品の色　part_wgt：部品の重さ　city_name：生産地

Parts

part_nbr	part_name	part_color	part_wgt	city_name
p1	Nut	Red	12	London
p2	Bolt	Green	17	Paris
p3	Cam	Blue	12	Paris
p4	Screw	Red	14	London
p5	Cam	Blue	12	Paris

p6	Cog	Red	19	London

このテーブルから重さの中央値を見つけることを考えよう。手続き型の考え方をするなら、まずテーブルを重さでソートし、下位3つの行を見つける。この3行の重さは{12, 12, 12}となり、この集合の中で一番重い値もやはり12だ。これが下位中央値とか左中央値と呼ばれる値だ。他方、上位3 つの集合は{14, 17, 19}となり、この中で一番軽いのは14 である。これが上位中央値または右中央値と呼ばれる。両者の平均は、（12＋14） / 2＝13である。もしテーブルの行数が奇数であれば、ソートしたあと一発で中央値が見つかる。

中央値は平均よりも中央の傾向を測る指標としては優れているが、データをソートしないと計算が難しいのが難点だ。これは手続き型言語と比較した場合のSQLの欠点であり、中央値の計算を独自拡張の関数で行うベンダーが多いのも、これが理由の1つかもしれない【訳注1】。標準偏差と分布もまたよく使われる指標であり、かつこれらは計算にソートが不要という利点があるのだが、商業的な用途にあまり向かない。

31.3.1　プログラミング問題としての中央値

1990 年代前半、2 つの有名なデータベース専門雑誌があった。『DBMS』と『Database Programming & Design』である。私も両方の雑誌で連載コラムを持っていたことがある。クリス・デイトもたびたび寄稿していた。やがて2 つの雑誌は同じ出版社に買収され、『Intelligent Enterprise』という雑誌に統合されたのだが、当時、デイトと私は、DBMSやSQLのトピックについて「コラボ」企画をやっ

たことがある。そのトピックの1つが、当時の標準SQLの機能を使って中央値を求めるクエリを書くというもので、これは多くの読者の興味をかきたてた。

デイトは、2 つの解法を提示した(Date, 1992a; Celko and Date, 1993)。彼の最初の解は、テーブルのすべての行をコピーして増やしたとしても、中央値は変わらないという性質を利用している。全行をコピーすれば、テーブルの行数は常に偶数になるのがポイントだ。これによって奇数の場合を考慮しなくてよくなる。

```
WITH Double(part_wgt)
AS
(SELECT part_wgt FROM Parts
  UNION ALL
  SELECT part_wgt FROM Parts),
Middle_Weights(part_wgt)
AS
(SELECT part_wgt
   FROM Double
  WHERE (SELECT COUNT(*) FROM Parts)
      <= (SELECT COUNT(*)
         FROM Double AS T1
        WHERE T1.part_wgt >= Double.part_wgt)
   AND (SELECT COUNT(*) FROM Parts)
      <= (SELECT COUNT(*)
         FROM Double AS T2
         WHERE T2.part_wgt <= Double.part_wgt))
SELECT AVG(DISTINCT part_wgt) AS median
 FROM Middle_Weights;
```

結果

```
  median
========
      13
```

AVG関数の中でDISTINCTを使っているのは重要なポイントだ。これがないと、中央値ではなく単純な平均を返してしまう。具体的

に、次のような集合の重さ列を例に考えてみよう。

```
(12, 12, 12, 14, 17, 19)
```

2倍したTemp1テーブルは次のようになる。

```
(12, 12, 12, 12, 12, 12, 14, 14, 17, 17, 19, 19)
```

Temp2は上位集合の下限と、下位集合の上限を求めているが、重複行があるため単純に(12, 14)ではなく(12, 12, 12, 12, 12, 12, 14, 14)になる。このため、単純に平均をとると100/8＝12.5になってしまうのだ。正しくは26/2＝13である。

さて、この解の問題点は明らかだ。行数をわざわざ2倍にするのは時間とストレージの無駄遣いである。しかも、デイトは「リレーショナル原理主義者」として世に聞こえた人物であり、冗長な重複行の存在には当然ながら否定的だった。その彼が、一時的とはいえ重複行の存在を認める解を考えたのは面白いことだ。

デイトは、これ以外にも行をコピーせずに中央値を求める解を考えた（Date 1995b）。またデイトのほかに数人がいくつかのバリエーションを思いついたが、いまとなってはかなり古いやり方なので、ここではもう少し新しい考えを紹介しよう。

31.3.2　特性関数を使った中央値

アナトリー・アブラモヴィッチ、イェリーナ・アレクサンドロヴァ、オイゲン・ビルガーは、『SQL Forum』に、中央値を求める方法に関する一連の記事を寄稿した（SQL Forum, 1993, 1994）。彼ら

は、彼らがデルタと呼ぶSIGN関数を使った特性関数を定義した。デルタ、あるいは特性関数は、入力にブール式を取り、その式がTRUEならば1を、FALSEまたはUNKNOWNの場合は0を返す。この関数は、CASE式を使えば簡単に作れる。

彼らはまた、統計的中央値と会計的中央値を区別した。前者は、中央値は集合の要素でなくてはならないが、後者は集合の2 つの要素の平均値でもよい。統計的中央値は、集合の要素数が奇数である場合にのみ存在する。偶数の場合は、下位集合の最大値（左中央値または下位中央値）と上位集合の最小値（右中央値または上位中央値）のどちらを使うかを決めなければならない。

一意な列（bin）に対する統計的な左中央値は、次のクエリで見つけられる。

```
SELECT P1.bin
  FROM Parts AS P1, Parts AS P2
 GROUP BY P1.bin
HAVING SUM(CASE WHEN (P2.bin <= P1.bin) THEN 1 ELSE 0 END)
        = (COUNT(*) / 2.0);
```

CASE 式の中の不等号の向きを変えれば、統計的な右中央値を得られる。重複値を含む列に対する統計的中央値は、次のクエリで見つけられる。基本的な考え方は上のクエリと同じだが、HAVING句を少し調整して、上位集合と下位集合が重なりを持つようにしている。

```
SELECT P1.part_wgt
  FROM Parts AS P1, Parts AS P2
 GROUP BY P1.part_wgt
HAVING SUM(CASE WHEN P2.part_wgt <= P1.part_wgt
                THEN 1 ELSE 0 END)
         >= (COUNT(*) / 2.0)
   AND SUM(CASE WHEN P2.part_wgt >= P1.part_wgt
                THEN 1 ELSE 0 END)
         >= (COUNT(*) / 2.0);
```

結果

```
 part_wgt
==========
       14
       12
```

この結果は、右中央値と左中央値のペアである。さらにこの2 つを平均すれば、会計的中央値が得られる。これは導出テーブルを使えば簡単だ。

```
SELECT AVG(DISTINCT 1.0 * part_wgt)
  FROM (SELECT P1.part_wgt
          FROM Parts AS P1, Parts AS P2
         GROUP BY P1.part_wgt
        HAVING SUM(CASE WHEN P2.part_wgt <= P1.part_wgt
                        THEN 1 ELSE 0 END)
                 >= (COUNT(*) / 2.0)
           AND SUM(CASE WHEN P2.part_wgt >= P1.part_wgt
                        THEN 1 ELSE 0 END)
                 >= (COUNT(*)/2.0)) AS TMP;
```

結果

```
   avg
======
    13
```

このクエリでは、計算方法のバリエーションをいくつか選択できる。上のクエリでは、AVG 関数にDISTINCT が付いているので、左中央値と右中央値は1 つずつしか使わないが、AVG(DISTINCT part_wgt)をAVG(part_wgt)に変えるだけで、左中央値と右中央値が複数ある場合に、数の多いほうへの加重平均を取れる。たとえば、Partsテーブルが(13, 13, 13, 14)という状態だとしよう。純粋な統計的中央値は、(13＋14)/2.0＝13.5である。だが、重み付け中央値は(13＋13＋13＋14)/4.0＝13.25である。これはより中央の傾向をよく代表している。

CASE式を使った会計的中央値を求める別解は次のとおりだ。

```
SELECT CASE MOD(COUNT(*), 2)
       WHEN 0 -- テーブルが偶数の場合
       THEN (P1.part_wgt + MIN(CASE WHEN P2.part_wgt > P1.part_wgt
                                    THEN P2.part_wgt
                                    ELSE NULL END))/2.0
       ELSE MAX(P2.part_wgt) END AS median -- テーブルが奇数の場合
  FROM Parts AS P1, Parts AS P2
 GROUP BY P1.part_wgt
HAVING COUNT(CASE WHEN P1.part_wgt >= P2.part_wgt
                  THEN 1 ELSE NULL END)
         = (COUNT(*) + 1) / 2;
```

この解は故ケン・ヘンダーソンによるものだ。

導出テーブルを使わない別解は次のとおりだ。

```
SELECT CASE WHEN MOD (COUNT(*), 2) = 0
             AND SUM(CASE WHEN P2.part_wgt > P1.part_wgt
                          THEN 1 ELSE 0 END)
                    = (COUNT(*) / 2)
            THEN (P1.part_wgt + MIN(CASE WHEN P2.part_wgt > P1.part_wgt
                                         THEN P2.part_wgt
                                         ELSE NULL END)) / 2.0
            ELSE P1.part_wgt END AS median
  FROM Parts AS P1, Parts AS P2
 GROUP BY P1.part_wgt
HAVING SUM(CASE WHEN P2.part_wgt <= P1.part_wgt
                THEN 1 ELSE 0 END)
           >= ((COUNT(*) + 1) / 2)
   AND SUM(CASE WHEN P2.part_wgt >= P1.part_wgt
                THEN 1 ELSE 0 END)
           >= (COUNT(*)/2 + 1);
```

これはマイケル・シーハンが考えた。彼自身、これはあまりきれいなコードではないと感じていたが。

31.3.3 セルコの中央値

中央値を求める他の考え方を見てみよう。ソートされた値の列を見て、中央値がどこに位置するかを考えるのだ。part_wgtの値により、テーブルを次の3つのパーティションに分割する。

- part_wgtより小さい
- part_wgtと等しい
- part_wgtより大きい

これら3 つの集合の要素数を、それぞれlesser、equal、greaterという変数で表す。値を3 つのパーティションに分類するには、サブクエリを使えばできる。そうすると次の問題は、このパーティションを使うと中央値はどう定義されるか、ということだ。中央値の定義より、(lesser = greater)となることは明らかだ。

もしgreaterがテーブルの行数の半分よりも大きければ、そのpart_wgtは中央値ではあり得ない。同様に、lesser がテーブルの行数の半分よりも大きい場合も、そのpart_wgtは中央値ではありえない。

もし(lesser + equal) = greater であれば、part_wgt は左中央値である。同様に、(greater + equal) = lesserなら、part_wgtは右中央値である。だが、part_wgtが純粋な中央値である場合は、lesserとgreaterはともに、テーブルの行数の半分よりも小さくなければならない。これをSQLに翻訳すると、次のようなコードになる。

```
SELECT AVG(DISTINCT part_wgt)
  FROM (SELECT P1.part_nbr, P1.part_wgt,
               SUM(CASE WHEN P2.part_wgt < P1.part_wgt
                        THEN 1 ELSE 0 END),
               SUM(CASE WHEN P2.part_wgt = P1.part_wgt
                        THEN 1 ELSE 0 END),
               SUM(CASE WHEN P2.part_wgt > P1.part_wgt
                        THEN 1 ELSE 0 END)
          FROM Parts AS P1, Parts AS P2
         GROUP BY P1.part_nbr, P1.part_wgt)
                  AS Partitions (part_nbr, part_wgt, lesser, equal, greater)
 WHERE lesser = greater
    OR (     lesser <= (SELECT COUNT(*) FROM Parts)/2.0
        AND greater <= (SELECT COUNT(*) FROM Parts)/2.0);
```

FROM句の導出テーブルをサブクエリ式に展開していない理由は、このテーブルは四分位のような別のパーティションを作るためにも使えるからである。SELECT句のAVG(DISTINCT part_wgt)は、AVG(part_wgt)で置き換えることもできる。これは先ほど見た重み付け中央値だ。

リチャード・レムレーは、このコードをシンプルにしてくれた。彼の考えた道筋はこうだ。WHERE句を見ると、ここでやっていることは、集約関数を含んだサブクエリに対する簡単な計算でしかない。ということは、これはHAVING句に移すことが可能だ。集約関数の演算を行うなら、そのほうがパフォーマンスが良い。だがその前に、WHERE句の計算を少し変形する必要がある。

各行に対するlesser、equal、greaterの値は、すでに導出テーブルで計算済みだ。そして、この3 つの和は必ずテーブル全体の行数と同じになるのだから、「lesser <= (SELECT COUNT(*) FROM Parts)/2.0」という式は、結局次の式と同値になる。

```
lesser <= (lesser + equal + greater)/2.0
```

右辺の分母である2.0を左辺に持ってくる。

```
2.0 * lesser <= lesser + equal + greater
```

さらに、右辺のlesserを左辺に持ってくる。

```
2.0 * lesser - lesser <= equal + greater
```

すると最終的には、次のようになる。

```
lesser <= equal + greater
```

これをクエリに反映してみよう。

```
SELECT AVG(DISTINCT part_wgt)
  FROM (SELECT P1.part_nbr, P1.part_wgt,
               SUM(CASE WHEN P2.part_wgt < P1.part_wgt
                        THEN 1 ELSE 0 END),
               SUM(CASE WHEN P2.part_wgt = P1.part_wgt
                        THEN 1 ELSE 0 END),
               SUM(CASE WHEN P2.part_wgt > P1.part_wgt
                        THEN 1 ELSE 0 END)
          FROM Parts AS P1, Parts AS P2
         GROUP BY P1.part_nbr, P1.part_wgt)
                  AS Partitions (part_nbr, part_wgt, lesser, equal, greater)
WHERE lesser = greater
   OR (    lesser <= equal + greater
       AND greater <= equal + lesser);
```

だが、WHERE句の変形はこれで終わりではない。

```
WHERE lesser = greater
   OR (    equal >= lesser - greater
       AND equal >= greater - lesser)
```

これは以下のようになる。

```
WHERE lesser = greater
   OR equal >= ABS(lesser - greater)
```

まだ終わりではない。もし最初の条件「`lesser = greater`」が真の場合、2 番目の条件も必ず真になる(なぜなら端的に「`equal >= 0`」になるからだ)。したがって、最初の条件はまったく冗長であり、削除してかまわない。すると最終的な条件はこうだ。

```
WHERE equal >= ABS(lesser - greater)
```

これで式変形は終わりだ。次にこの条件判定を、WHERE句からHAVING句に移そう。これによって、導出テーブルに残す列は1つだけでよくなる。それゆえ、パフォーマンスも向上する。

```
SELECT AVG(DISTINCT part_wgt)
  FROM (SELECT P1.part_wgt
          FROM Parts AS P1, Parts AS P2
         GROUP BY P1.part_nbr, P1.part_wgt
        HAVING SUM(CASE WHEN P2.part_wgt = P1.part_wgt
                        THEN 1 ELSE 0 END)
                 >= ABS(SUM(CASE WHEN P2.part_wgt < P1.part_wgt
                                 THEN 1
                                 WHEN P2.part_wgt > P1.part_wgt
                                 THEN -1
                                 ELSE 0 END))) AS Partitions;
```

31.3.4　ウィンドウ関数を使った中央値

ウィンドウ関数を使うと行をナンバリングできるので、中央値を非常に簡単に求められる。これは要するにテーブルをソートしているのと同じ効果があるからだ。

```
SELECT AVG(x) AS median
  FROM (SELECT part_wgt,
               ROW_NUMBER() OVER(ORDER BY part_nbr ASC),
               ROW_NUMBER() OVER(ORDER BY part_nbr DESC)
          FROM Parts) AS SortedData (x, hi, lo)
 WHERE hi IN (lo, lo+1, lo-1);
```

ほとんど同じだが、少し変形した次のような解もある。

```
SELECT AVG(x)
  FROM (SELECT part_wgt,
               ROW_NUMBER() OVER(ORDER BY part_nbr ASC),
               ROW_NUMBER() OVER(ORDER BY part_nbr DESC)
          FROM Parts) AS SortedData (x, hi, lo)
 WHERE (hi - lo) BETWEEN -1 AND 1;
```

この2つの解は、いずれもパフォーマンスという観点では期待したほど速くないかもしれない。というのは、2つのウィンドウ関数のソート順が違うため、ソートが2回必要だからだ。賢いオプティマイザならば、n行のテーブルをソートしてi番目の行は、逆順にソートした

場合には(n–i ＋1)番目の行になることを利用して、ソートを1回で済ませるかもしれないが、断言はできない。

　ウィンドウ関数を使ったシンプルな中央値を考えてくれたのは、ペソだ。これは少し計算過程を考えなければならない。イメージとしては、集合の要素数を2倍に増やして、もとの集合の要素数を引けば、ちょうど半分の要素が残る、というものだ。要素数が増えても中央値の値は変動しない、という性質をうまく利用している。しかも奇数と偶数の場合分けも必要なくなる。その点で、アイデアとしてはテーブルの行数を2倍に増やしたデイトの解に似ている。

```
SELECT AVG(part_wgt)
  FROM (SELECT part_wgt,
               2 * ROW_NUMBER() OVER(ORDER BY part_wgt)
                 - COUNT(*) OVER() AS y
          FROM Parts) AS TMP
 WHERE y BETWEEN 0 AND 2;
```

　これを重み付け中央値を求めるよう変更すると次のようになる。

```
-- 重み付け中央値
SELECT SUM(y) / SUM(t)
  FROM (SELECT SUM(part_wgt) OVER(PARTITION BY part_wgt) AS y,
               2 * ROW_NUMBER() OVER(ORDER BY part_wgt)
                 - COUNT(*) OVER() AS z,
               COUNT(*) OVER(PARTITION BY part_wgt) AS t
          FROM Parts) AS TMP
 WHERE z BETWEEN 0 AND 2;
```

　これはおそらく本節における最適解だ。だが中央値を求める思考プロセスを示すためにも、やや古い解も含めて紹介してきた。

　アダム・マチャニクはROW_NUMBER関数を使って中央値を求める著者の最初の解に、問題点を指摘した。それが何かを見るために、次のようなテストテーブルを作って重複する値を投入しよう。

```
CREATE TABLE Foo (x INTEGER NOT NULL);
INSERT INTO Foo
VALUES (1), (2), (2), (3), (3), (3);
```

ROW_NUMBER関数を使って中央値を求める最も一般的な方法は次のようなものだ。

```
SELECT AVG(x)
  FROM (SELECT x,
               ROW_NUMBER() OVER(ORDER BY x DESC) AS a,
               ROW_NUMBER() OVER(ORDER BY x ASC) AS b
          FROM Foo) AS Med
 WHERE (b - a) BETWEEN -1 AND 1;
```

残念なことにこれはうまく動かない。結果がNULLになるのだ。その理由を知るために、導出テーブルMedの行を見てみよう。

x	a	b	(a-b)
1	6	1	5
2	4	2	2
2	5	3	2
3	1	4	-3
3	2	5	-3
3	3	6	-3

御覧のように(a–b)が突然一致しなくなる！　問題はROW_NUMBER関数の2 回呼び出しが、別々に行われていることだ。したがって、重複行（このサンプルではxが2と3の行）について2 回とも同じ順序で

ソートされるという保証はない。いわばこの解法は、次のように行がソートされるだろうと暗黙に前提としていたのだ。

x	a	b	(a-b)	
1	6	1	5	
2	5	2	3	
2	4	3	1	◀ 中央値の候補
3	3	4	-1	◀ 中央値の候補
3	2	5	-3	
3	1	6	-3	

この前提は、降順と昇順のROW_NUMBER関数の値は次の式を使って同時に計算されるという考えから来ている。

```
ROW_NUMBER() OVER(ORDER BY x ASC)
  = (COUNT(*) - ROW_NUMBER() OVER(ORDER BY x DESC) + 1)
```

これは巧みなオプティマイザのトリックではあるが、オプティマイザが必ずこういう計算をしてくれる保証があるわけではない。そのため、ORDER BYのリストにキーを追加して、昇順のリストが降順のリストを物理的に反対にしたものと一致することを保証しなければならない。そうしたキーがないテーブルにおいては、このウィンドウ関数の方法だと実行するたびに結果が変わってしまう。

31.4

分散と標準偏差

標準偏差は、正規分布においてある値が平均からどの程度離れているかを示す指標である。これをSQLで計算するのは難しい。なぜなら、これを計算するには平方根が必要になるがSQL は基本的な四則演算しか持っていないからだ。多くのベンダーはそれ以外の数学関数も提供しているが、贔屓目なしで見て、ほとんどのSQL製品は工学的あるいは統計的な計算に使われるニーズはほとんど、というかまったくないと言ってよい。こうした計算を行う一般的な方法は、SQLで計算することを諦め、適切なホスト言語に生データを渡してやることだ。

分散は標準偏差の2 乗として定義される。ということは、平方根を使う必要がないのでピュアSQLの中で計算することができる。そのクエリは次のようになる。

```
CREATE TABLE Samples (x REAL NOT NULL);
INSERT INTO Samples (x)
VALUES (64.0), (48.0), (55.0), (68.0), (72.0),
       (59.0), (57.0), (61.0), (63.0), (60.0),
       (60.0), (43.0), (67.0), (70.0), (65.0),
       (55.0), (56.0), (64.0), (61.0), (60.0);

SELECT ((COUNT(*) * SUM(x * x)) - (SUM(x) * SUM(x)))
       /(COUNT(*) * (COUNT(*)-1)) AS variance
  FROM Samples;
```

結果

```
     variance
=================
 48.9894736842105
```

実際にこのコードを動かして結果をチェックしてみると、正しい答えは48.9894...または丸めの精度によっては49 になるかもしれない。もし読者のSQL 製品が標準偏差の演算子を持っていれば代わりにそれを使ってほしい【訳注2】。

31.5

平均偏差

平均偏差は、絶対値を求めるABS関数を使って求めることができる。コードは次のようになる。

```
BEGIN
  SELECT AVG(x) INTO :in_average FROM Samples;
  SELECT SUM(ABS(x - :in_average)) / COUNT(x) AS AverDeviation
    FROM Samples;
END;
```

これは各値が平均からどの程度離れているかを、離れている方向を無視して求める指標だ。Samplesテーブルに対しては5.1となる。

31.6

累積統計

累積統計とは、あるデータの値とデータ全体の集合との関係を調べる方法だ。よくある例としては、ある集計値の時系列の変化を見るものがある。銀行の預金残高は、預け入れや引き出しが行われることで、継時的に変化していく。

昔のSQL のバージョンでは累積統計を求めるには、一時テーブルと自己結合を使う必要があった。これは非常に複雑で読みにくかった。いまではウィンドウ関数を使えば簡単に書くことができる。預金残高の累積を求めるクエリは次のようになる。

```
CREATE TABLE BankAccounts
(acct_nbr   INTEGER,
 trans_time TIMESTAMP,
 trans_amt  INTEGER,
     PRIMARY KEY(acct_nbr, trans_time));
```

BankAccounts：銀行口座テーブル　acct_nbr：口座番号　trans_date：取引時刻
trans_amt：取引額

```
INSERT INTO BankAccounts
VALUES (1, '2013-02-01 14:00:00', 100),
       (1, '2013-02-01 14:01:00', -50),
       (1, '2013-02-02 18:00:00', 200),
       (2, '2013-02-01 10:00:00', 800),
       (2, '2013-02-05 10:00:00', 200),
       (2, '2013-02-07 10:00:00', -300);
```

```
SELECT acct_nbr, trans_time, trans_amt,
       SUM(trans_amt)
         OVER(PARTITION BY acct_nbr
                  ORDER BY trans_time
              ROWS BETWEEN UNBOUNDED PRECEDING
                       AND CURRENT ROW) AS acct_balance
  FROM BankAccounts;
```

結果

```
acct_nbr      trans_date       trans_amt  acct_balance
=========================================================
       1  2013-02-01 14:00:00        100           100  ←100
       1  2013-02-01 14:01:00        -50            50  ←100 + (-50)
       1  2013-02-02 18:00:00        200           250  ←100 + (-50) + 200
       2  2013-02-01 10:00:00        800           800  ←800
       2  2013-02-05 10:00:00        200          1000  ←800 + 200
       2  2013-02-07 10:00:00       -300           700  ←800 + 200 + (-300)
```

同じ日に預け入れより先に引き出しが行われた場合、預金残高がマイナスなることもあり得る。これは取り付け騒ぎといった、望まれないアクションの引き金になるかもしれない。銀行と付き合うときのルールの1つは、同じ日の中では引き出しよりも預け入れを先にすべし、というものだ。このテーブルでは、取引額が正なら預け入れ、負なら引き出しとみなしているが、これは設計方針としてはあまりよくない。トランザクションの種別を一時的な値に結び付けて解釈しているからだ。これは正しい事実を破壊し、間違った情報をテーブルに登録する危険がある。本来は次のようにトランザクション種別を管理する列を追加すべきだ。`D`（deposit）は預け入れ、`W`（Withdrawal）は引き出しを意味する。

```
trans_type CHAR(1) DEFAULT 'W' NOT NULL
    CHECK (trans_type IN ('D', 'W'))
```

31.6.1　累積差分

累積統計のもう1 つの例が累積差分である。これはさまざまな時点のなんらかの合計値が、直近からどの程度変化したかを求めるものだ。たとえばこんな例を考えよう。タンクの中にある薬品が入って

いて、ある時刻における薬品の量を記録している。そのデータを使って、前回計測したときからの変化量を報告しなければならない。以下にサンプルの結果と、我々が得たい結果を示す。

```
CREATE TABLE Deliveries
(tank_nbr     CHAR(3),
 reading_time TIMESTAMP,
 tank_qty     INTEGER,
     PRIMARY KEY(tank_nbr, reading_time));
```

Deliveries：タンク容量テーブル　tank_nbr：タンク番号　reading_time：計測時刻　tank_qty：タンク容量

tank_nbr	reading_time	tank_qty	tank_diff	
50A	2015-02-01 07:30	300	NULL	◀ 開始時点
50A	2015-02-01 07:35	500	200	
50A	2015-02-01 07:45	1200	700	
50A	2015-02-01 07:50	800	-400	
50A	2015-02-01 09:00	1300	500	
51A	2015-02-01 00:45	5000	NULL	◀ 開始時点
51A	2015-02-01 01:00	2500	-2500	
51A	2015-02-01 07:20	6000	3500	
51A	2015-02-01 07:22	8000	2000	

累積差分を計算するトリックは相関サブクエリを使うことである。

```
SELECT tank_nbr, reading_time,
       (tank_qty
           - (SELECT tank_qty
                FROM Deliveries AS D1
               WHERE D1.tank_nbr = D0.tank_nbr -- 同じタンク
                 AND D1.reading_time
                     = (SELECT MAX (D2.reading_time) -- 直近
                          FROM Deliveries AS D2
                         WHERE D2.tank_nbr = D0.tank_nbr -- 同じタンク
                           AND D2.reading_time < D0.reading_time))) AS tank_diff
  FROM Deliveries AS D0
 ORDER BY tank_nbr, reading_time;
```

あるいは、ウィンドウ関数を使って次のように書いてもよい。

```
SELECT tank_nbr, reading_time, tank_qty,
       (tank_qty - MAX(tank_qty)
                   OVER(PARTITION BY tank_nbr
                            ORDER BY reading_time ASC
                             ROWS BETWEEN 1 PRECEDING
                                      AND 1 PRECEDING)) AS tank_diff
  FROM Deliveries
 ORDER BY tank_nbr, reading_time;
```

31.6.2　累積率

累積率は累積和や累積差分よりも複雑だ。これは、現在のデータの値が、集合全体の値に対し何パーセントであるかを示す指標だ。これも言葉で説明するより、サンプルを見るほうが理解しやすい。セールスマンの売上げを管理する次のようなテーブルを考える。

```
CREATE TABLE Sales
(salesman_id CHAR(10)NOT NULL,
 client_name CHAR(10) NOT NULL,
 sales_amt DECIMAL (9,2) NOT NULL,
     PRIMARY KEY (salesman_id, client_name));
```

Sales：売上げテーブル　　salesman_id：セールスマンID　　client_name：顧客名
sales_amt：売上げ額

```
INSERT INTO Sales
VALUES ('S01', 'C01', 100), ('S01', 'C02', 200), ('S01', 'C03', 200),
       ('S02', 'C02',  50), ('S02', 'C03', 100),
       ('S03', 'C04', 500), ('S03', 'C02', 600), ('S03', 'C05', 500),
       ('S04', 'C03', 300), ('S04', 'C04', 100);
```

やりたいことは、各セールスマンの売上げに占める顧客の割合を求めることだ。それを使って、大口顧客の順にソートしたい。この問題は、ある小さな広告代理店で実際に営業レポートを作ろうとしたときの問題をもとにしている。これをもとに、ビジネスの予測を行い、売上げに貢献していない顧客（下位20%に位置する）は、新しいセールスマンにやらせてみるという決断がなされた。セールスマンは皆、このレポートを食い入るように読んで、どの顧客が自分の上位80%売上げに入っているかを確認していた。

昔はこの問題を解くには導出テーブルを使う必要があったが、これは非常にパフォーマンスが悪かった。いまではウィンドウ関数を使えばもっと効率的に書ける。

```
SELECT salesman_id, client_name, sales_amt,
       SUM(sales_amt)
           OVER(PARTITION BY salesman_id
                 ORDER BY sales_amt DESC
                RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS salesman_running_tot,
       SUM(sales_amt)
           OVER(PARTITION BY salesman_id) AS salesman_tot,
       SUM(sales_amt)
           OVER(PARTITION BY salesman_id
                 ORDER BY sales_amt DESC
                RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        /  SUM(sales_amt)
           OVER(PARTITION BY salesman_id) * 100 AS cume_percent
  FROM Sales;
```

結果

```
salesman_id client_name sales_amt salesman_running_tot salesman_tot cume_percent
================================================================================
S01         C02            200.00               400.00       500.00        80.00
S01         C03            200.00               400.00       500.00        80.00
S01         C01            100.00               500.00       500.00       100.00
S02         C03            100.00               100.00       150.00        66.66
S02         C02             50.00               150.00       150.00       100.00
S03         C02            600.00               600.00      1600.00        37.50
S03         C04            500.00              1600.00      1600.00       100.00
S03         C05            500.00              1600.00      1600.00       100.00
S04         C03            300.00               300.00       400.00        75.00
S04         C04            100.00               400.00       400.00       100.00
```

この結果は、分散関数の結果とは微妙に異なるので注意してほしい。分散関数は、ある行について、パーティション内における相対順位を計算し、0から1の数値で表現する。相対度数を求めるPERCENT_RANKと累積分散を求めるCUME_DISTの2つがある。これらはROW_NUMBER()、RANK()、DENSE_RANK()といった通常の順序関数の拡張である。

PERCENT_RANKは、ある行の相対順位を(RK-1)/(NR-1)で定義する。RKはパーティション内に含まれる当該行の順位で、NR はパーティション内に含まれる全行数である。CUME_DIST は、ある行の相対順位をNP/NR で定義する。NPはウィンドウ内で当該の行よりも順位が上の行の数で、NR はパーティション内に含まれる全体の行数である。

```
SELECT salesman_id, client_name, sales_amt,
       PERCENT_RANK()
         OVER(PARTITION BY salesman_id
                  ORDER BY sales_amt DESC) * 100 AS percentage_of_total,
       CUME_DIST()
         OVER(PARTITION BY salesman_id
                  ORDER BY sales_amt DESC) * 100 AS cum_percent
  FROM Sales;
```

結果

```
 salesman_id  client_name sales_amt percentage_of_total cum_percent
=================================================================
 S01          C02            200.00                   0       66.67
 S01          C03            200.00                   0       66.67
 S01          C01            100.00                 100         100
 S02          C03            100.00                   0          50
 S02          C02             50.00                 100         100
 S03          C02            600.00                   0       33.33
 S03          C04            500.00                  50         100
 S03          C05            500.00                  50         100
 S04          C03            300.00                   0          50
 S04          C04            100.00                 100         100
```

相対度数は定義上、最初の値の行は必ず0になる点に注意してほしい。

31.6.3　順序関数

次のようなSalesReportテーブルがあるとして、セールスマンに順位をつけるにはどうすればいいだろう？

```
CREATE TABLE SalesReport
(salesman_id CHAR(20) NOT NULL PRIMARY KEY,
 sales_tot   INTEGER  NOT NULL);
```

SalesReport：売上げ報告テーブル　　salesman_id：セールスマンID　　sales_tot：売上げ

SalesReport

salesman_id	sales_tot
Wilson	990
Smith	950

Richards	800
Quinn	700
Parker	345
Jones	345
Hubbard	345
Date	200
Codd	200
Blake	100

　要するにランキング（1 位、2 位、3 位……）を出したいのだが、難しいのは、同位タイの扱い方だ。もし複数のセールスマンが同じ売上げだった場合、順位も同じになる。問題は、そのあとの順位に欠番を作るかどうかである。欠番を許すのがRANK関数、許さないのがDENSE_RANK 関数である。次のクエリは、DENSE_RANKと同じ結果を返す。

```
SELECT S1.salesman_id, S1.sales_tot,
       (SELECT COUNT(DISTINCT sales_tot)
          FROM SalesReport AS S2
         WHERE S2.sales_tot >= S1.sales_tot) AS rank
  FROM SalesReport AS S1;
```

結果

```
salesman_id  sales_tot  rank
============================
Wilson             990     1
Smith              950     2
Richards           800     3
Quinn              700     4
Parker             345     5
Jones              345     5
Hubbard            345     5
Date               200     6
Codd               200     6
Blake              100     7
```

読者は、これは極値関数の一般化と同じクエリであることを覚えているだろう（29.4節参照）。このクエリをベースとして、結果を微妙に変える4つのバリエーションを考えることができる。

バージョン1　COUNT(DISTINCT)と>=

```
SELECT S1.salesman_id, S1.sales_tot,
       (SELECT COUNT(DISTINCT sales_tot)
          FROM SalesReport AS S2
         WHERE S2.sales_tot >= S1.sales_tot) AS rank
  FROM SalesReport AS S1;
```

これがベースである。

バージョン2　COUNT(DISTINCT)と> …… 0から始まる

```
SELECT S1.salesman_id, S1.sales_tot,
       (SELECT COUNT(DISTINCT sales_tot)
          FROM SalesReport AS S2
         WHERE S2.sales_tot > S1.sales_tot) AS rank
  FROM SalesReport AS S1;
```

結果

```
salesman_id  sales_tot  rank
============================
Wilson             990     0
Smith              950     1
Richards           800     2
Quinn              700     3
Parker             345     4
Jones              345     4
Hubbard            345     4
Date               200     5
Codd               200     5
Blake              100     6
```

バージョン3　COUNT(ALL)と>= …… 1から始まる

```
SELECT S1.salesman_id, S1.sales_tot,
       (SELECT COUNT(sales_tot)
          FROM SalesReport AS S2
         WHERE S2.sales_tot >= S1.sales_tot) AS rank
  FROM SalesReport AS S1;
```

結果

```
salesman_id  sales_tot  rank
============================
Wilson           990        1
Smith            950        2
Richards         800        3
Quinn            700        4
Parker           345        7
Jones            345        7
Hubbard          345        7
Date             200        9
Codd             200        9
Blake            100       10
```

バージョン4　COUNT(ALL)と> …… 0から始まる

```
SELECT S1.salesman_id, S1.sales_tot,
       (SELECT COUNT(sales_tot)
          FROM SalesReport AS S2
         WHERE S2.sales_tot > S1.sales_tot) AS rank
  FROM SalesReport AS S1;
```

結果

```
salesman_id  sales_tot  rank
============================
Wilson             990     0
Smith              950     1
Richards           800     2
Quinn              700     3
Parker             345     4
Jones              345     4
Hubbard            345     4
Date               200     7
Codd               200     7
Blake              100     9
```

イギリスでは、学校の成績や競馬のランキングに、欠番を認めつつ、少し変わった体系を使っている。歯抜けの生じる方向が逆なのだ。たとえば、次のような得点と順位が含まれたテーブルを考える。

Marks	class_standing
100	1
90	2
90	2
70	4

90点の生徒2人は2位にランクされる。なぜなら、最高点をマークした生徒が1人しかいないからだ。70点の生徒は4位である。これは、前に3人の生徒がいるからだ【訳注3】。我々が使っているサンプルデータを使うと、このタイプの結果を求めるクエリは次のようになる。

```
SELECT S1.salesman_id, S1.sales_tot,
       (SELECT COUNT(S2.sales_tot)
          FROM SalesReport AS S2
         WHERE S2.sales_tot > S1.sales_tot) + 1 AS british
  FROM SalesReport AS S1;
```

結果

```
salesman_id  sales_tot  british
==============================
Wilson             990        1
Smith              950        2
Richards           800        3
Quinn              700        4
Parker             345        5
Jones              345        5
Hubbard            345        5
Date               200        8
Codd               200        8
Blake              100       10
```

私は常に、SQL で問題を解くコツは、集合の観点から考えることだと説いている。我々がこのランキングを求めるクエリで探しているのは、SalesReportテーブルの部分集合に順序数を割り当てる方法である。この部分集合は、カレント行のセールスマンと同じか、より多くの売上げを挙げたセールスマンからなる。言い換えれば、SalesReport のコピーの1 つ（S2）は部分集合の要素を与え、もう1つのコピー（S1）は下限の境界を与えるのだ。

こうして得られる要素のカウントは、実は入れ子集合のシーケンスである。集合論の講義を覚えていれば、これがジョン・フォン・ノイマンによる順序数の再帰的定義と同型であることに気づくだろう【訳注4】。ノイマンは、n番目の数を、n−1番目までの数を含む集合として定義したのだ。

31.6.4　五分位と関連の指標

ランキングを算出できれば、そのデータセットを五分位、十分位、および任意のパーセンタイル（百分位数）に分類するのはかなり簡単だ。分位数は、データセットを同じ要素数の複数のパーティションに切り分けたものだ（テーブルの行数によっては、完全にパーティションごとの要素数が同じになるわけではないが）。五分位は5個の、十分位は10個のパーティションに分ける。パーセンタイルは、適当な100分率で切り分ける（たとえば90パーセンタイルといえば、上位10%と下位90%という2つのパーティションに分類する）。ここでは、五分位を計算するクエリを紹介する。あとはこのクエリを少し変更すれば好きなように一般化できる。この統計指標は学校でも習うので馴染み深いだろう。

```
SELECT salesman_id, sales_tot, rank,
       CASE WHEN T1.rank <= 0.2 * T2.population_size THEN 1
            WHEN T1.rank <= 0.4 * T2.population_size THEN 2
            WHEN T1.rank <= 0.6 * T2.population_size THEN 3
            WHEN T1.rank <= 0.8 * T2.population_size THEN 4
            ELSE 5 END AS quintile
  FROM (SELECT S1.salesman_id, S1.sales_tot,
               (SELECT COUNT(S2.sales_tot)
                  FROM SalesReport AS S2
                 WHERE S2.sales_tot >= S1.sales_tot)
          FROM SalesReport AS S1) T1(salesman_id, sales_tot, rank)
            CROSS JOIN
              (SELECT COUNT(*) FROM SalesReport) AS T2(population_size);
```

結果

```
salesman_id  sales_tot  rank  quintile
====================================
Wilson             990     1         1
Smith              950     2         1
Richards           800     3         2
Quinn              700     4         2
Parker             345     7         4
Jones              345     7         4
Hubbard            345     7         4
Date               200     9         5
Codd               200     9         5
Blake              100    10         5
```

考え方は明快だ。まず各要素のランキングを算出し、全体の行数から決定される各パーティションのサイズに合うよう行を割り振っていく。こうした計算を行うための拡張は、ベンダーによってサポート状況がバラバラなので、注意してほしい。

31.7

クロス表

クロス表は、統計レポートでよく使われるフォーマットだ。これを作るためのベンダーが提供するツールもある。たとえば、IBMのQMFでは、ACROSSオプションを使えば可能だ。他にも、SPSSやSASは、クロス表を作るためのライブラリや機能を持っている。また、多くのスプレッドシートは、データをロードしてクロス表を作る機能を持っている。代表例はExcelのピボットテーブルだ。

もしサーバー側で動作するレポーティングパッケージを使っているなら、これから紹介する方法には目もくれず、そちらを使ってほしい。そのほうが速いしリソース消費も少ない。しかし、そのようなパッケージを持っていないか、クライアント側でレポーティングを行わねばならない場合は、いまから紹介する方法をサーバー側で実行するほうが速いだろう。貧弱なクライアントで重い処理を実行するのは性能の担保が難しく、またネットワーク回線の負荷も大きくなる。

といっても、現在の標準SQLは、GROUP BY句でCUBEとROLLUPオプションを使えば、クロス表のデータを得ることができる。したがって以下のプログラミングトリックは、いまとなっては主に歴史的価値しかない。

クロス表のうち、1 つしか軸を持たないクロス表は、テーブルを「水平展開」してそれを帳票フォーマットにしたものである。たとえば、製品別の売上げと製品が売れた日を管理するテーブルを考える。ここから、1年ごとの製品の売上げを求めるレポートを10年分出

力したい。これを作る方法の1つは、年のビット列を持ったクロステーブルを作り、それと結合するという方法がある。

```
CREATE TABLE Sales
(product_name CHAR(15) NOT NULL,
 product_price DECIMAL(5,2) NOT NULL,
 order_qty INTEGER NOT NULL,
 sales_year INTEGER NOT NULL);
```

Sales：売上げテーブル　　product_name：製品名　　product_price：価格
order_qty：注文数量　　sales_year：売上げ年

```
CREATE TABLE Crosstabs
(sales_year INTEGER NOT NULL,
 year1 INTEGER NOT NULL,
 year2 INTEGER NOT NULL,
 year3 INTEGER NOT NULL,
 year4 INTEGER NOT NULL,
 year5 INTEGER NOT NULL,
 row_total INTEGER NOT NULL);
```

Crosstabs：クロス表テーブル　　sales_year：売上げ年　　year1～5：年1～5
row_total：行合計

クロス表テーブルには、次のようなデータを入れる。

sales_year	year1	year2	year3	year4	year5	row_total
1990	1	0	0	0	0	1
1991	0	1	0	0	0	1
1992	0	0	1	0	0	1
1993	0	0	0	1	0	1

1994	0	0	0	0	1	1

レポートを作成するクエリは次のようになる。

```
SELECT S1.product_name,
       SUM(S1.order_qty * S1.product_price * C1.year1),
       SUM(S1.order_qty * S1.product_price * C1.year2),
       SUM(S1.order_qty * S1.product_price * C1.year3),
       SUM(S1.order_qty * S1.product_price * C1.year4),
       SUM(S1.order_qty * S1.product_price * C1.year5),
       SUM(S1.order_qty * S1.product_price * C1.row_total)
  FROM Sales AS S1, Crosstabs AS C1
 WHERE S1.sales_year = C1.sales_year
 GROUP BY S1.product_name;
```

当然ながら、SUM(S1.product_price * S1.order_qty)は、ある1 年についての各製品の売上げ合計である。yearN 列の値は、1または0 である。もし0 なら、SUM 関数で計算される合計額は0になる。もし1なら、SUM関数の合計額は変化しない。このクエリの利点は、適当な年の列を作ることで、期間の範囲を調節できることだ。

2軸のクロス表は2つの変数を取り、一方の変数のすべての値を行に、もう一方の変数のすべての値を列に展開した表を作る。テーブルの各セルの値は、2つの変数を共通して持つエンティティの数である。NULLとクロス表の相性は、NULLを1つのグループとして扱うよう取り決めるか、排除しない限り、あまりよいとは言えない。

個々の変数のセル以外にも、列合計と行合計、および総計のセルが存在する。一般にn 個の変数のクロス表は、n次元のスプレッドシートを作ることで定義される。だが、n次元を2次元の紙に印刷することは難しい。これをやる際によく使われるトリックは、結果を2 次元のグリッドで表し、その片方または両方の軸を、ブレイクダウン

可能な木構造として定義することだ。たとえば「性別の中の人種」は、性別を人種でブレイクダウン可能なことを意味する。他方、「人種の中の性別」は、人種が性別でブレイクダウン可能ということだ。

社員番号を主キーとするテーブルを考えよう。どの列もNULLは許可しないとする。いま、社員を性別と人種の2軸で分類するクロス表を作る。結果は次のような形だ。

```
CREATE TABLE Personnel
(emp_nbr  INTEGER PRIMARY KEY,
 sex      CHAR(1),
 race     CHAR(8),
 job_nbr  INTEGER,
 salary_amt INTEGER);
```

Personnel：社員テーブル　emp_nbr：社員番号　sex：性別　race：人種
job_nbr：職種番号　salary_amt：給与

```
INSERT INTO Personnel
VALUES (1, '1', 'Asian', 1, 300),
       (2, '2', 'Asian', 2, 500),
       (3, '2', 'Black', 1, 300),
       (4, '1', 'Black', 2, 550),
       (5, '2', 'Black', 1, 300),
       (6, '1', 'Black', 2, 500),
       (7, '1', 'Black', 1, 700),
       (8, '1', 'Cauc', 1, 700),
       (9, '2', 'Cauc', 1, 600),
       (10, '1', 'Hisp', 1, 500),
       (11, '2', 'Hisp', 1, 300),
       (12, '1', 'Hisp', 1, 700);
```

求めたい結果の形は次のとおりだ。

```
sex       cauc  black  asian  latino  other  race_total
=========================================================
2            1      2      1       1      0           5
1            1      3      1       2      0           7
Sex Total    2      5      2       3      0          12
```

最初に思いつくのは、GROUP BYを使ったシンプルなクエリだろう。

```
SELECT sex, race, COUNT(*)
  FROM Personnel
 GROUP BY sex, race;
```

結果

```
sex  race   count
================
1    Asian      1
1    Black      3
1    Cauc       1
1    Hisp       2
2    Asian      1
2    Black      2
2    Hisp       1
2    Cauc       1
```

このアプローチは、2変数の場合はうまくいく。結果をレポート出力用のプログラムに渡せば、欲しかった形の表が得られる。だが1 つ問題がある。列と行の合計はどこへ行ったのだろう？ これらを得るには別個に次のようなクエリを書いてUNIONでつなげる必要がある。

```
SELECT race, COUNT(*) FROM Personnel GROUP BY race;
SELECT sex, COUNT(*) FROM Personnel GROUP BY sex;
```

CUBEを使うことでも同じ結果を得られる。

```
SELECT sex, race, COUNT(*)
  FROM Personnel
 GROUP BY CUBE(sex, race);
```

結果

```
sex  race   count
================
               12
     Asian      2
     Black      5
     Cauc       2
     Hisp       3
1               7
1    Asian      1
1    Black      3
1    Cauc       1
1    Hisp       2
2               5
2    Asian      1
2    Black      2
2    Cauc       1
2    Hisp       1
```

ここでさらに、職種番号という第3の変数を軸に加えるとしたらどうだろうか。求める結果は、職種別の性別と人種の値とそれぞれの軸の合計になる。CUBEを使えばこのような組み合わせを作るのは簡単だ。

31.7.1 CASE 式によるクロス表

おそらく最良の方法はCASE 式を使うことだ。もし最終行に合計も欲しければ、次のようにUNIONで追加すればいい。

```
SELECT sex,
       SUM(CASE race WHEN 'Cauc' THEN 1 ELSE 0 END) AS cauc,
       SUM(CASE race WHEN 'Black' THEN 1 ELSE 0 END) AS black,
       SUM(CASE race WHEN 'Asian' THEN 1 ELSE 0 END) AS asian,
       SUM(CASE race WHEN 'Hisp' THEN 1 ELSE 0 END) AS latino,
       SUM(CASE race WHEN 'Other' THEN 1 ELSE 0 END) AS other,
       COUNT(*) AS race_total
  FROM Personnel
 GROUP BY sex
UNION ALL
SELECT 'Sex Total',
       SUM(CASE race WHEN 'Cauc' THEN 1 ELSE 0 END),
       SUM(CASE race WHEN 'Black' THEN 1 ELSE 0 END),
       SUM(CASE race WHEN 'Asian' THEN 1 ELSE 0 END),
       SUM(CASE race WHEN 'Hisp' THEN 1 ELSE 0 END),
       SUM(CASE race WHEN 'Other' THEN 1 ELSE 0 END),
       COUNT(*) AS race_total
  FROM Personnel;
```

結果

```
sex        cauc  black  asian  latino  other race_total
=====================================================
2             1      2      1       1      0          5
1             1      3      1       2      0          7
Sex Total     2      5      2       3      0         12
```

この方法の利点は何といっても、フォーマットまできちんとしたクロス表の形で出力されるので、整形の必要がないことだ。現在ならば次のようにCUBEを使って書いてもよい。

```
SELECT sex,
       SUM(CASE race WHEN 'Cauc' THEN 1 ELSE 0 END) AS cauc,
       SUM(CASE race WHEN 'Black' THEN 1 ELSE 0 END) AS black,
       SUM(CASE race WHEN 'Asian' THEN 1 ELSE 0 END) AS asian,
       SUM(CASE race WHEN 'Hisp' THEN 1 ELSE 0 END) AS latino,
       SUM(CASE race WHEN 'Other' THEN 1 ELSE 0 END) AS other,
       COUNT(*) AS row_total
  FROM Personnel
 GROUP BY CUBE(sex);
```

31.8

調和平均と幾何平均

調和平均は、逆数の算術平均の逆数として定義される。これは、比率などを扱うのに適している。用途は限られているが、主に平均料率などの計算に見られる。

```
SELECT COUNT(*) / SUM(1.0 / nbr) AS harmonic_mean
  FROM NumberTable;
```

幾何平均は、相乗平均とも呼ばれ、数値を足すのではなく積の冪根をとったものである。幾何平均は、特に時系列で変化するデータの場合、単純な算術平均よりも中央の性質をよく表す場合がある【訳注5】。

```
SELECT EXP (AVG (LOG (nbr))) AS geometric_mean
  FROM NumberTable;
```

なお、幾何平均は負数には使えない。0以下の数に対する対数が定義されていないからだ。

31.9

SQL の記述統計における多変量

近年、SQL 製品はどんどん複雑な記述統計の関数をライブラリに追加していっている。これまでコードを紹介してきた、1 列で機能する集約関数の多くは、ビルトイン関数だった。こうした関数は、浮動小数点や丸めの精度もどんどん改善されていくだろう。

だが、記述統計の関数のすべてが1 列だけで機能するわけではない。複数の変数間の関係を記述したり予測したい場合も多いだろう。ここで、そうした統計の代表例を取り上げ、これをいままでに見た集約関数で記述しようとするとどのような問題があるか、ということを見ていこう。

31.9.1　共分散

共分散は、2つの変数がどの程度一緒に変動するかを測るための指標である。金融分析では、2 つの有価証券の収益が時系列で見てどの程度関連しているかを見るためにこの指標を使う。共分散が高い値を示すことは、2 つの変数が連動して動くことを意味する。以下のコードはスティーブ・キャスによる。

```
CREATE TABLE Samples
(sample_nbr INTEGER NOT NULL PRIMARY KEY,
 x FLOAT NOT NULL,
 y FLOAT NOT NULL);

INSERT INTO Samples
VALUES (1, 3, 9), (2, 2, 7), (3, 4, 12), (4, 5, 15), (5, 6, 17);

WITH A (n, x, x_avg, y, y_avg)
AS
(SELECT COUNT(*) OVER() AS n,
         x, AVG(x) OVER() AS x_avg,
         y, AVG(y) OVER() AS y_avg
   FROM Samples)
SELECT ((1.0/n) * SUM((x - x_avg) * (y - y_avg))) AS covariance
  FROM A
 GROUP BY n;
```

結果

```
 covariance
============
        5.2
```

31.9.2　ピアソンのr

統計指標の中で非常に便利な1 つがピアソンのr、または線形相関係数である。これは、2 つの変数の間の線形の関係の強さを測る。日本語で言えば、変数の組(x1, y1), (x2, y2), ... , (xn, yn)があった場合に、一方の変数が大きくなったり小さくなったりしたとき、もう一方の変数がどのような動きを見せるかを測るための概念だ。

相関係数は常に＋1と–1の範囲に収まる。＋1は2 つの変数が完全に同期していることを意味する。–1は反対に、一方の変数が大きくなると必ずもう一方の変数が小さくなることを意味する。0に近いほど2つの変数は無関係であることを意味する。

現実においては、＋1や−1の相関係数を見ることはそうそうない。というのも、このような値を取るということは、2変数は統計的関係というよりも物理法則によって結び付けられているからだ。実際には、絶対値で0.7以上ならば強い相関とみなしてよい。

相関係数を計算する公式は、SQLでもダイレクトに記述できる。

```
CREATE TABLE Sample
(student_id CHAR(3) NOT NULL PRIMARY KEY,
 reading_score DECIMAL(5,2) NOT NULL,
 gpa_score DECIMAL(5,2) NOT NULL);

INSERT INTO Sample
VALUES
('S1', 38, 2.1),
('S2', 54, 2.9),
('S3', 43, 3.0),
('S4', 43, 2.3),
('S5', 50, 2.6),
('S6', 61, 3.7),
('S7', 57, 3.2),
('S8', 25, 1.3),
('S9', 36, 1.8),
('S10', 39, 2.5),
('S11', 48, 3.4),
('S12', 46, 2.6),
('S13', 44, 2.4),
('S14', 39, 2.5),
('S15', 48, 3.3);

WITH A (n, xy_sum, x_sum, y_sum, x2_sum, y2_sum)
AS
(SELECT COUNT(*),
        SUM(reading_score * gpa_score),
        SUM(reading_score),
        SUM(gpa_score),
        SUM(reading_score * reading_score),
        SUM(gpa_score * gpa_score)
  FROM Sample)
SELECT ((n * xy_sum) - (x_sum * y_sum))
/ (SQRT ((n * x2_sum -(x_sum * x_sum))
        *(n * Y2_sum -(y_sum * y_sum)))) AS pearson
  FROM A;
```

結果

```
 pearson
========
   0.874
```

31.9.3　多変量の記述統計におけるNULL

もし(x, y) = (NULL, NULL)なら、集約関数はこのペアを計算から除外する。これはSQL の通常のルールに即した動作だ。だが、(x, y)のペアのうち片方だけがNULLの場合の正しい（言い換えれば合理的な）計算はどのようなものだろうか。これはいくつかのパターンが考えられる。

1. 片方でもNULL だった場合は、計算から除外する。これは手っ取り早く安易な解決策だ。「WHERE x IS NOT NULL AND y IS NOT NULL」という条件をクエリに付けておけばよい。これを支持する理由としては、もし一方または両方の値がわからないのなら、関係だってわからないはずだ、というものだ。
2. (x, NULL)を(x, AVG(y))に、(NULL, y)を(AVG(x), y)に変換する。これは合理的な代替値によってNULLを推測しようという考えである。平均を使うよりもよい代替値がある場合もあるかもしれないが、いずれにせよ、アイデア自体は悪くない。
3. (NULL, NULL)を適当な値aを使って(a, a)に変換する。これは、どうせNULLを保存するならNULLは1 つのグループをなすのだから、適当な値で置き換えてもよい、という考えに基づいている。この種の「擬似同値性」は、GROUP BY の演算の基礎に

なっている。だがこのルールだと、(x, NULL)と(y, NULL)の場合に、NULLをどんな値で代替するのが適当かわからない。あまり現実的とは言えない選択肢だ。

4. まず既知のペアから線形回帰を計算する。たとえば、y =(a + b * x)という公式を使う。そして、期待される値を計算する。すなわち、(x[i], NULL)に対しては、y[i]= a + b *x[i]、(NULL, y[i])に対しては、x[i]=(y-a)/ b という式で算出する。問題は、2つの変数の間に必ずしも線形の相関が存在するとは前もって断言できないことだ。
5. 標準SQLの流儀に従えば、集約関数がNULLを除外する場合、SQLSTATEに警告のコードがセットされる。これをキャッチして、ユーザーにNULL が存在したことを知らせ、判断を促す。COUNT(*)とCOUNT(x + y)を使えば、NULL の数もわかる。実際、非NULL のペアのほうが少なくて、NULLばかりのデータセットだったりしたら、上に挙げたような方法で既知の値からNULLを推計したとしても、精度が悪くなり、信頼がおけない。

この問題に唯一の正しい答えは存在しない。読者の扱うデータ特性に応じて適宜判断してほしい。

31.10

SQL:2006 における統計関数

以下にSQL:2006 で標準に取り入れられた統計用の関数を紹介する。これらはまだ広く実装されているわけではないが、SQLもこうした基本的な統計解析機能ぐらいは持っておくことが求められる時代になったということである。通常のプログラミング言語の同じ機能と違うところは、NULLを扱うためにSQL向けのアレンジが加えられていることだ。なお、統計の用語や概念については本書では解説しないので、適切な統計の書籍をあたってほしい。

率直に言って、SQL がこのような拡張を行うことについては賛成できない。できればこうした機能がユーザーの心をつかまないことを望む。原理的なことを言えば、こうした機能はデータを情報の縮約するためにあるのに対し、SQLはデータを整合的に保ち、検索を行うための言語だ。そもそもSQLにこうした拡張を行うのは、筋違いであると思う。また現実的な問題としても、SQLは統計言語が必要とする精度と厳密さを持った演算を行う力を持っていない。

31.10.1　分散、標準偏差、記述統計

独立変数X と従属変数Y を持つテーブルがあるとする。N をこのテーブルの行数としよう。SUMX は列X の合計値、SUMY は列Y の合計値である。SUMX2を列X の2 乗の合計値、SUMY2は従属変数の2 乗の合計値である。さらに、SUMXYはXとYの値を乗じたものである。このとき、以下のような指標が定義できる。

- **STDDEV_POP**：母標準偏差。これはSQRT(VAR_POP(X))で求められる。

- **STDDEV_SAMP**：標本標準偏差。
- **VAR_POP**：母分散。これはSQRT(VAR_SAMP(X))で求められる。
- **VAR_SAMP**：標本分散。
- **COVAR_POP**：母集団の共分散。
- **COVAR_SAMP**：標本の共分散。もしN が1 の場合、COVAR_SAMPの値はNULLになる。それ以外は、(SUMXY-SUMX * SUMY/N)/(N-1)で求められる。

31.10.2　相関

- **CORR**：ピアソンのr の相関。もしN * SUMX2 = SUMX * SUMXなら、結果はNULL になる。もしN * SUMY2 = SUMY * SUMYなら、やはりNULLである。それ以外の場合、結果はSQRT(POWER(N * SUMXY-SUMX * SUMY, 2)/((N * SUMX2-SUMX * SUMX)*(N * SUMY2-SUMY * SUMY)))で求められる。
- **REGR_SLOPE**：線形回帰の傾向を1つの直線で表した指標。
- **REGR_INTERCEPT**：線形回帰方程式のy切片。
- **REGR_COUNT**：結果はNである。もしNが0なら、結果はNULLである。
- **REGR_R2**：もしN * SUMX2 = SUMX * SUMXなら、結果はNULLである。もしN * SUMY2がSUMY * SUMYと等しいなら、結果は1 である。それ以外の場合、以下の式になる。POWER (N * SUMXY-SUMX * SUMY, 2)/((N * SUMX2-SUMX *

SUMX)*(N * SUMY2-SUMY * SUMY))。もし演算の結果、指数が実装でデータ型に定義されている範囲に収まらなかった場合、結果はNULLになる。

以下に簡略版を示す。

```
REGR_AVGX = SUMX/N
REGR_AVGY = SUMY/N
REGR_SXX = (SUMX2-SUMX * SUMX/N)
REGR_SYY = (SUMY2-SUMY * SUMY/N)
REGR_SXY = (SUMXY-SUMX * SUMY/N)
```

【訳注1】　中央値は、OracleがMEDIANという関数名でサポートしている。

【訳注2】　分散は、Oracle、PostgreSQLがサポートしている（どちらもVARIANCEという名前）。

【訳注3】　このランキング体系では、同位が並んだ「後」の順位（3位）が欠番になっている。いわゆる「1224」型である。反対に同位の「前」を欠番にするのが「1334」型である。こちらのが日本含め一般的であろう。

【訳注4】　フォン・ノイマンによる順序数の定義は以下のように、0を空集合として、再帰的定義を使って行う。

0 = φ

1 = {0}

2 = {0, 1}

3 = {0, 1, 2}

...

この方法とランキングを求めるクエリとのつながりに

ついては、拙著『達人に学ぶ SQL 徹底指南書』の1-2章でも解説しているので興味がある方は参照してもらいたい。

【訳注5】　たとえば、成長率を求める際には幾何平均がよく使われる。

32

SQLにおける数列の扱い

Sub-Sequences, Regions, Runs, Gaps, and Islands

本書では、これまでに何度もGROUP BY句の使い方について見てきた。GROUP BY句が作るグループには順序はない。だが、グループ間になんらかの順序を持たせたい場合も多いだろう。こういうデータの例としては、チケットナンバーやある固定的な期間による時系列のデータ等が考えられる。こうしたデータの一部が欠けていたり、あるいは関心のある一部の数列だけを抜き出したいと思うこともあるだろう。具体例を見たほうがわかりやすい。次のような連番のキーを持つ7行のテーブルを考えよう。

```
CREATE TABLE List
(list_seq INTEGER NOT NULL PRIMARY KEY,
 list_val INTEGER NOT NULL UNIQUE);
```

List：リストテーブル　　list_seq：リスト連番　　list_val：リスト値

```
INSERT INTO List
VALUES (1, 99), (2, 10), (3, 11), (4, 12), (5, 13), (6, 14), (7, 0);
```

list_val 列の区間は増加、減少、変化しないの3 つの傾向のいずれかになる。ここで値が1 ずつ増加している区間に着目しよう。このような傾向を持つ区間は3つある。(10, 11, 12)、(11, 12, 13)、(12,

13, 14)である。だが最も長い区間は(10, 11, 12, 13, 14)という5個の数字を持つ区間である。

区間と似た概念に単調区間というものがある。これは含まれる数字が隣り合っている必要がなく、ただ数列が単調増加または単調減少していればよい。たとえば、{(1, 1), (2, 2), (3, 12), (4, 15), (5, 23)}という単調区間が与えられた場合は、(1, 2, 12)、(2, 12, 15)、(12, 15, 23)という3 つの単調区間が含まれている。

次にリージョンという概念がある。これは連続して同じ値を持つ区間だ。たとえば、{(1, 1), (2, 0), (3, 0), (4, 0), (5, 25)}は0について長さ3のリージョンを含んでいる。

手続き型言語でこうしたシーケンスを見つけ出すには、単純にデータをソートしてスキャンすればよい。かつてSQLでは、すべてを集合の観点から定義し、複雑な結合を使ってデータを順序づける必要があった。今日ではROW_NUMBER 関数を使えば、もっと簡単にデータを順序づけられる。

32.1

大きさn のリージョンを見つける

以下のサンプルは『SQL and Its Applications』(Lorie and Daudenarde, 1991)から借用した。次のような劇場の席を管理するテーブルがあるとしよう。

```
CREATE TABLE Theater
(seat_nbr INTEGER NOT NULL PRIMARY KEY, -- 連番
 occupancy_status CHAR(1) DEFAULT 'A' NOT NULL -- 値
     CONSTRAINT valid_occupancy_status
     CHECK (occupancy_status IN ('A', 'S')));
```

Theater：座席テーブル　seat_nbr：座席番号　occupancy_status：座席状態

```
INSERT INTO Theater
VALUES (1, 'A'), (2, 'A'), (3, 'A'), (4, 'S'), (5, 'A'),
       (6, 'A'), (7, 'A'), (8, 'A'), (9, 'S'), (10, 'A');
```

ここでoccupancy_status のコードが'A'（Available）だと利用可能であり、'S'（Sold）だと売約済を意味する。いま我々は、友達と数人で並んで座れる席を取ろうとしている。そのような連続でoccupancy_status が'A'のリージョンをすべて見つけたい。たとえば3 人で劇場に出かけるとするなら、次のような自己結合のクエリが考えられる。

```
SELECT T1.seat_nbr, T2.seat_nbr, T3.seat_nbr
  FROM Theater AS T1, Theater AS T2, Theater AS T3
WHERE T1.occupancy_status = 'A'
  AND T2.occupancy_status = 'A'
  AND T3.occupancy_status = 'A'
  AND T2.seat_nbr = T1.seat_nbr + 1
  AND T3.seat_nbr = T2.seat_nbr + 1;
```

結果

```
seat_nbr seat_nbr seat_nbr
==========================
       1        2        3
       5        6        7
       6        7        8
```

　このやり方の問題点は、人数が3人の場合にしか使えないことだ。もちろんこのパターンをn 人に拡張することはできるのだが、我々が本当に欲しいのは人数をパラメータとして渡すことのできる一般化されたクエリだ。

　ポイントは、与えられた`seat_nbr`から(n−1)個目までの席がすべて利用可能であるかどうかを調べることだ。このときの重要な工夫は、「ここからあそこまでのすべての席が空いている」という表現を「空いているのはここからあそこまでのすべての席である」と表現をひっくり返したことだ。これは全称量化の形なので、SQLで表すことができる。

```
SELECT seat_nbr, ' thru ', (seat_nbr + (:n - 1)) AS end_nbr
  FROM Theater AS T1
 WHERE occupancy_status = 'A'
   AND (seat_nbr + (:n - 1)) <= (SELECT MAX(seat_nbr)
                                   FROM Theater)
   AND 'A' = ALL (SELECT occupancy_status
                    FROM Theater AS T2
                   WHERE T2.seat_nbr > T1.seat_nbr
                     AND T2.seat_nbr <= T1.seat_nbr + (:n - 1));
```

結果

```
seat_nbr  thru  end_nbr
=======================
       1  thru        3
       5  thru        7
       6  thru        8
```

このクエリが返すのは、パラメータnで指定された大きさのリージョンであることに注意してほしい。たとえば、(1, 2, 3, 4, 5)の席が利用可能だったとしても、このクエリは(1, 2, 3)、(2, 3, 4)、(3, 4, 5)という3行を返す。

32.2

リージョンに番号を割り振る

リージョンを探す代わりにリージョンが現れた順番で番号を振っていくとしよう。たとえば、カードの支払い履歴を管理するテーブルを考えよう。支払いが延滞しているかどうかを調べるため、期日どおり支払ったかどうかでリージョンを分けていく。

```
CREATE TABLE PaymentHistory
(payment_nbr INTEGER NOT NULL PRIMARY KEY,
 paid_on_time_flg CHAR(1) DEFAULT 'Y' NOT NULL
     CHECK(paid_on_time_flg IN ('Y', 'N')));
```

PaymentHistory：支払い履歴テーブル　　payment_nbr：支払い番号
paid_on_time_flg：延滞フラグ

```
INSERT INTO PaymentHistory
VALUES (1006, 'Y'), (1005, 'Y'),
       (1004, 'N'),
       (1003, 'Y'), (1002, 'Y'), (1001, 'Y'),
       (1000, 'N');
```

我々が欲しいのは、次のように各リージョンに番号を割り当てた結果だ。

結果

grping	payment_nbr	paid_on_time_flg
1	1006	Y
1	1005	Y
2	1004	N
3	1003	Y

3	1002	Y
3	1001	Y
4	1000	N

フーゴ・コーネリスの考えた解は支払い番号が連番になっているという前提に立っている。

```
SELECT (SELECT COUNT(*)
          FROM PaymentHistory AS H2,
               PaymentHistory AS H3
         WHERE H3.payment_nbr = H2.payment_nbr + 1
           AND H3.paid_on_time_flg <> H2.paid_on_time_flg
           AND H2.payment_nbr >= H1.payment_nbr) + 1 AS grping,
       payment_nbr, paid_on_time_flg
  FROM PaymentHistory AS H1;
```

この解の良いところは、行動パターンが細分化されて、支払いが間に合ったか、遅れたか以外に行動パターンの数が増えたとしても対応可能なことだ。

もう1 つの方法は順序関数を使うことだ。テストの評価をA からD で評価するテーブルを考えよう。ここから評価が同じグループを抽出しようと思う。

```
CREATE TABLE Tests
(sample_time TIMESTAMP NOT NULL PRIMARY KEY,
 sample_score CHAR(1) NOT NULL);
```

Tests：テストテーブル　　sample_time：実施日時　　sample_score：スコア

```
INSERT INTO Tests
VALUES ('2014-01-01 12:00:00', 'A'),
       ('2014-01-01 12:01:00', 'A'),
       ('2014-01-01 12:02:00', 'A'),
       ('2014-01-01 12:03:00', 'B'),
       ('2014-01-01 12:04:00', 'B'),
       ('2014-01-01 12:05:00', 'A'),
       ('2014-01-01 12:06:00', 'A'),
       ('2014-01-01 12:07:00', 'A'),
       ('2014-01-01 12:08:00', 'A'),
       ('2014-01-01 12:09:00', 'B'),
       ('2014-01-01 12:10:00', 'B'),
       ('2014-01-01 12:11:00', 'A'),
       ('2014-01-01 12:12:00', 'A'),
       ('2014-01-01 12:13:00', 'C'),
       ('2014-01-01 12:14:00', 'D'),
       ('2014-01-01 12:15:00', 'A');

SELECT MIN(X.sample_time) AS cluster_start,
       MAX(X.sample_time) AS cluster_end,
       MIN(X.sample_score) AS cluster_score
FROM (SELECT sample_time, sample_score,
             (ROW_NUMBER () OVER (ORDER BY sample_time)
           - ROW_NUMBER() OVER (PARTITION BY sample_score
                       ORDER BY sample_time))
FROM Tests) AS X(sample_time, sample_score, cluster_nbr)
GROUP BY cluster_nbr
ORDER BY 1;
```

結果

```
cluster_start         cluster_end           cluster_score
=============================================================
2014-01-01 12:00:00   2014-01-01 12:02:00   A
2014-01-01 12:03:00   2014-01-01 12:04:00   B
2014-01-01 12:05:00   2014-01-01 12:08:00   A
2014-01-01 12:09:00   2014-01-01 12:10:00   B
2014-01-01 12:11:00   2014-01-01 12:12:00   A
2014-01-01 12:13:00   2014-01-01 12:13:00   C
2014-01-01 12:14:00   2014-01-01 12:14:00   D
2014-01-01 12:15:00   2014-01-01 12:15:00   A
```

こうしたグループはクラスタまたは島と呼ばれている。

32.3

最も大きいリージョンを見つける

先ほどは特定の大きさのリージョンを見つける方法を考えた。今度はひとかたまりの大きなリージョンを見つける方法を考えよう。

```
SELECT T1.seat_nbr AS start_seat_nbr,
       T2.seat_nbr AS end_seat_nbr
  FROM Theater AS T1, Theater AS T2
 WHERE T1.seat_nbr < T2.seat_nbr
   AND NOT EXISTS
        (SELECT *
           FROM Theater AS T3
          WHERE (T3.seat_nbr BETWEEN T1.seat_nbr
                                 AND T2.seat_nbr
            AND T3.occupancy_status <> 'A')
             OR (T3.seat_nbr = T2.seat_nbr + 1
            AND T3.occupancy_status = 'A')
             OR (T3.seat_nbr = T1.seat_nbr - 1
            AND T3.occupancy_status = 'A'));
```

結果

```
start_seat_nbr  end_seat_nbr
============================
             1             3
             5             8
```

この方法のトリックはリージョン内の開始と終了の席を探すことだ。開始の座席番号は売約済みの席より1つ右である。一方終了の席番号は売約済みの席番号より1つ左である。もしこのテーブルが空いている席の番号しか保持していないのなら、コードは少し簡単になる。

またこの問題を一般化するとシーケンシャルだが連続していないデータを持つ任意のテーブルに適用できる。

```
CREATE TABLE AvailableSeating
(seat_nbr INTEGER NOT NULL
     CONSTRAINT valid_seat_nbr
       CHECK (seat_nbr BETWEEN 001 AND 999));
```

AvailableSeating：利用可能座席　　seat_nbr：座席番号

```
INSERT INTO AvailableSeating
VALUES (199), (200), (201), (202), (204),
       (210), (211), (212), (214), (218);
```

　この場合まず欲しいのは、テーブルの一続きのシーケンスの開始と終了の値である。

結果

start_seat_nbr	end_seat_nbr
199	202
204	204
210	212
214	214
218	218

　これは実務上でも売約済みのチケットや未支払いの明細のシーケンスにおいて、失われた値を探すためによく使われる方法だ。テーブルに存在するする数値には● 、存在しない数値には○が割り当てられているような数直線を想像してもらうとわかりやすいだろう。ここからどうやって連続するシーケンスを見つけようか。まず明らかなのは、シーケンスに含まれる数値の数は以下の式で得られる。

終了−開始＋1＝空座席の数

これはつまり、自己結合を使う必要があるということだ。テーブルのコピーを2つ用意し、一方は開始番号をもう一方は終了番号を表す。この2 つの番号が得られれば、上の公式でシーケンスの長さを計算できる。

```
SELECT S1.seat_nbr, MAX(S2.seat_nbr) -- 開始値
  FROM AvailableSeating AS S1
         INNER JOIN
           AvailableSeating AS S2    -- 自己結合
    ON S1.seat_nbr <= S2.seat_nbr
   AND (S2.seat_nbr - S1.seat_nbr + 1) -- 長さの式
         = (SELECT COUNT(*) -- シーケンス内の数
              FROM AvailableSeating AS S3
             WHERE S3.seat_nbr BETWEEN S1.seat_nbr
                                   AND S2.seat_nbr)
               AND NOT EXISTS (SELECT *
                                 FROM AvailableSeating AS S4
                                WHERE S1.seat_nbr - 1
                                        = S4.seat_nbr)
 GROUP BY S1.seat_nbr;
```

最後に終了の番号として選んだ番号が本当にシーケンスの右端に位置するかを保証する必要がある。n個の数値を持つシーケンスは、当然ながら開始番号が同じn個のサブシーケンスを持つ。したがって、開始番号を`GROUP BY`句のキーとして`MAX`関数を使うことで、右端の値を得ることができる。

これをもう少し高速にしたバージョンもある。この解法は、最長のシーケンスが持つもう1つの性質に着目している。いま、シーケンスの右端の値より右には値が存在しない。同様に、開始番号より左にも値は存在しない。いわば、境界を一歩超えたところの失われた値が、シーケンスの範囲を決定していると言ってよい。もちろん、

シーケンスの中にいかなる歯抜けもあってはならない。これをSQLに翻訳すると、次のようになる。

```
SELECT S1.seat_nbr, MIN(S2.seat_nbr) -- 開始値と左端の値
  FROM AvailableSeating AS S1, AvailableSeating AS S2
 WHERE S1.seat_nbr <= S2.seat_nbr
   AND NOT EXISTS -- シーケンスの境界値
         (SELECT * FROM AvailableSeating AS S3
           WHERE S3.seat_nbr NOT BETWEEN S1.seat_nbr
                                     AND S2.seat_nbr
             AND (S3.seat_nbr = S1.seat_nbr - 1
              OR S3.seat_nbr = S2.seat_nbr + 1))
 GROUP BY S1.seat_nbr;
```

シーケンスの右端の値が得られることについては心配する必要はないが、左端の値には注意が必要だ。これを保証するためにMIN 関数を使っている。この2 つ目の方法は、オリジナルのテーブルのコピーを3 つしか使っていないので、最初の解よりも多少速いはずだ。またEXISTS 述語はインデックスを利用できることが多いので、テーブルスキャンが必要なサブクエリ式よりも高速だろう。

順序関数を使うと、自己結合を使わなくても、あるいは明示的な連番の列がなくてもこうした問題を解くことができる。自己結合は高コストなので、順序関数を使うほうがさらに高速だろう。たとえば、次のような1列だけのテーブルを考える。

```
CREATE TABLE Foobar (data_val INTEGER NOT NULL PRIMARY KEY);

INSERT INTO Foobar
VALUES (1), (2), (5), (6), (7), (8), (9), (11), (12), (22);
```

シーケンスを得るためのクエリは次のようになる。

```
WITH X (data_val, data_seq, absent_data_grp)
AS
(SELECT data_val,
        ROW_NUMBER() OVER (ORDER BY data_val ASC),
        (data_val - ROW_NUMBER()
                      OVER (ORDER BY data_val ASC))
   FROM Foobar)
SELECT MIN(data_val) AS start_data_val,
       MIN(data_val) + COUNT(*) -1 AS end_data_val,
       COUNT(*)
  FROM X
 GROUP BY absent_data_grp;
```

結果

```
 start_data_val  end_data_val  count
=====================================
              1             2      2
              5             9      5  ⬅最大
             11            12      2
             22            22      1
```

共通表式は以下のような結果を作る。`absent_data_grp` はテーブルのデータに含まれていない値が何個あるかを示す。

data_val	data_seq	absent_data_grp
1	1	0
2	2	0
5	3	2
6	4	2
7	5	2
8	6	2

9	7	2
11	8	3
12	9	3
22	10	12

　したがって、最大の連続したシーケンスは{5, 6, 7, 8, 9}であることがわかる。

32.4

境界クエリ

　下限と上限の値によって境界が決まっているシーケンスの中で、全体としてどのような傾向があるかを調べる別の形のクエリを考えてみよう。例として株価を時系列に収めたテーブルを考えよう。ここから株価が上昇傾向にある期間を見つけよう。次のようなデータを考える。

```
CREATE TABLE MyStock
(sale_date   DATE PRIMARY KEY,
 stock_price INTEGER);
```

MyStock：株価テーブル　　sale_date：営業日　　stock_price：株価

MyStock

```
sale_date   stock_price
=======================
2011-12-01           10
2011-12-02           15
2011-12-03           13
2011-12-04           12
2011-12-05           20
```

　たとえばこのデータでは、12 月1日から12 月5日の期間で株価は上昇している。つまり、開始よりも終了のほうが株価が上がっているということだ。ただし、途中で下降傾向を見せたこともある。このように開始と終了で株価が上がっていることを調べるクエリは次のようになる。

```
SELECT S1.sale_date AS start_date, S2.sale_date AS finish_date
  FROM MyStock AS S1, MyStock AS S2
 WHERE S1.sale_date < S2.sale_date
   AND NOT EXISTS
       (SELECT *
          FROM MyStock AS S3
         WHERE S3.sale_date BETWEEN S1.sale_date
                                AND S2.sale_date
           AND S3.stock_price NOT BETWEEN S1.stock_price
                                      AND S2.stock_price);
```

結果

```
 start_date  finish_date
=======================
 2011-12-01  2011-12-02
 2011-12-01  2011-12-03
 2011-12-01  2011-12-04
 2011-12-01  2011-12-05
 2011-12-02  2011-12-05
 2011-12-03  2011-12-05
 2011-12-04  2011-12-05
```

32.5

ランを求めるクエリ

　ランとは簡単に言うと歯抜けを許したシーケンスだ。つまり、順序がなんらかの意味を持つ一意な数列のことである。ただし、この数列は必ずしも連続しているとは限らない。このタイプのデータとしては、不定期にサンプリングされる時系列の情報などが挙げられる。ランを求めるクエリはシーケンスを求めるクエリに少し検索条件を変えてやるだけで作ることができる。次のようなテストテーブルを使って考えよう。

```
CREATE TABLE Runs
(list_seq INTEGER NOT NULL PRIMARY KEY,
 list_val INTEGER NOT NULL);
```

Runs

list_seq	list_val
1	6
2	41
3	12
4	51
5	21
6	70
7	79
8	62

9	30
10	31
11	32
12	34
13	35
14	57
15	19
16	84
17	80
18	90
19	63
20	53
21	3
22	59
23	69
24	27
25	33

ランを求めるときの問題の1つは、シーケンスのときと同じでランの大きさnを自由に調節できるようなクエリが望ましい。次のクエリ

は大きさがn以上のランをすべて見つける。もし大きさがnちょうどのランを見つけたいのなら不等号を等号に変えればよい。

```
SELECT R1.list_seq AS start_list_seq,
       R2.list_seq AS end_list_seq_nbr,
       R1.list_val
  FROM Runs AS R1, Runs AS R2
 WHERE R1.list_seq < R2.list_seq -- 開始値と終了値のペア
   AND (R2.list_seq - R1.list_seq) > (:n - 1) -- 長さの制限
   AND NOT EXISTS -- 終了値までの順序
       (SELECT *
          FROM Runs AS R3, Runs AS R4
         WHERE R4.list_seq BETWEEN R1.list_seq AND R2.list_seq
           AND R3.list_seq BETWEEN R1.list_seq AND R2.list_seq
           AND R3.list_seq < R4.list_seq
           AND R3.list_val > R4.list_val);
```

このクエリがやっていることは、R1 で開始点の番号を作りR2で終了点の番号を作っている。NOT EXISTS 述語の中の巨大なサブクエリはランの中で順序に違反する並びの行がないかを探している。もしそのような行がなければランは妥当なものとして認められる。このクエリが何をやっているかを理解するには、数直線を書いてみるのが一番わかりやすい。ちなみにシーケンスは期間内のすべての数値が1ずつ増加するようなランのことだった。これはつまり、シーケンスにおいては最大値から最小値をひいて1を足すとシーケンスの大きさになるということだ。この性質をクエリに追加してやれば、逆にシーケンスを求めるクエリができる。

```
SELECT R1.list_seq AS start_list_seq,
       R2.list_seq AS end_list_seq_nbr
  FROM Runs AS R1, Runs AS R2
 WHERE R1.list_seq < R2.list_seq
   AND (R2.list_seq - R1.list_seq) = (R2.list_val - R1.list_val) -- 順序の条件
   AND (R2.list_seq - R1.list_seq) > (:n - 1) -- 長さの制限
   AND NOT EXISTS
         (SELECT *
            FROM Runs AS R3
           WHERE R3.list_seq BETWEEN R1.list_seq AND R2.list_seq
             AND ((R3.list_seq - R1.list_seq)
                    <> (R3.list_val - R1.list_val)
              OR (R2.list_seq - R3.list_seq)
                    <> (R2.list_val - R3.list_val)));
```

NOT EXISTS 述語のサブクエリは開始と終了の間で順序の条件に違反する点が1 つもないことを保証している。

いまは開始と終了の間で増加しているか、減少しているかだけを見ているが、比較の条件を少し変えてやることで、単調増加または単調減少しているものだけを選択するクエリにすることもできる。また、増分を1から適当な値kに変えることも可能だ。

32.5.1　歯抜けの穴埋め

数列をキーに使っている場合、歯抜けになった値を再利用したいと思うプログラマは多いだろう。原則として、著者はこのような無意味かつ検証不可能な値をキーに使うことは推奨しないが、失われた数値を穴埋めするという問題は、練習としては純粋に面白い。一般的な前提条件は次のとおりだ。

1. シーケンスは1から始まる。もし1が失われている場合テーブルは空である。
2. 最初に再利用するのは失われている値の最小値である。

3. シーケンスには上限値が決められており、これを超過してはいけない。もしシーケンスが最後まで使われている場合は、警告またはNULLを返す。あるいは別の選択肢として最大値＋1を返すという方法も考えられる。

本章の最初に定義を示したListテーブルに次のようなサンプルデータを入れよう。次の回答は物事を1行ずつ処理するのではなく、集合の観点から考えた好例だ。

```
DELETE FROM List;
INSERT INTO List
VALUES (1, 99), (2, 10), (3, 11), (5, 12), (7, 13), (8, 14), (9, 0);

SELECT MIN(CASE WHEN list_seq + 1 NOT IN (SELECT list_seq FROM List)
                 THEN list_seq + 1
                WHEN 1 NOT IN (SELECT list_seq FROM List)
                 THEN 1 ELSE NULL END) AS min_gap
  FROM List
 WHERE list_seq BETWEEN (SELECT MIN(list_seq)
                           FROM List)
                    AND (SELECT MAX(list_seq)
                           FROM List);
```

結果

```
 min_gap
========
      4
```

考え方は失われた値のテーブル式を作り、そこから最小値を選択している。シーケンスの開始である1 は例外として扱っている。シーケンスに歯抜けがない場合は、最大値＋1 を返す。CASE式の中に集約関数を入れた次のような書き方もできる。

```
SELECT CASE WHEN MAX(list_seq) = COUNT(*)
              THEN MAX(list_seq) + 1
            WHEN MIN(list_seq) > 1
              THEN 1
            WHEN MAX(list_seq) <> COUNT(*)
              THEN (SELECT MIN(list_seq) + 1
                      FROM List
                     WHERE (list_seq) + 1
                           NOT IN (SELECT list_seq
                                     FROM List))
            ELSE NULL END
  FROM List;
```

最初のWHEN句はシーケンスに歯抜けがないかをチェックしている。歯抜けがない場合は、最大値＋1を返す。2番目のWHEN句はシーケンスの最小値が1かどうかをチェックしている。もし1でなければ1を返す。3番目のWHEN句はシーケンスの中間に歯抜けがあるかどうかをチェックしている。もし歯抜けがあれば、最小値を返す。

ELSE句はなんらかのエラーが起きたときの場合に使われるが、普通ここには到達しないはずである。ここで重要なのは、WHEN 句の実行順序である。WHEN 句は短絡評価が行われるので、TRUEになったWHEN句以降のWHEN句は評価されない。したがって、複雑なサブクエリ式を一番最後のWHEN 句に持ってくるのがパフォーマンス上効率的だ。これはちょっとした高速化のテクニックである。

集合の特性を利用したより簡単な解法は以下のようになる。

```
SELECT COALESCE(MIN(L1.list_seq) + 1, 1)
  FROM List AS L1
         LEFT OUTER JOIN
           List AS L2
    ON L1.list_seq = L2.list_seq - 1
 WHERE L2.list_seq IS NULL;
```

ただしこのクエリは、(3, 4, 5)のようにシーケンスの「前半部」が欠けている場合には1 ではなく、最大値＋1を返す。こういうケースに対応するには、次のように記述する。

```
SELECT MIN(list_seq + 1)
  FROM (SELECT list_seq FROM List
        UNION ALL
        SELECT *
          FROM (VALUES (0)) AS A) AS X(list_seq)
 WHERE (list_seq +1)
          NOT IN (SELECT list_seq FROM List);
```

歯抜けのシーケンスをすべて見つけるならば次のコードが使える。ただしやはり、「前半部」が欠けている場合には歯抜けがないとみなす。

```
DELETE FROM List;
INSERT INTO List
VALUES (3, 99), (7, 10), (8, 11), (12, 12);

SELECT (s + 1) AS gap_start,
       (e - 1) AS gap_end
  FROM (SELECT L1.list_seq, MIN(L2.list_seq)
          FROM List AS L1, List AS L2
         WHERE L1.list_seq < L2.list_seq
         GROUP BY L1.list_seq) AS G(s, e)
 WHERE (e - 1) > s;
```

結果

```
gap_start  gap_end
==================
        4        6
        9       11
```

導出テーブルをなくせば次のようになる。

```
SELECT (L1.list_seq + 1) AS gap_start,
       (MIN(L2.list_seq) - 1) AS gap_end
  FROM List AS L1, List AS L2
 WHERE L1.list_seq < L2.list_seq
 GROUP BY L1.list_seq
HAVING (MIN(L2.list_seq) - L1.list_seq) > 1;
```

32.6

数列の歯抜け

数列の歯抜けを見つける問題をもう少し考えよう。歯抜け数列や日付列には、歯抜けが存在することがある。たとえば、集合{1, 6, 7, 8, 9} には{2, 3, 4, 5}という要素が欠けている。特別な事情がない限り、歯抜けを探す範囲は最小値と最大値の間であり、したがってこの数列は{10, 11, 12}という歯抜けを持つわけではないと考える。

いま、チケット購入者のテーブルがあるとしよう。購入されたチケットは連番であるとする。ここから、購入者別にチケット番号の歯抜けを見つけたい。

```
CREATE TABLE Tickets
(buyer_id CHAR(5) NOT NULL,
 ticket_nbr INTEGER DEFAULT 1 NOT NULL
     CHECK (ticket_nbr > 0),
     PRIMARY KEY (buyer_id, ticket_nbr));
```

Tickets：チケットテーブル　　buyer_id：購入者ID　　ticket_nbr：チケット番号

```
INSERT INTO Tickets
VALUES ('a', 2), ('a', 3), ('a', 4),
       ('b', 4),
       ('c', 1), ('c', 2), ('c', 3), ('c', 4), ('c', 5),
       ('d', 1), ('d', 6), ('d', 7), ('d', 9),
       ('e', 10);
```

もしテーブルの行数が比較的小さいなら、1 からn までの連番を持つ普通の数列テーブルを使って以下のように書ける。

```
SELECT DISTINCT T1.buyer_id, S1.seq
  FROM Tickets AS T1, Series AS S1
 WHERE seq <= (SELECT MAX(ticket_nbr) -- 範囲を設定する
                 FROM Tickets AS T2
                WHERE T1.buyer_id = T2.buyer_id)
   AND seq NOT IN (SELECT ticket_nbr -- 失われた番号を得る
                     FROM Tickets AS T3
                    WHERE T1.buyer_id = T3.buyer_id);
```

結果

```
 buyer_id seq
=============
 a          1
 b          1
 b          2
 b          3
 d          2
 d          3
 d          4
 d          5
 d          8
 e          1
 e          2
 e          3
 e          4
 e          5
 e          6
 e          7
 e          8
 e          9
```

これはつまるところ、歯抜けとは「テーブルに存在しない数列だ」と言っているに等しい。

連続した歯抜けの性質として、開始値が1ではなく、かつ終了値＋1の値が両方のテーブルに存在する。さらに、開始値と終了値の間にはテーブルに値が存在しない。この性質を利用することで、どこからどこまでが歯抜けか、境界を確定することができる。

```
SELECT T1.buyer_id,
       (T1.ticket_nbr + 1) AS gap_start,
       (MIN(T2.ticket_nbr) - 1) AS gap_end
  FROM (TABLE Tickets
        UNION ALL
        SELECT DISTINCT buyer_id, 0
          FROM Tickets) AS T1(buyer_id, ticket_nbr),
       Tickets AS T2
 WHERE T1.ticket_nbr < T2.ticket_nbr
   AND T1.buyer_id = T2.buyer_id
 GROUP BY T1.buyer_id, T1.ticket_nbr
HAVING MIN(T2.ticket_nbr) - T1.ticket_nbr > 1;
```

結果

```
 buyer_id gap_start gap_end
============================
 a                1       1
 b                1       3
 d                2       5
 d                8       8
 e                1       9
```

ここでのトリックは、ナンバリングを1から開始することだ。それによって、1が集合から失われている場合には、境界値から0 は歯抜けとみなさない。もし0 を付け加えたければ、UNION ALLで付加すればいい。

直観的に、これの「逆」問題、つまり存在するすべての数の範囲をリストするのは、失われた数の範囲を求めるより簡単だ。

```
SELECT X.buyer_id, MIN(X.initial), X.final
  FROM (SELECT T1.buyer_id, T1.ticket_nbr AS initial,
               MAX(T2.ticket_nbr) AS final
          FROM Tickets AS T1, Tickets AS T2
         WHERE T1.ticket_nbr <= T2.ticket_nbr
           AND (SELECT COUNT(DISTINCT T3.ticket_nbr)
                  FROM Tickets AS T3
                 WHERE T1.buyer_id = T3.buyer_id
                   AND T3.ticket_nbr
                          BETWEEN T1.ticket_nbr AND T2.ticket_nbr)
               = (T2.ticket_nbr - T1.ticket_nbr +1)
         GROUP BY T1.buyer_id, T1.ticket_nbr) AS X(buyer_id, initial, final)
 GROUP BY buyer_id, final;
```

結果

```
 buyer_id min final
==================
 a          2     4
 b          4     4
 c          1     5
 d          1     1
 d          6     7
 d          9     9
 e         10    10
```

32.7

数列の合計

順序関数が存在しなかった頃、テーブルの値を累計するのは難しいことだった。連番と値の2列だけを持つ`List`という簡単なテーブルを考えよう。累計を求めるクエリは、いまではウィンドウ関数を使って次のように書ける。

```
SELECT list_seq, list_val,
         SUM (list_val)
           OVER (ORDER BY list_seq
                 ROWS BETWEEN UNBOUNDED PRECEDING
                          AND CURRENT ROW) AS running_tot
  FROM List;
```

結果はたとえば、次のようになる。

list_seq	list_val	running_tot
1	6	6
2	41	47
3	12	59
4	51	110
5	21	131
6	70	201
7	79	280
8	62	342
…		

このタイプの1段階の合計が、ほとんどの問題に使えるパターンだろう。だがより複雑な問題もある。次に示すのはフランシスコ・モレノからの問題だ。これは見た目には簡単に見える。まず、次のようなテーブルを作りデータを投入する。

```
CREATE TABLE Series
(list_seq INTEGER NOT NULL,
 list_val REAL NOT NULL,
 running_avg REAL);

INSERT INTO Series
VALUES (0, 6.0, NULL),
       (1, 6.0, NULL),
       (2, 10.0, NULL),
       (3, 12.0, NULL),
       (4, 14.0, NULL);
```

やりたいことはこうだ。まず最初の2 項について平均を求める。次にその平均と3 番目の`list_val`の値の平均を求める。以下同様に常に2つの項に対して平均を求めていく。これは次のような累積平均とは同じではない。これは平均の計算対象となる項が増えていくからだ。

```
SELECT list_seq, list_val,
       AVG(list_val)
         OVER (ORDER BY list_seq
               ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW) AS running_avg
  FROM Series;
```

結果

```
 list_seq  list_val  running_avg
================================
        0         6            6
        1         6            6
        2        10          7.3
        3        12          8.5
        4        14          9.6
```

　このデータの場合、我々が欲しい答えは以下のようになる。`list_seq=0`の行については、2 項揃っていないので`NULL`を返す点に注意してほしい。

list_seq	list_val	running_avg
0	6.0	NULL
1	6.0	6.0
2	10.0	8.0
3	12.0	10.0
4	14.0	12.0

　まず単純に考えると思いつく答えは、次のように直接計算を行うものだろう。

```
UPDATE Series
   SET running_avg = (Series.list_val
                      + (SELECT S1.list_val
                           FROM Series AS S1
                          WHERE S1.list_seq = Series.list_seq - 1)) / 2.0
 WHERE running_avg IS NULL;
```

結果

```
list_seq list_val running_avg
============================
       0        6        NULL
       1        6           6
       2       10           8
       3       12          11
       4       14          13
```

だがこの方法は間違いだ。なぜなら、これは結局のところ、`list_val`の平均になってしまうが、求めたいのは`list_val`と`running_avg` の平均だからだ。いわば、ここで我々がやろうとしていることは、2 段階の集計である。これは連続的な分数によって定義される。最初の4 項について愚直に計算の実行ステップを見て、どのようなパターンになるか観察しよう。

```
answer1 = (12)/2 = 6
answer2 = ((12)/2 + 10)/2 = 8
answer3 = (((12)/2 + 10)/2 + 12)/2 = 10
answer4 = (((((12)/2 + 10)/2 + 12)/2) + 14)/2 = 12
```

ここでのポイントはちょっとした式変形を行って、入れ子になった`()`を取り除くことだ。

```
answer1 = (12)/2 = 6
answer2 = (12/4) + (10/2) = 8
answer3 = (12/8) + (10/4) + (12/2) = 10
answer4 = (12/16) + (10/8) + (12/4) + (14/2) = 12
```

こうすると、分母を2の累乗で表すことができることに気がつく。そこで、さらに以下のように変換する。

```
answer1 = (12)/2^1 = 6
answer2 = (12/(2^2)) + (10/(2^1)) = 8
answer3 = (12/(2^3)) + (10/(2^2)) + (12/(2^1)) = 10
answer4 = (12/2^4) + (10/(2^3)) + (12/(2^2)) + (14/(2^1)) = 12
```

問題は前項の合計に対して、より高次の累乗を計算するために現在の`list_val`から後ろへ戻らなければいけないことだ。これはシンプルに`(current_list_val - previous_list_val + 1)`という式で表される。これらのことをすべてまとめると、次のような`UPDATE`文になる。

```
UPDATE Series
   SET running_avg
         = (SELECT SUM(list_val
                       / POWER(2, CASE WHEN S1.list_seq > 0
                                       THEN Series.list_seq - S1.list_seq + 1
                                       ELSE NULL END))
              FROM Series AS S1
             WHERE S1.list_seq <= Series.list_seq);
```

32.8

リストの値を入れ替えたり、移動させる

値のリストを操作し、シーケンスにおける値の位置を変えたいと思うことは多いだろう。そうした操作のうち最も単純なのは、2つの値を入れ替えることだ。

```
CREATE PROCEDURE SwapValues
(low_list_seq IN INTEGER, high_list_seq IN INTEGER)
IS
    least_list_seq INTEGER;
    greatest_list_seq INTEGER;
BEGIN
    least_list_seq := CASE WHEN low_list_seq <= high_list_seq
                           THEN low_list_seq
                           ELSE high_list_seq END;
    greatest_list_seq := CASE WHEN low_list_seq <= high_list_seq
                              THEN high_list_seq
                              ELSE low_list_seq END;
UPDATE Runs -- 入れ替え
   SET list_seq = least_list_seq
                  + ABS(list_seq - greatest_list_seq)
 WHERE list_seq IN (least_list_seq, greatest_list_seq);
END;
```

CASE式はUPDATE文の中に入れ込むこともできるが、コードが読みにくくなる。

新しい値をテーブルに挿入するのは簡単だ。

```
INSERT INTO Runs (list_seq, list_val)
VALUES ((SELECT MAX(list_seq) FROM Runs) + 1, :new_value);
```

ある値を新しい位置に動かして残りの値をごそっとずらすのは、少しトリッキーだ。これはあたかも物理的なキューの動きを真似するような形になる。デイブ・ポータスが考えた答えは次のようなものだ。

```
CREATE OR REPLACE PROCEDURE SlideValues
(old_list_seq IN INTEGER, new_list_seq IN INTEGER)
IS
BEGIN
    UPDATE Runs
       SET list_seq =
              CASE WHEN list_seq = old_list_seq
                     THEN new_list_seq
                   WHEN list_seq BETWEEN old_list_seq
                                     AND new_list_seq
                     THEN list_seq - 1
                   WHEN list_seq BETWEEN new_list_seq
                                     AND old_list_seq
                     THEN list_seq + 1
                   ELSE list_seq END
     WHERE list_seq BETWEEN old_list_seq AND new_list_seq
        OR list_seq BETWEEN new_list_seq AND old_list_seq;
END;
```

これは対象の値を高いほうにも低いほうにも動かすことができる。こうした操作を繰り返したり、少し変えることで他の操作も行うことができる。

このトリックの有用な応用例としては、ユリウス通日の列を持ったカレンダーテーブルが考えられる【訳注1】。時間データを操作する代わりに、日付を整数のシーケンスに変換し、リージョンやランを扱うクエリを利用すればよい。

32.9

シーケンスの始点と終点を見つける

　やりたいことは数のリストをとり、それを連続的な範囲に詰めることだ。つまり、シーケンスの最小値と最大値を示したい。もし最小値と最大値が同じ値なら、その範囲には1つの数しか存在しない。以下の答えはスティーブ・キャスによるものだ。

```
DELETE FROM List;
INSERT INTO List
VALUES (1, 1), (2, 2), (3, 3), (6, 6), (7, 7), (10, 10);

SELECT MIN(i) AS low, MAX(i) AS high
  FROM (SELECT L1.list_seq,
               COUNT(L2.list_seq) - L1.list_seq
          FROM List AS L1, List AS L2
         WHERE L2.list_seq <= L1.list_seq
         GROUP BY L1.list_seq) AS N(i, gp)
 GROUP BY gp;
```

結果

```
 low | high
===========
   1 |    3
   6 |    7
  10 |   10
```

32.10

数のリストを折り返して表示する

数列テーブルは、自己結合の代わりに、互いに関連付けられた値を与える列を与える用途に使える。たとえば、1 からn までの数がある場合、これをk 個の列に広げたいとしよう。k=3とした場合、どのようなパターンになるか見てみよう。

```
DELETE FROM List;
INSERT INTO List
VALUES (1, 1), (2, 2), (3, 3), (4, 4), (5, 5),
       (6, 6), (7, 7), (8, 8), (9, 9), (10, 10);

SELECT list_seq,
       CASE WHEN MOD((list_seq + 1), 3) = 2
             AND list_seq + 1 <= :n
            THEN (list_seq + 1)
            ELSE NULL END AS second,
       CASE WHEN MOD((list_seq + 2), 3) = 0
             AND (list_seq + 2) <= :n
            THEN (list_seq + 2)
            ELSE NULL END AS third
  FROM List
 WHERE MOD((list_seq), 3) = 1
   AND list_seq <= :n;
```

結果

```
 list_seq second third
=======================
        1      2     3
        4      5     6
        7      8     9
       10
```

値が割り当てられなかった列はNULL になる。たとえば、n=10 の場合、不完全な行は(10, NULL, NULL)になるし、n=7 の場合は、(7, NULL, NULL)になる。(NULL, NULL, NULL)のような行が生じることはない。

実際の応用例として、ゴルフトーナメントの組み合わせを考えよう。まずスコアが最高の人と最低の人を次のラウンドで同じ組にする。次に2 番目に高い人と低い人を同じ組にする。参加者の数が奇数だった場合は、真ん中のスコアの人は次のラウンドを欠場する。このようなペアは次のクエリで簡単に作れる。

```
SELECT list_seq AS low_score,
       CASE WHEN list_seq <= (:n - list_seq)
            THEN (:n - list_seq) + 1
            ELSE NULL END AS high_score
  FROM List
 WHERE list_seq
         <= CASE WHEN MOD(:n, 2) = 1
                 THEN FLOOR(:n / 2) + 1
                 ELSE (:n / 2) END;
```

結果

```
low_score high_score
====================
        1          10
        2           9
        3           8
        4           7
        5           6
```

基本的な数学関数を使うと、いろいろなことができることがわかるだろう。

32.11

オーバーラップする被覆

ヴァディム・トロパシュコは、数直線上に重なり合う複数の期間が与えられた場合、それらをすべてカバーする最小の期間を返すクエリは何か、という問題を提起した。それも、なるべく短いクエリ、という条件付きだ。たとえば、xとyで定義される期間をカバーする最小の期間は何だろうか？

この問題は主に時間の範囲を扱う際に現れる。だが、問題の骨子は整数の数列で考えたほうがわかりやすい。たとえば次のようなテーブルがあり、範囲が指定されたとする。

```
CREATE TABLE Intervals
(x INTEGER NOT NULL,
 y INTEGER NOT NULL,
     CHECK (x <= y),
     PRIMARY KEY (x, y));
```

Intervals：期間テーブル　　x：始点　　y：終点

```
INSERT INTO Intervals
VALUES (1, 3),(2, 5),(4, 11),
       (10, 12), (20, 21),
       (120, 130), (120, 128), (120, 122),
       (121, 132), (121, 122), (121, 124), (121, 123),
       (126, 127);
```

ここから、次のような結果を得たい。

min_x	max_y
1	12
20	21

120	132

ディーター・ネースは、ウィンドウ関数を使った次のような解を考えた。

```
SELECT min_x, MAX(y)
  FROM (SELECT x, y,
               MAX(CASE WHEN x <= MAX_Y THEN NULL ELSE x END)
                 OVER (ORDER BY x, y
                       ROWS UNBOUNDED PRECEDING) AS min_x
          FROM (SELECT x, y,
                       MAX(y)
                         OVER(ORDER BY x, y
                              ROWS BETWEEN UNBOUNDED PRECEDING
                                       AND 1 PRECEDING) AS max_y
                  FROM Intervals) AS DT) AS DT
 GROUP BY min_x;
```

同じアプローチで、自己結合と3レベルの相関サブクエリを使うクエリは次のようになる。

```
SELECT I1.x, MAX(I2.y) AS y
  FROM Intervals AS I1
         INNER JOIN
           Intervals AS I2
    ON I2.y > I1.x
 WHERE NOT EXISTS
        (SELECT *
           FROM Intervals AS I3
          WHERE I1.x - 1 BETWEEN I3.x AND I3.y)
            AND NOT EXISTS
                 (SELECT *
                    FROM Intervals AS I4
                   WHERE I4.y > I1.x
                     AND I4.y < I2.y
                     AND NOT EXISTS
                          (SELECT *
                             FROM Intervals AS I5
                            WHERE I4.y + 1 BETWEEN I5.x AND I5.y))
 GROUP BY I1.x;
```

以下は本質的には同じフォーマットだが、サブクエリの代わりに左外部半結合を使っている。これは簡潔とは言いがたいが、プラットフォームによってはこちらのが速い可能性もある。

```
SELECT I1.x, MAX(I2.y) AS y
  FROM Intervals AS I1
         INNER JOIN
           Intervals AS I2
             ON I2.y > I1.x
               LEFT OUTER JOIN
                 Intervals AS I3
                   ON I1.x - 1 BETWEEN I3.x AND I3.y
                     LEFT OUTER JOIN
                       (Intervals AS I4
                          LEFT OUTER JOIN
                            Intervals AS I5
                              ON I4.y + 1 BETWEEN I5.x AND I5.y)
                        ON I4.y > I1.x
                       AND I4.y < I2.y
                       AND I5.x IS NULL
 WHERE I3.x IS NULL
   AND I4.x IS NULL
 GROUP BY I1.x;
```

もしテーブルが大きい場合、相関サブクエリや5回の自己結合は遅いだろう。しかし、速いのではなく短いクエリという要望だったので、どちらも候補になる。トニー・アンドリュースは、次のような解を思いついた。

```
SELECT Starts.x, Ends.y
  FROM (SELECT x, ROW_NUMBER() OVER(ORDER BY x) AS rn
          FROM (SELECT x, y,
                       LAG(y) OVER(ORDER BY x) AS prev_y
                  FROM Intervals) AS Foo
                 WHERE prev_y IS NULL
                    OR prev_y < x) AS Starts,
       (SELECT y, ROW_NUMBER() OVER(ORDER BY y) AS rn
          FROM (SELECT x, y,
                       LEAD(x) OVER(ORDER BY y) AS next_x
                  FROM Intervals) AS Boo
         WHERE next_x IS NULL
            OR y < next_x) AS Ends
 WHERE Starts.rn = Ends.rn;
```

ジョン・ギルソンは、なるべく短いほうがいいという条件に応えるには、再帰を使うのがよいと考えた。彼は整数ではなく時刻データを使った。

```
CREATE TABLE Sessions
(user_name VARCHAR(10) NOT NULL,
 start_timestamp TIMESTAMP(0) NOT NULL,
     PRIMARY KEY (user_name, start_timestamp),
 end_timestamp TIMESTAMP(0) NOT NULL,
     CONSTRAINT End_GE_Start
       CHECK (start_timestamp <= end_timestamp));
```

Sessions：セッションテーブル　user_name：ユーザー名　start_timestamp：開始時刻　end_timestamp：終了時刻

```
INSERT INTO Sessions
VALUES
('User1', '2011-12-01 08:00:00', '2011-12-01 08:30:00'),
('User1', '2011-12-01 08:30:00', '2011-12-01 09:00:00'),
('User1', '2011-12-01 09:00:00', '2011-12-01 09:30:00'),
('User1', '2011-12-01 10:00:00', '2011-12-01 11:00:00'),
('User1', '2011-12-01 10:30:00', '2011-12-01 12:00:00'),
('User1', '2011-12-01 11:30:00', '2011-12-01 12:30:00'),
('User2', '2011-12-01 08:00:00', '2011-12-01 10:30:00'),
('User2', '2011-12-01 08:30:00', '2011-12-01 10:00:00'),
('User2', '2011-12-01 09:00:00', '2011-12-01 09:30:00'),
('User2', '2011-12-01 11:00:00', '2011-12-01 11:30:00'),
('User2', '2011-12-01 11:32:00', '2011-12-01 12:00:00'),
('User2', '2011-12-01 12:04:00', '2011-12-01 12:30:00'),
('User3', '2011-12-01 08:00:00', '2011-12-01 09:10:00'),
('User3', '2011-12-01 08:15:00', '2011-12-01 08:30:00'),
('User3', '2011-12-01 08:30:00', '2011-12-01 09:00:00'),
('User3', '2011-12-01 09:30:00', '2011-12-01 09:30:00);

WITH RECURSIVE Sorted_Sessions
AS
(SELECT user_name, start_timestamp, end_timestamp,
        ROW_NUMBER()
        OVER(PARTITION BY user_name
                 ORDER BY start_timestamp, end_timestamp)
        AS session_seq
   FROM Sessions),
Sessions_Groups
AS
(SELECT user_name, start_timestamp, end_timestamp,
        session_seq, 0 AS grp_nbr
   FROM Sorted_Sessions
 WHERE session_seq = 1
 UNION ALL
 SELECT S2.user_name, S2.start_timestamp,
```

```
        CASE WHEN S2.end_timestamp > S1.end_timestamp
        THEN S2.end_timestamp ELSE S1.end_timestamp END,
        S2.session_seq,
        S1.grp_nbr +
          CASE WHEN S1.end_timestamp
                  < COALESCE (S2.start_timestamp,
                        CAST('9999-12-31' AS TIMESTAMP(0)))
              THEN 1 ELSE 0 END AS grp_nbr
   FROM Sessions_Groups AS S1
 INNER JOIN
 Sorted_Sessions AS S2
 ON S1.user_name = S2.user_name
    AND S2.session_seq = S1.session_seq + 1)
SELECT user_name,
       MIN(start_timestamp) AS start_timestamp,
       MAX(end_timestamp) AS end_timestamp
 FROM Sessions_Groups
GROUP BY user_name, grp_nbr;
```

最後に、以下のアプローチを見てみよう。通常の数列テーブルがあるとする。期間の範囲内に存在する欠番をすべて見つけ、それをビューまたはWITH句に保持する。

```
CREATE VIEW Holes (hole)
AS
SELECT seq
  FROM Series
 WHERE seq <= (SELECT MAX(y) FROM Intervals)
   AND NOT EXISTS
        (SELECT *
           FROM Intervals
          WHERE Series.seq BETWEEN x AND y)
UNION (SELECT list_seq FROM (VALUES (0)) AS L(list_seq)
       UNION
       SELECT MAX(y) + 1 FROM Intervals AS R(list_seq));
```

以下のクエリは、欠番の開始と終了の数値を抽出し、範囲内の欠番の長さを数える。被覆は、もちろん内部に1つの欠番も持たない。

```
SELECT Starts.x, Ends.y
  FROM Intervals AS Starts,
       Intervals AS Ends,
       Series AS S -- 通常の数列テーブル
 WHERE S.seq BETWEEN Starts.x AND Ends.y -- seq列の制限
   AND S.seq < (SELECT MAX(hole) FROM Holes)
   AND S.seq NOT IN (SELECT hole FROM Holes) -- 欠番ではない
   AND Starts.x - 1 IN (SELECT hole FROM Holes)  -- 左端
   AND Ends.y + 1 IN (SELECT hole FROM Holes)    -- 右端
 GROUP BY Starts.x, Ends.y
HAVING COUNT(DISTINCT seq) = Ends.y - Starts.x + 1;
```

【訳注1】　ユリウス通日は紀元前4713 年1月1日を起点として通し番号で日付を表す歴法。数年をまたぐ2つの日数を計算するのに便利。

33

SQLにおける配列

Matrices in SQL

配列は、行列（マトリクス）とは違う。配列はその要素に添え字（インデックス）と呼ばれる数字でアクセスするデータ構造である。行列は、数学的演算が定義された配列である。行列は、1次元、2次元、3次元、あるいはそれ以上の次元の構造になることがある。

SQLは、配列も行列も持っていない。なぜならリレーショナルモデルにおける唯一のデータ構造はテーブルだからだ。配列は「すべての列がスカラ値であるべし」という第1正規形のルールに違反するため、リレーショナルデータベースにおいて存在を許されない。

だが、このルールは不幸なことにSQL:2003において変更された。コレクション型が追加されたのだ。現在では、SQL の1 つの列のデータ型として配列を宣言することが可能だ。配列は1 次元であり、何の数学的演算も定義されていない。配列構築子と比較演算があるのみだ。構築子は、カンマ区切りのリストまたは`ORDER BY` 句付きのクエリのどちらかである。多次元配列の列を結果セットとして表示したりホスト言語に受け渡す明確な方法は存在しない。言語とコンパイラによって列方向と行方向のどちらの順序を優先してソートするかは異なり、標準の取り決めがないのだ。

非スカラ値に対する制約を書く明確な手段もない。SQLの目的は幅広い種類のホスト言語と親和性の高いデータベース言語であること

だ。この目的を達成するために、SQLはホスト言語のデータ型に対応できるよう、非常に豊富な、かつ可能な限りシンプルなスカラデータ型を用意してきた。それもこれもホスト言語へのデータの受け渡しをスムーズに行いたいがためだ。だが、SQL-99以降の拡張は、このSQLの利点を台無しにした。

33.1

名前を持った列による擬似配列

SQL 以外のプログラミング言語において、配列は名前と添え字を持ち、これによって要素にアクセスできる。配列の要素はすべて同じデータ型で、添え字はシーケンシャルな整数である。C言語などは添え字を0から始めるが、1から始める言語もある。上限と下限の値をユーザーが定義できる言語もある。たとえば、Pascalの配列宣言は、次のような形になる。

```
foobar : ARRAY [1..5] OF INTEGER;
```

これは、`foobar[1]`、`foobar[2]`、`foobar[3]`、`foobar[4]`、`foobar[5]`という整数の要素を持つ。これと同じ構造をSQLで表現しようとすると、次のような宣言になるだろう。

```
CREATE TABLE Foobar1
(element1 INTEGER NOT NULL,
 element2 INTEGER NOT NULL,
 element3 INTEGER NOT NULL,
 element4 INTEGER NOT NULL,
 element5 INTEGER NOT NULL);
```

このテーブルにおいては、配列のように添え字で各要素にアクセスすることはできない。たとば、Pascalの配列で各要素を0に初期化しようとすれば、`FOR`ループを使って次のように書く。

```
FOR i := 1 TO 5 DO foobar[i] := 0;
```

SQLで同じことをやろうとするには、次のようになる。

```
UPDATE Foobar1
   SET element1 = 0,
       element2 = 0,
       element3 = 0,
       element4 = 0,
       element5 = 0;
```

SQLには添え字はないので、ループ内で使うことはできない。SQLにおけるあらゆるアクセスは、列名経由で行う必要があり、添え字の出る幕はない。SQLで無理やり「擬似添え字」を使おうとすると（つまり、列名に添え字を埋め込むと）、動的SQLで列名を生成するコードが必要になるが、これは遅いだけでなく危険である。さらに向こう見ずなユーザーは、テーブル名にも同じやり方を適用して論理データモデルを破壊している。

たとえば、4つの独立した子どもの列を持つ社員テーブルを設計すると考えよう。まずはテーブルが空の状態から始める。

1. 子どもの数が4 人以下の社員を雇った場合は何が起きるだろう？ 即座にクビにするか、それともももっとたくさん子どもを作れとせっつくか。テーブルが4 人以下の子どもも受け入れるよう再構成することもできる。通常取られる解決策は、子ども列にNULLを許可することだ。これによってNULLにまつわる諸問題を背負い込むことになる。
2. 子どもが5人いる社員を雇った場合は何が起きるだろう？ 即座にクビにするか、それとも子どもを1人殺すよう命令するか。テーブルが5人の子どもを持てるよう再構成することもできる。5～8 人目の子どもの情報は2 行目に持つという方法もありうる。だがこれは、社員IDの一意性を破壊することになり、主キ

ーとして使えなくなる。連番列を追加することで、この問題には対処できるが、そもそも「社員」テーブルなのに社員で一意にならないのは、モデリング上問題がある。

3. 社員が死んだ場合は何が起きるだろう？ 会社としては残された子どもに何らかの補償をしてあげたいのに、社員とともに子どものデータもすべて削除することになる。これに対処するには、物理削除をせず、「死亡フラグ」を立てるなど論理削除に切り換えるという方法がある。だがこれは、あまりきれいなやり方ではない。
4. 社員の子どもが死んだ場合は何が起きるだろう？ 彼をクビにするか、それとも至急もう1人代わりの子どもを作れと命令するか。テーブルが4人以下の子どもも受け入れるよう再構成することもできる。通常取られる解決策は子ども列にNULLを許可することだ。これによってNULLにまつわる諸問題を背負い込むことになる。だが、おそらくそれ以外に方法はあるまい。もしこのとき、2.の複数行の案を採用していたらやっかいだ。というのも、5人の子どもを持つ社員の5人目の子どもが死んだ場合、完全に空の行が残されるからだ。この行を削除するべきだろうか？
5. 社員の子どもが1人死んで、新しい子どもが生まれた場合は何が起きるだろう？ 新しい子どものデータで、死んだ子どもの列のNULLを上書きするべきだろうか？ それとも新しい空きの列のNULLを上書きするべきだろうか？

上に挙げた選択肢の中には、テーブルの再構築を必要とする案もあれば、データベースに合うよう現実を再構成しようとする馬鹿げた案もある。この問題の本質は、子どもを挿入したり削除したりするという同じ操作が、社員の持つ子どもの数によって異なるプロシージャになることだ。ファイルシステムは、可変長のレコードを持っているので、繰り返しグループのサイズを自由に変えることもできた。SQLでは、こういう考え方はしない。代わりに、社員と子どものテーブルを分割する。

```
CREATE TABLE Personnel
(emp_id INTEGER NOT NULL PRIMARY KEY,
 emp_name CHAR(30) NOT NULL,
 ...);
```

Personnel：社員テーブル　emp_id：社員ID　emp_name：社員名

```
CREATE TABLE Dependents
(emp_id INTEGER NOT NULL
     REFERENCES Personnel(emp_id)
       ON UPDATE CASCADE,
 child_name CHAR(30) NOT NULL,
     PRIMARY KEY (emp_id, child_name),
 birth_date DATE NOT NULL,
 sex_code CHAR(1) NOT NULL);
```

Dependents：被扶養者テーブル　emp_id：社員ID　child_name：子どもの名前
birth_date：誕生日　sex_code：性別

子どもを追加したければ、被扶養者テーブルに行を追加する。子どもを削除したければ、被扶養者テーブルから行を削除する。対象の子どもが4人目でも5人目でも関係ないし、どちらのテーブルでもNULLを許可する必要がない。トレードオフは、スキーマにおけるテ

ーブル数が増えることだが、ストレージの消費量はむしろ少なくなる。なぜなら、存在する子どものデータしか持つ必要がないので、NULLの余計なスペースが不要になるからだ。目的は、任意のホスト言語がデータを使いやすいように可能な限りシンプルなフォーマットでデータを保持することだ。

　ガブリエル・ウィコウスキは、DB2の教育クラスで、次のようなセールスマンの業績を月別に管理するサンプルテーブルを使っていた。

```
CREATE TABLE AnnualSales1
(salesman CHAR(15) NOT NULL PRIMARY KEY,
 jan DECIMAL(5,2),
 feb DECIMAL(5,2),
 mar DECIMAL(5,2),
 apr DECIMAL(5,2),
 may DECIMAL(5,2),
 jun DECIMAL(5,2),
 jul DECIMAL(5,2),
 aug DECIMAL(5,2),
 sep DECIMAL(5,2),
 oct DECIMAL(5,2),
 nov DECIMAL(5,2),
 "dec" DECIMAL(5,2)); -- DEC[IMAL]は予約語なのでダブルクォートで囲む
```

　このようなフォーマットのテーブルだと、各月の列にはNULLを許可せざるを得ない。これに対し、次のような行持ちの形にすれば、NULLを使う必要はなく、テーブルサイズも小さくなる。

```
CREATE TABLE AnnualSales2
(salesman CHAR(15) NOT NULL,
 sale_month CHAR(3)
     CONSTRAINT valid_month_abbrev
     CHECK (sale_month IN ('Jan', 'Feb', 'Mar', 'Apr',
                           'May', 'Jun', 'Jul', 'Aug',
                           'Sep', 'Oct', 'Nov', 'Dec')),
 sales_amt DECIMAL(5,2) NOT NULL,
     PRIMARY KEY(salesman, sale_month));
```

ウィコウスキのDB2 における実際のサンプルではDASD ストレージにおける損益分岐点は4月だった。つまり、AnnualSales1テーブルとAnnualSales2が同じ程度の容量を必要とするのが、4月だったのだ。

個々のセールスマンを扱うクエリは、AnnualSales2よりもAnnualSales1を使ったほうがずっと速い。これは、AnnualSales1 では、1 人のセールスマンについてのすべてのデータが1 行に保持されているからだ。NULLの列を扱うための処理が必要とされるが、それほど複雑ではない。したがって、AnnualSales1を使う唯一の理由は、DWHにおいて高いパフォーマンスが要求される場合だ。このような設計は、OLTPでは許されない。

33.2

添え字列を使った配列

多次元配列を擬似的にテーブルにマッピングする別の方法がある。それは、添え字の列を持ったテーブルを作ることだ。テーブル定義は次のようになる。

```
CREATE TABLE Foobar2
(i INTEGER NOT NULL PRIMARY KEY
     CONSTRAINT valid_array_index
       CHECK(i BETWEEN 1 AND 5),
 element_val REAL NOT NULL);
```

これは最初のアプローチよりも難しく見えるが、実際はオリジナルのPascalの宣言が裏でやっていることに近い。添え字は、一意な物理アドレスを解決する。したがって`foobar[i]`が同時に2つの値を持つことは不可能である。これはつまり、iはキーだということだ。Pascalのコンパイラは、添え字が宣言された範囲内に収まっているかどうかをチェックする。これは、`CHECK`制約に相当する。こう考えれば、意外に両者の考え方が近いことがわかる。

このアプローチの利点は何といっても、多次元配列を扱うのが簡単なことだ。添え字用の列を追加するだけでよい。たとえば、Pascalの次のような3次元配列の宣言があるとする。

```
ThreeD : ARRAY [1..3, 1..4, 1..5] OF REAL;
```

これは次のようなテーブル定義に変換できる。

```
CREATE TABLE ThreeD
(i INTEGER NOT NULL
     CONSTRAINT valid_i
       CHECK(i BETWEEN 1 AND 3),
 j INTEGER NOT NULL
     CONSTRAINT valid_j
       CHECK(j BETWEEN 1 AND 4),
 k INTEGER NOT NULL
     CONSTRAINT valid_k
       CHECK(k BETWEEN 1 AND 5),
 element_val REAL NOT NULL,
     PRIMARY KEY (i, j, k));
```

GROUP BY句で添え字を指定したSELECT文を使えば、行列の合計を求められることは明らかだ。

```
SELECT i, j, SUM(element_val) -- k列の合計
  FROM ThreeD
 GROUP BY i, j;

SELECT i, SUM(element_val) -- j列とk列の合計
  FROM ThreeD
 GROUP BY i;

SELECT SUM(element_val) -- 配列全体の合計
  FROM ThreeD;
```

もし最初の1 要素／ 1 列のアプローチを採用していたら、テーブルはelement_val_111～element_val_345までの60列を保持しなければいけなかっただろう。こんな多くの列を操作するのは骨が折れるし、GROUP BY句で配列に対する射影を行うこともできない。

添え字列を使うもう1つの利点は、添え字に整数以外のデータ型も使えることだ。たとえば、DATEやTIMEを使うと便利なことは多いし、文字型や近似的な数値型を使うこともできる。

33.3

SQL における行列操作

行列は配列と同じではない。ここでは深く論じないが、行列は、特定の性質を備えた数学的構造である。詳しくは、大学の代数入門コースのテキストを見てほしい。SQL でも行列に対する多くの操作はやろうと思えば可能だが、SQLでやるのはおすすめできない。なぜなら、そのようなクエリは複雑でリソースを食うし、時間もかかるからだ。SQLは計算向きの言語ではない。

添え字列を使って2次元配列のテーブルを持っているとしよう。全列にNOT NULL制約が付与されているとする。NULLの存在は線形代数では定義されていないし、線形代数に3値論理を持ち込むつもりもない。

SQLで行列を扱うときの問題として、行列における行と列が、SQLにおける行と列と同じ概念ではない、という問題がある。本節の以下の記述を読むにあたり、両者を混同しないよう気をつけてほしい。

```
CREATE TABLE MyMatrix
(element_val INTEGER NOT NULL, -- 任意の数値型
 i INTEGER NOT NULL CHECK (i > 0),
 j INTEGER NOT NULL CHECK (j > 0),
     CHECK ((SELECT MAX(i) FROM MyMatrix)
          = (SELECT COUNT(i) FROM MyMatrix)),
     CHECK ((SELECT MAX(j) FROM MyMatrix)
          = (SELECT COUNT(j) FROM MyMatrix)));
```

MyMatrix：行列テーブル　element_val：要素の値　i：添え字i　j：添え字j

2 つのCHECK 制約は、各要素の添え字が適切な範囲内にあることを保証している。ここでは添え字は1から始めるが、少しロジックを

変更すれば任意の値から開始できる。

33.3.1　行列の相等性

次に紹介する行列の相等性テストのクエリは、「SQL Matrix Processing」（S. Mrdalj, B. Vujovic, V. Jovanovic著, *Database Programming and Design*, 1996年8月号）に掲載されたものだ。2つの行列が等しい条件は、両者のカーディナリティと、両者の交差のカーディナリティが等しいことである。

```
CREATE TABLE MatrixA
(i INTEGER NOT NULL
     CHECK (i BETWEEN 1 AND 10),
 j INTEGER NOT NULL
     CHECK (j BETWEEN 1 AND 10),
 element_val INTEGER NOT NULL,
     PRIMARY KEY (i, j));
```

MatrixA

```
i k element_val
=================
1 1 2
1 2 -3
1 3 4
2 1 -1
2 2 0
2 3 2
```

```
CREATE TABLE MatrixB
(i INTEGER NOT NULL
     CHECK (i BETWEEN 1 AND 10),
 j INTEGER NOT NULL
     CHECK (j BETWEEN 1 AND 10),
 element_val INTEGER NOT NULL,
     PRIMARY KEY (i, j));
```

MatrixB

```
k j element_val
=================
1 1 2
1 2 -3
1 3 4
2 1 -1
2 2 0
2 3 2
```

```
SELECT COUNT(*) FROM MatrixA
UNION
SELECT COUNT(*) FROM MatrixB
UNION
SELECT COUNT(*)
  FROM MatrixA AS A, MatrixB AS B
 WHERE A.i = B.i
   AND A.j = B.j
   AND A.element_val = B.element_val;
```

結果

```
 count
======
     6
```

もしこのクエリが1 つの数を返せば、MartixA とMatrixB は等しい。これは次のように差集合演算でテストすることもできる。

```
NOT EXISTS
(SELECT * FROM MatrixA
 EXCEPT
 SELECT * FROM MatrixB)
AND NOT EXISTS
(SELECT * FROM MatrixB
 EXCEPT
 SELECT * FROM MatrixA)
```

33.3.2　行列の加法

行列の加法と減法は、同じ型の行列の間でのみ可能である【訳注1】。単純な加法の方法は明らかだ。

```
SELECT A.i, A.j, (A.element_val + B.element_val) AS total
  FROM MatrixA AS A, MatrixB AS B
 WHERE A.i = B.i
   AND A.j = B.j;
```

結果

```
 i  j  total
============
 1  1      4
 1  2     -6
 1  3      8
 2  1     -2
 2  2      0
 2  3      4
```

より厳密には行列が互いに同じ型であるかをチェックする処理を追加すべきだろう。添え字の開始番号は1または0と仮定する。

```
SELECT A.i, A.j, (A.element_val + B.element_val) AS total
  FROM MatrixA AS A, MatrixB AS B
 WHERE A.i = B.i
   AND A.j = B.j
   AND (SELECT COUNT(*) FROM MatrixA)
     = (SELECT COUNT(*) FROM MatrixB)
   AND (SELECT MAX(i) FROM MatrixA)
     = (SELECT MAX(i) FROM MatrixB)
   AND (SELECT MAX(j) FROM MatrixA)
     = (SELECT MAX(j) FROM MatrixB);
```

加算の結果をテーブルに反映するには、このクエリをUPDATE文に組み込めばよい。

```
UPDATE MatrixA
   SET element_val = element_val
                     + (SELECT element_val
                          FROM MatrixB
                         WHERE MatrixB.i = MatrixA.i
                           AND MatrixB.j = MatrixA.j)
 WHERE (SELECT COUNT(*) FROM MatrixA)
     = (SELECT COUNT(*) FROM MatrixB)
   AND (SELECT MAX(i) FROM MatrixA)
     = (SELECT MAX(i) FROM MatrixB)
   AND (SELECT MAX(j) FROM MatrixA)
     = (SELECT MAX(j) FROM MatrixB);
```

33.3.3　行列の乗法

スカラ定数による乗法は、直接的で簡単だ。

```
UPDATE MyMatrix
   SET element_val = element_val * :constant;
```

行列同士の乗法は、可換ではない点に気をつければ、それほど難しくはない。乗法では、行列Aの行数と行列Bの列数は一致する。これはつまり、A[i, k]* B[k, j]= C[i, j]ということである。例を使って説明しよう。

MatrixA

```
i j element_val
=================
1 1 2
1 2 -3
1 3 4
2 1 -1
2 2 0
2 3 2
```

MatrixB

```
i j element_val
================
1 1 -1
1 2 2
1 3 3
2 1 0
2 2 1
2 3 7
3 1 1
3 2 1
3 3 -2
```

```
SELECT MatrixA.i, MatrixB.j,
       SUM(MatrixA.element_val * MatrixB.element_val) AS element_val
  FROM MatrixA, MatrixB
 WHERE MatrixA.j = MatrixB.i
 GROUP BY MatrixA.i, MatrixB.j;
```

結果

```
 i j element_val
================
 1 1           2
 1 2           5
 1 3         -23
 2 1           3
 2 2           0
 2 3          -7
```

これは行列の乗法の定義から直接導かれる。

33.3.4　行列の転置

行列の転置は簡単だ。

```
SELECT j, i, element_val FROM MatrixA;
```

更新の形にするならこうなる【訳注2】。

```
UPDATE MatrixA
   SET i = j, j = i;
```

列または行のベクトルによる乗法は、行列の乗法の特殊ケースにすぎないので、少し簡単になる。ベクトルVと行列MatrixAが与えられているとき、以下のようになる。

```
SELECT i, SUM(A.element_val * V.element_val)
  FROM MatrixA AS A, VectorV AS V
 WHERE V.j = A.i
 GROUP BY A.i;
```

クロス表や統計関数は、伝統的にデータを保持するために配列を使ってきた。だが、SQLでそれらを扱うために行列を使う必要はない。

これら以外にも、SQL で行列操作を行うことは可能だが、コードが複雑になり、実行時間も長くなるため、わざわざSQLでやる価値はない。もし読者が固有値と行列式のクエリを考えてくれたら、本書の将来の版において紹介させてほしい。

33.3.5　行列のソート

標準SQLではROW_NUMBER関数を使えば行または列をソートできる。

```
UPDATE MatrixA
   SET i = ROW_NUMBER() OVER (ORDER BY element_val)
 WHERE j = :my_column;
```

同様に、行をソートするには次のようにする。

```
UPDATE MatrixA
   SET j = ROW_NUMBER() OVER (ORDER BY element_val)
 WHERE i = :my_column;
```

33.3.6　その他の行列操作

もし読者が数学について高度な教育を受けていれば、連立方程式を解くための行列と関連の深い「行列式」、「逆関数」、「固有値」といった用語を覚えているかもしれない。再帰共通表式などを駆使すれば、こうしたものを求める一般的なクエリを書くことも可能かもしれない。だがSQLは計算向きの言語ではないため、行列のサイズが大きくなると高速には動作しないだろう。そうした目的には、MathematicaやMathLabといった製品を使うのが適している。

33.4

テーブルを配列に展開する

レポートやDWHのサマリテーブルでは、1 つの配列を水平方向に1行で保持するフォーマットを作りたいことが多い。この場合、添え字は列名の中に埋め込み、「mon_12」のように時間的期間を表すことが多い。

例を使って説明しよう。トラックドライバーのタイムカードから計算した労働時間を保持するテーブルを考える。このテーブルには、各週におけるドライバーの労働時間が格納される（1年分を保持することから、週の通し番号は0から51または52までだ）。このテーブルから、ドライバーにつき1 行で、1 ページに6 週ずつの労働時間を記載するレポートを作成したい。もととなるタイムカードのテーブルは次のような定義だ。

```
CREATE TABLE Timecards
(driver_name CHAR(25) NOT NULL,
 week_nbr INTEGER NOT NULL
     CONSTRAINT valid_week_nbr
       CHECK(week_nbr BETWEEN 0 AND 52),
 work_hrs INTEGER
     CONSTRAINT zero_or_more_hours
       CHECK(work_hrs >= 0),
     PRIMARY KEY (driver_name, week_nbr));
```

Timecards：タイムカードテーブル　driver_name：ドライバー名　week_nbr：週番号　work_hrs：労働時間

レポートに必要な行集合を作るには、このテーブルを「水平展開」する必要がある。そこで、次のような一時テーブルを作る。

```
CREATE TEMPORARY TABLE TimeReportWork -- working storage
(driver_name CHAR(25) NOT NULL PRIMARY KEY,
 wk1 INTEGER, -- これらの列はNULLを許可する
 wk2 INTEGER,
 wk3 INTEGER,
 wk4 INTEGER,
 wk5 INTEGER,
 wk6 INTEGER);
```

TimeReportWork：労働時間レポートワークテーブル　　driver_name：ドライバー名
wk1～Wk6：1週目から6週目の労働時間

このテーブルについて重要な点は、wk1～wk6列はNULLを許可する必要があることだ。

```
INSERT INTO TimeReportWork (driver_name, wk1, wk2, wk3, wk4, wk5, wk6)
SELECT driver_name,
       SUM(CASE WHEN week_nbr = :rpt_week_nbr
                THEN work_hrs ELSE 0 END) AS wk1,
       SUM(CASE WHEN week_nbr = :rpt_week_nbr - 1
                THEN work_hrs ELSE 0 END) AS wk2,
       SUM(CASE WHEN week_nbr = :rpt_week_nbr - 2
                THEN work_hrs ELSE 0 END) AS wk3,
       SUM(CASE WHEN week_nbr = :rpt_week_nbr - 3
                THEN work_hrs ELSE 0 END) AS wk4,
       SUM(CASE WHEN week_nbr = :rpt_week_nbr - 4
                THEN work_hrs ELSE 0 END) AS wk5,
       SUM(CASE WHEN week_nbr = :rpt_week_nbr - 5
                THEN work_hrs ELSE 0 END) AS wk6
  FROM Timecards
 WHERE week_nbr BETWEEN :rpt_week_nbr AND (:rpt_week_nbr - 5)
 GROUP BY driver_name;
```

WHERE句の週の数は、レポートでカバーすべき範囲によって変わる。このクエリでは、パラメータ：rpt_week_nbrでレポートが作成される週を受け取り、そこからさかのぼって5 週間を表示している。ドライバーがある週にまったく働いていなければ、その週の計算結果の列は0となる。ただし、ドライバーが過去6 週間まったく働いて

いなければ、そのドライバーはレポートに現れない（タイムカードの情報がない以上、サマリにも現れない）。情報の欠落を防ぐには、社員テーブルと外部結合しなければならない。

　このクエリはNULL を0 に変換する。もし「...ELSE 0」句を削除すれば、SUM 関数はすべてNULLの週に対してはNULLを返すようになる。だが、そもそもレポート期間に働いていないドライバーと、特定の週に働いた時間が0 だったドライバーを区別するため、ELSE 句はあったほうがいいだろう。この違いは、給与を計算する際に重要だからだ。

33.5

テーブル形式の配列を計算する

データがテーブルの形式で与えられている場合、ある配列または値の集合を別の集合と比較する必要があることは多い。注意すべきは、集合の比較においては要素の順序は関係ないが、配列の場合は関係することだ。例として、社員と被扶養者の2 つのテーブルを考える。子どもたちは出生順の添え字を振られている（最初に生まれた子どもは1だ）。

```
CREATE TABLE Personnel
(emp_id INTEGER PRIMARY KEY,
 emp_name CHAR(15) NOT NULL);
```

Personnel：社員テーブル　emp_id：社員ID　emp_name：社員名

```
CREATE TABLE Dependents
(emp_id INTEGER NOT NULL, -- 親
 dependent_name CHAR(15) NOT NULL, -- 配列要素
 birth_seq INTEGER NOT NULL, -- 添え字
     PRIMARY KEY (emp_id, dependent_name));
```

Dependents：被扶養者テーブル　emp_id：社員ID　dependent_name：被扶養者名　birth_seq：出生順

「子どもが同じ名前の集合を持つ社員のペアを見つける」というクエリは、極めて強い絞り込み条件を持つ。「子どもが生まれた順序も一致していること」という条件を付けるとさらに条件が強くなる。社員XとYは、同数の子どもを持っていて、さらに名前の集合が一致しなければならない。複数の子どもに同じ名前を付けている親はいないと仮定する（すべての子どもにジョージと名付けたジョージ・フォアマンのような人物は、ここでは除外する）。また、双子

であっても生まれた順序は異なるとする。被扶養者テーブルに、次のようなテストデータを登録しよう。

Dependents

emp_id	dependent_name	birth_seq
1	Tom	1
1	Dick	2
1	Harry	3
2	Harry	1
2	Tom	2
2	Dick	3
3	Tom	1
3	Dick	2
3	Harry	3
4	Harry	1
4	Tom	2
5	Curly	2
5	Harry	3
5	Moe	1

このテストデータにおいて、社員1、2、3 は皆、Tom、Dick、Harry という名前の子を持っている。社員1と3に関しては、出生順序も一致するが、社員2は不一致だ。

この問題を解くクエリはいろいろなパターンが考えられるが、次の解は、他の解にはない柔軟性がある。まず、各社員の被扶養者の数を得るビューを作る。

```
CREATE VIEW FamilySize (emp_id, tally)
AS
SELECT emp_id, COUNT(*)
  FROM Dependents
 GROUP BY emp_id;
```

次に、子どもの数が同じ社員のペアを保持するもう1つのビューを作る。このビューは、他の統計的作業のためにも有用だ。

```
CREATE VIEW Samesize (emp_id1, emp_id2, tally)
AS
SELECT F1.emp_id, F2.emp_id, F1.tally
  FROM Familysize AS F1, Familysize AS F2
 WHERE F1.tally = F2.tally
   AND F1.emp_id < F2.emp_id;
```

要素数（つまり子どもの数）が同数の集合の相等性をテストするには、自己結合を使う。もし1つの集合が他の集合と完全に1対1対応し、かつ出生順も同じなら、2つの集合は等しい。

```
SELECT D1.emp_id, ' named his ', S1.tally, ' kids just like ', D2.emp_id
  FROM Dependents AS D1, Dependents AS D2, Samesize AS S1
 WHERE S1.emp_id1 = D1.emp_id
   AND S1.emp_id2 = D2.emp_id
   AND D1.dependent_name = D2.dependent_name
   AND D1.birth_seq = D2.birth_seq
 GROUP BY D1.emp_id, D2.emp_id, S1.tally
HAVING COUNT(*) = S1.tally;
```

もし出生順が重要でなければ、
「D1.birth_seq=D2.birth_seq」という条件は削除してよい。
これは2列目に対する厳密な関係除算となっている。

```
 emp_id   named his tally  kids just like emp_id
================================================
      1   named his     3  kids just like      3
```

【訳注1】　行列の型は行数と列数によって定義される。2つの行列の型が等しいとは、行数と列数が等しいことである。

【訳注2】　この更新はOracleでは正常に動作するが、PostgreSQLでは一意制約違反によってエラーになる。おそらくPostgreSQL では、内部的に1 行更新されるたびに制約チェックが行われているのだと思われる。またMySQLは、左から右へ順序性を持った更新を行うため、結果が不正になる。

34

集合演算

Set Operations

ここで述べる集合演算は、和（UNION）、交差（INTERSECTION）、差（EXCEPT）を想定している。SQLにおける集合は、もちろんテーブルである。これらの演算は集合論においても基本的な演算子であり、米国ではここ数十年、公立学校の教育カリキュラムの中に含まれている。リレーショナルモデルは集合論に基づいており、SQL は初期からかなりの種類の集合演算を備えていた。だが、それはあくまで建前だ。標準SQLはたしかに基本的な集合演算を持っていたが、それらは実際の製品においては十分にサポートされていない。

SQLにはまた、高校の集合論の授業では習わなかった問題がある。SQLのテーブルは多重集合（バッグとも呼ばれる）であり、集合と違って重複要素を許す。コッドは、リレーショナルモデルはより厳密であるべきで、本当の集合しか使うべきでないと考えていた。SQLは、重複行を、`ALL`と`DISTINCT` 修飾子をさまざまな場所で使うことで制御している。`ALL`は重複行を保存し、`DISTINCT`は除外する。

集合演算の結果を形式的に論じるため、RをTableAまたはTableB（両方であることもある）の重複行であるとしよう。m はTableA におけるR の重複行の行数、n をTableB におけるR の重複行の行数とする(m >=0かつ n >=0)。非形式的に言うと、エンジンは集合演算

を行う際、行と行を比較して2 つのテーブルをペアにする。各演算子の動作を具体的に見ていこう。サンプルとして次のようなテーブルを作る。

```
CREATE TABLE S1 (a1 CHAR(1));

INSERT INTO S1
VALUES ('a'), ('a'), ('b'), ('b'), ('c');

CREATE TABLE S2 (a2 CHAR(1));

INSERT INTO S2
VALUES ('a'), ('b'), ('b'), ('b'), ('c'), ('d');
```

34.1

UNIONとUNION ALL

UNIONはSQL-86からサポートされている。構文は次のとおりだ。

```
<table expression> UNION [ALL] <table expression>
```

UNION が入力にとる2 つの式はテーブルであり、結果として1 つのテーブルを生成する。2 つのテーブルはUNION互換である必要がある。これは以下のすべての条件を満たすことをいう。

1. 2つのテーブルが同じ列数である
2. 同じ位置の各列は同じデータ型（または自動的な変換が可能）である

要するに、行が同じ構造を持っているということである。もちろん、その構造は結果テーブルにも引き継がれる。多くの実装が、結果テーブルを作るために何らかのデータ型の変換を行うが、それらは実装依存なので、詳細は各自調べてほしい。

UNIONには、UNIONとUNION ALLの2 種類がある。単純なUNIONは高校の授業で習ったのと同じ演算子である。2つのテーブルのどちらか一方に存在していれば、その行を、冗長な重複行を排除して返す。「冗長な重複行」という表現は奇妙だと思うかもしれない。これはつまり、結果テーブルには1 つのコピーだけ残してあとは削除するという意味だ。サンプルテーブルを使うと、以下のようになる。

```
(SELECT a1 FROM S1
 UNION
 SELECT a2 FROM S2);

===
a
b
c
d
```

古いSQL の実装では、2 つのテーブルをマージソートして、ソートの間に重複行を削除するという方法が用いられることが多かった。この方法の副作用として、結果テーブルがソートされた。しかし、現在ではハッシュ、インデックス、パラレル処理といったやり方が採用されるようになったため、必ずしも結果がソートされる保証はない。くどいようだが、SQLの演算子で結果セットの順序を保証するものは存在しないのだ。

UNION ALLは、結果テーブルにおいて重複行を保存する。古い実装では、単純に1 つのストレージをもう1つのストレージにアペンドすることで行われていた。当時の実装は、物理的に連続なファイルシステムを使っていたので、これは容易なことだった。だがやはり、この演算の結果がソートされる保証はない。サンプルテーブルを使うと以下のようになる。

```
(SELECT a1 FROM S1
 UNION ALL
 SELECT a2 FROM S2);

===
a
a
a
b
b
b
b
b
c
c
d
```

結果の列に名前を与えたければ、導出テーブルとASを使えば可能だ。

```
SELECT rent, utilities, phone
  FROM (SELECT a, b, c
          FROM OldLocations WHERE city = 'Boston'
        UNION
        SELECT x, y, z
          FROM NewLocations WHERE city = 'New York')
    AS Cities (rent, utilities, phone);
```

製品によっては、同じテーブルに対するUNIONをOR やINに変換することで最適化を図ることがある。たとえば、

```
SELECT city_name, 'Western'
  FROM Cities
 WHERE market_code = 't'
UNION ALL
SELECT city_name, 'Eastern'
  FROM Cities
 WHERE market_code = 'v';
```

このクエリは、次のような（おそらくより効率的な）コードに書き換えられる。このコードが効率的なのは、テーブルに対するスキャンが減るからだ。

```
SELECT city_name,
       CASE market_code
       WHEN 't' THEN 'Western'
       WHEN 'v' THEN 'Eastern' END
  FROM Cities
 WHERE market_code IN ('v', 't');
```

なお、まったく同じテーブルに対するUNIONは、結果だけ見ればSELECT DISTINCTと同じだが、SELECT DISTINCTのほうが速いし、列名も保存されるので、あえてUNIONを使う必要はない。

34.1.1 UNION の実行順序

UNIONとUNION ALLは、括弧がない限り、左から右に実行される。UNIONには結合法則と交換法則が成り立つので、UNION を複数使ってもテーブルの順序は結果に影響しない。ただし、順序とグルーピングはパフォーマンスには影響する。たとえば、2つの互いに重複する行の多い2つのテーブルSmallTable1とSmallTable2があるとする。もしオプティマイザがテーブルのサイズを考慮しなければ、次のクエリを実行する際、SmallTable1をBigTableにマージし、次にSmallTable2をその結果にマージするだろう。

```
(SELECT * FROM SmallTable1)
 UNION
(SELECT * FROM BigTable)
 UNION
(SELECT * FROM SmallTable2);
```

だがこれよりは、SmallTable1とSmallTable2を先にマージする次のクエリのほうが速いだろう。

```
(SELECT * FROM SmallTable1)
 UNION
(SELECT * FROM SmallTable2)
 UNION
(SELECT * FROM BigTable);
```

繰り返すが、UNIONの最適化はかなり実装依存なので、実際にどうなるかは各自実験してほしい。

34.1.2 UNION とUNION ALL の混在

もし重複行が存在しなかったり、存在していても気にする必要がない場合は、UNIONの代わりにUNION ALLを使うほうがパフォーマンスが良い。たとえば、BigTableが、SmallTable1およびSmallTable2と重複行を持っていないとする。次のクエリは先のクエリと同じ結果を返すが、ずっと速い。

```
((SELECT * FROM SmallTable1)
 UNION
(SELECT * FROM SmallTable2))
 UNION ALL
(SELECT * FROM BigTable);
```

だが、UNIONとUNION ALLを混在させるときは注意点が1 つある。左から右の順序で実行されるということは、最後に使われた演算子が結果に影響を与えるということだ。たとえば、ずっとUNION ALLを使ってきても、最後だけUNIONを使えば、結果から重複行は削除される。

34.1.3　同じテーブルの列に対するUNION

同じテーブルの複数の列を、1 つの列にまとめるには、クロス結合とCASE式を使うのが便利だ。たとえば、

```
SELECT CASE WHEN S1.seq = 1 THEN F1.col1
            WHEN S1.seq = 2 THEN F1.col2
            ELSE NULL END
  FROM Foobar AS F1
         CROSS JOIN
           Series AS S1(seq)
 WHERE S1.seq IN (1, 2)
```

これはあたかもUNION ALL のように動作する。SELECT を SELECT DISTINCTに変えれば、UNIONになる。これが普通にUNIONを使うよりも優れている点は、テーブルスキャンが1回で済むことだ。テーブルサイズが大きくなるほど、このトリックのパフォーマンス上の利点は大きい。

34.2

INTERSECTとEXCEPT

INTERSECTとEXCEPTは、2つのテーブルを入力にとり、新たなテーブルを出力する集合演算子である。入力の2つのテーブルはUNION互換である必要がある。

UNIONと同様、INTERSECT やEXCEPT の結果テーブルに列名を与えたければ、ASを使えばよい。EXCEPTを最初に導入した大手ベンダーはOracle だが、MINUSという名前の演算子で実装した。この由来は、差集合とは、最初のテーブルから2 番目のテーブルに存在する行を削除する演算だからだ。普通の言葉で言えば、「セールスマン以外の社員を教えてくれ」という質問に似ている。

サンプルに、2つの多重集合S1とS2を用意し、演算の基本的なモデルを説明しよう。

```
S1 = {a, a, b, b,    c   }
      ↕     ↕  ↕     ↕
S2 = {a,    b, b, b, c, d}
```

INTERSECTとEXCEPTは演算の前に入力テーブルから重複行を削除する。したがって、まずは2つの集合は次のようになる。

```
S1 = {a, b, c   }
      ↕  ↕  ↕
S2 = {a, b, c, d}
```

結果はこうなる。

```
S1 INTERSECT S2 = {a, b, c}
```

```
S2 EXCEPT S1 = {d}
S1 EXCEPT S2 = {}
```

UNIONと同じく、ALLオプションも使うことができるが、動作は少しトリッキーになる。これは、重複行を保存する。

```
S1 INTERSECT ALL S2 = {a, b, b, c}
```

aが結果に1つしか現れないのは、S2にaが1つしかないからだ。これに対し、bはS1とS2に2つずつ存在するので、結果にも2つ現れる。EXCEPT ALLの結果は次のようになる。

```
S2 EXCEPT ALL S1 = {b, d}
```

結果にbが現れるのは、bはS2に3つ、S1に2つあるので、3–2＝1ということだ。

INTERSECTとEXCEPTを他の演算子で書こうとするのは、思ったより難しい。キーがない場合や列数が不明の場合まで一般化せねばならないので、かなりたいへんだ。

標準SQLは、重複行に対する動作を、一致する行数を基準に定義している。mをS1のある値の行数、n をS2 の同じ値の行数としよう。UNION ALL は(m + n)の行数を生成する。INTERSECT ALLは LEAST(m, n)を生成する。EXCEPT ALLは、最初のテーブルの行数から2 番目のテーブルの行数を引いた数または0である。

プログラマの直観として、これらの演算子はEXISTS述語を使って書けばよいと思うかもしれない。だが、これはうまくいかない。理

由はNULLである。これは、コードを使って説明するほうがわかりやすい。先ほどのサンプルを再利用しよう。

```
CREATE TABLE S1 (a1 CHAR(1));
INSERT INTO S1
VALUES ('a'), ('a'), ('b'), ('b'), ('c'), (NULL), (NULL);

CREATE TABLE S2 (a2 CHAR(1));
INSERT INTO S2
VALUES ('a'), ('b'), ('b'), ('b'), ('c'), ('d'), (NULL);
```

まず、2つのテーブルの各値の行数を保持するビューを作る。

```
CREATE VIEW DupCounts (a, s1_dup, s2_dup)
AS
SELECT S.a, SUM(s1_dup), SUM(s2_dup)
  FROM (SELECT S1.a1, 1, 0
          FROM S1
        UNION ALL
        SELECT S2.a2, 0, 1
          FROM S2) AS S(a, s1_dup, s2_dup)
 GROUP BY S.a;
```

GROUP BYはNULLを1つのグループにまとめて、その行数も求める。さて、このビューを使うと、それぞれの演算子は標準SQLで直接的な書き方ができる。

```
-- S1 EXCEPT ALL S2
SELECT DISTINCT D1.a,
       (s1_dup - s2_dup) AS dups
  FROM DupCounts AS D1,
         Series AS S1
 WHERE S1.seq <= (s1_dup - s2_dup);

 a   dups
=========
 a      1
NULL    1
```

```
-- S1 INTERSECT ALL S2
SELECT DISTINCT D1.a,
       CASE WHEN s1_dup <= s2_dup
            THEN s1_dup ELSE s2_dup END AS dups
  FROM DupCounts AS D1,
        Series AS S1
 WHERE S1.seq <= CASE WHEN s1_dup <= s2_dup
                      THEN s1_dup ELSE s2_dup END;

 a   dups
=========
 a      1
 b      2
 c      1
NULL    1
```

INTERSECT ALLの結果には少し注意が必要だ。これは、bが2つあることをdups列で表現している。だが実際にINTERSECT ALLの演算を行うと、bが2行現れる。これとまったく同じ形式にするには、DISTINCTオプションを除外すればよい。結果は以下のようになる。

```
 a   dups
=========
 a      1
 b      2
 b      2  --重複行
 c      1
NULL    1
```

重複行を排除するケースも、標準の定義から簡単に書ける。重複行の件数列は1 がセットされている。

```
-- S1 INTERSECT S2
SELECT D1.a
  FROM DupCounts AS D1
 WHERE s1_dup > 0
   AND s2_dup > 0;

 a
====
 a
 b
 c
NULL
```

```
-- S1 EXCEPT S2
SELECT D1.a
  FROM DupCounts AS D1
 WHERE s1_dup > 0
   AND s2_dup = 0;

 a
====
(0行)
```

```
SELECT *
  FROM S1
 WHERE EXISTS
       (SELECT *
          FROM S2
         WHERE S1.a1 = S2.a2);

 a1
===
 a
 a
 b
 b
 c
```

34.2.1　NULLと重複行がない場合のINTERSECTとEXCEPT

INTERSECTとEXCEPTは、入力のテーブルがNULLを持っていない場合はずっと書くのが簡単だ。交差は単純に次のようになる。

```
SELECT *
  FROM S1
 WHERE EXISTS
       (SELECT *
          FROM S2
         WHERE S1.a1 = S2.a2);

 a1
===
 a
 a
 b
 b
 c
```

このサンプルデータにはNULLがあるため、結果がINTERSECTと一致しない。

次のように書くこともできる。

```
SELECT *
  FROM S2
 WHERE EXISTS
       (SELECT *
          FROM S1
         WHERE S1.a1 = S2.a2);
```

あるいはこう書いてもいい。

```
SELECT DISTINCT S2.*
  FROM (S2 INNER JOIN S1 ON S1.a1 = S2.a2);
```

NULLのため、上記3 つの結果はすべて不一致である。これらは、次節で見せる解のための踏み台として見せたものだが、結合とEXISTS述語のどちらが速いかは、SQLエンジンによるので、どちらの書き方もマスターしておくべきだ。

差集合は、外部結合によって記述できる。次のコードは、ジム・パンタッヤによるものだ。

```
SELECT DISTINCT S2.*
  FROM (S2 LEFT OUTER JOIN S1
    ON S1.a1 = S2.a2)
 WHERE S1.a1 IS NULL;
```

34.2.2 NULL と重複行に対するINTERSECT と EXCEPT

以下に示すINTERSECTとEXCEPTのコードは、イツァク・ベン-ガンによるものだ。UNIONとDISTINCT演算子をうまく使って集合論の定義を実装している。

```
-- S1 INTERSECT S2
SELECT D.a
  FROM (SELECT DISTINCT a1 FROM S1
UNION ALL
SELECT DISTINCT a2 FROM S2) AS D(a)
 GROUP BY D.a
HAVING COUNT(*) > 1;

 a
====
 a
 b
 c
NULL
```

```
-- S1 INTERSECT ALL S2
SELECT D2.a
  FROM (SELECT D1.a, MIN(cnt) AS mincnt
          FROM (SELECT a1, COUNT(*)
                  FROM S1
                 GROUP BY a1
                UNION ALL
                SELECT a2, COUNT(*)
                  FROM S2
                 GROUP BY a2) AS D1(a, cnt)
         GROUP BY D1.a
        HAVING COUNT(*) > 1) AS D2
          INNER JOIN
            Series
    ON seq <= mincnt;

 a
====
 a
 b
 b
 c
NULL
```

```
-- S1 EXCEPT ALL S2
SELECT D2.a
  FROM (SELECT D1.a, SUM(cnt)
          FROM (SELECT a1, COUNT(*)
                  FROM S1
                 GROUP BY a1
                UNION ALL
                SELECT a2, -COUNT(*)
                  FROM S2
                 GROUP BY a2) AS D1(a, cnt)
         GROUP BY D1.a
        HAVING SUM(cnt) > 0) AS D2(a, dups)
          INNER JOIN
            Series ON seq <= D2.dups;

 a
====
 a
NULL
```

35

部分集合

Subsets

本章では、クエリを使って部分集合演算を定義しようと思う。これは、与えられた集合から特定の部分集合を抽出するもので、集合同士に作用する集合演算の逆だ。テーブルから部分集合を抽出する簡単な方法は、`WHERE` 句を使うことである。そうすれば条件に合致する行を選択できる。だが、我々の欲しい部分集合のすべてをこのような単純な方法で抽出できるわけではない。以下では、少し複雑だが、有用な部分集合をテーブルから抽出するトリックの数々を紹介しよう。

35.1

テーブルにおけるN 番目ごとの行

SQLは集合指向言語であり、各行のディスク上における物理的位置を特定することはできない。その代わり、論理的な一意キーによって行を引き出す。もし社員のファイルがあって、そこから何かのサンプル調査のためn番目ごとの社員の行をピックアップするならば、それは非常にたやすい。ファイル全体をループしてn番目ごとの社員を別ファイルに出力するプロシージャを書けばよい。

これをSQLでやろうとすると、簡単に考えれば`MOD(emp_nbr, :n)`を計算して0になる行だけを抽出すればよい。この方法が通じないのは、社員番号が連続的な整数ではない場合だ。識別子は一意であることは必須だが、必ずしも連続的である必要はない。かつては、連続的な識別子を得るにはベンダーの拡張機能を使って物理的な行ロケータを参照する必要があった。これらの機能は極めて実装依存だったが、低レベルの機能だったため高速に動作するのが利点だった。今日では順序関数があるので、こうした機能に頼る必要はない。

```
SELECT P1.emp_nbr
  FROM (SELECT emp_nbr,
               ROW_NUMBER()
                 OVER(ORDER BY emp_nbr)
          FROM Personnel) AS P1(emp_nbr, emp_seq)
 WHERE MOD (emp_seq, :n) = 0;
```

順序関数を使わずに同じことをするなら次のようになる。

```
SELECT P1.emp_nbr
  FROM Personnel AS P1, Personnel AS P2
 WHERE P1.emp_nbr >= P2.emp_nbr
 GROUP BY P1.emp_nbr
HAVING MOD (COUNT(*), :n) = 0;
```

　このクエリはP1の値よりも小さいP2の行数を数えている。この古いトリックはROW_NUMBER より遅いので、いまさら使う価値はないが、古いコードでは見ることもあるだろう。

35.2

テーブルからランダムな行を選択する

単刀直入に言って、SQLでテーブルから行をランダムに選択することは直接的にはできない。標準SQLには乱数生成の機能はないし、擬似乱数を生成する関数を独自拡張として持っているベンダーも稀だ【訳注1】。統計サンプルのためにランダムな行を選択したいなら擬似乱数の生成機能を持った他の言語で行うことだ。

仮にSQLが乱数関数を持っていれば、次のような形だろう。

```
RANDOM(x) および RANDOM(DISTINCT x)
```

ここでxはシードまたはダミー値である。乱数を生成するアルゴリズムは、大きく重複を考慮するか否かで分かれる。現実世界の例にたとえれば、ポーカーで配られるカードは重複なしの乱数、クラップスのサイコロは重複ありの乱数だ。もしポーカーをしていて複数のプレイヤーが同じカードを得たら、間違えてピノクルのカードを使っている。一方、クラップスでは、それぞれのサイコロの出る目は互いに独立なので、同じ目が複数出ることがある。

問題は、SQL が集合指向言語であるため、行に対する操作を「一度にすべて」行おうとすることである。だが、ランダムな集合を決定する手続きは定義から非決定的であるため、決定的な論理式で表現できないのだ。

SQL/PSMは、`CREATE PROCEDURE`の構文に`DETERMINISTIC`または`NOT DETERMINISTIC`のオプションを持っている。`DETERMINISTIC`オプションは、特定の入力の集合に対して一度計算した結果は、同じ入力の集合に対しては、他のあらゆる場所で利用

できることを意味する。反対に、NOT DETERMINISTICオプションは、同じパラメータを与えても、必ずしも同じ結果にはならないという意味だ。

残念なことにSQL製品は統一性のない独自拡張を持っている。たとえば、Oracleの乱数関数は非決定的だが、SQL Server のそれは決定的だ（つまり擬似乱数関数である）。サンプルで見てみよう。RANDOMを乱数関数とする。

```
CREATE TABLE RandomNbrs
(seq_nbr INTEGER NOT NULL PRIMARY KEY,
 random_nbr FLOAT NOT NULL);

INSERT INTO RandomNbrs
VALUES (1, RANDOM()),
       (2, RANDOM()),
       (3, RANDOM());
```

SQL Server では、この結果は、3 行とも同じrandom_nbr 列の値になるが、Oracleでは異なる値になる可能性がある。

DEFAULT句ではサブクエリを使うことはできないが、CURRENT_TIMESTAMPやCURRENT_USER のようなシステム生成の値は許されている。いくつかの実装では、乱数関数もデフォルト値として使用できる。

```
CREATE TABLE RandomNbrs2
(seq_nbr INTEGER PRIMARY KEY,
 random_nbr FLOAT
     DEFAULT ((CASE (CAST(RANDOM() + 0.5 AS INTEGER) * -1)
                WHEN 0.0 THEN 1.0 ELSE -1.0 END)
                  * MOD (CAST(RANDOM() * 100000 AS INTEGER), 10000)
                  * RANDOM())
     NOT NULL);

INSERT INTO RandomNbrs2
VALUES (1, DEFAULT),
       (2, DEFAULT),
       (3, DEFAULT),
       (4, DEFAULT),
       (5, DEFAULT),
       (6, DEFAULT),
       (7, DEFAULT),
       (8, DEFAULT),
       (9, DEFAULT),
       (10, DEFAULT);
```

もっとも、本当にこういうことをやりたいなら、最善の方法は、テーブルに乱数を保持する列を宣言して、外側の言語の算術ライブラリの擬似乱数生成関数を使ってデータをロードすることだ。

乱数生成関数の初期値は「シード」と呼ばれる（以降、Random[0]と表記する）。これはシステム時刻など一意に近い値が望ましい。シードはシーケンスにおける最初の値Random[1]を作る。以降、呼び出しのたびにRandom[n]が生成される。このとき関数は次の数を作るために1つ前の値を入力に使う。

「擬似乱数生成関数」は多くの場合、単に「乱数生成関数」を指して使われるが、これは技術的には誤った使い方だ。本当の乱数を得るには放射線崩壊などカオス系における物理現象から導く必要がある。すべての乱数関数は、シーケンスにおいて前に登場した値をもう一度生成することがあり、そうすると循環を始める。手続きは決定的であり、真にランダムな結果を得るのは難しい。とはいえ、

サイクルが十分に長くて、その範囲内において一定のランダム性の基準をクリアできれば、十分実用的だ。

乱数生成のアルゴリズムには多様なものがある。たとえば、線形合同を使うものは以下のような形の式になる。

```
Random[n + 1]:= MOD ((x * Random[n] + y), m);
```

x、y、mは、互いに素である必要がある。クヌースは、Random[0]が2または5の倍数ではなく、m = 10^e(e >= 5)かつy = 0のとき、MOD (x, 200)は、(3, 11, 13, 19, 21, 27, 29, 37, 53, 59, 61, 67, 77, 83, 91, 109, 117, 123, 131, 133, 139, 141, 147, 163, 171, 173, 179, 181, 187, 189, 197)の中に含まれ、サイクルは5 * 10^(e-2)になるという証明を与えた。

次の式は、C言語プログラマだった人は見たことがあるかもしれない。

```
Random(n + 1) := (Random(n) * 1103515245) + 12345;
Random(n + 1) := MOD ((16807 * Random(n)), ((2^31) - 1));
```

最初の式はMOD 関数が不要という利点がある。とはいえ、最も単純でおすすめの乱数関数（Park and Miller）は、MOD関数を使う次のものだ【訳注2】。

```
Random(n + 1) := MOD ((48271 * Random(n)), ((2^31) - 1));
```

剰余は素数であることに注意してほしい。これは重要なことだ。この関数のサイクルは((2^31)–2)、つまり2, 147, 483, 646 ——関数

が繰り返しを始めるまで20 億以上もある。これが読者の使うアプリケーションにとって十分な長さか判断が必要だ。

もし使っている製品で排他的論理和を計算する関数があれば、シフトレジスタのアルゴリズムを使うこともできる。最近は小さなマシンであっても32ビットを扱えるものがほとんどだ。いくつか有用なものを挙げよう。

```
Random(n + 1) := Random(n - 103) XOR Random(n - 250);
Random(n + 1) := Random(n - 1063) XOR Random(n - 1279);
```

もしベンダーのライブラリに乱数関数がないなどの理由で、これを自前で書かねばならないならば、主キーの列からシードを取得し、システムクロックの小数位を呼び出すことだ。

```
keycol + EXTRACT (SECOND FROM CURRENT_TIME) * 1000
```

主キーはある種のパターンに従う可能性があるが、システムクロックと組み合わせることでその問題を回避できるし、結果も一意性を保証される。

もう1 つの方法は、主キーまたは一意な列にハッシュアルゴリズムを適用することだ。ハッシュ関数は出力が一様に分散する性質があるし、うまくアルゴリズムを選べばほぼ一意な乱数を生成できる。ただし、これを実現するにはSQL の限られた能力で書くことができるぐらい、ハッシュアルゴリズムが簡単でなければならない。

いったん乱数の列を取得できれば、次のSQL文によってそれをランダムな順序の数列に変えることができる。

```
UPDATE RandomNbrs
   SET random_nbr = (SELECT COUNT(*)
                       FROM Series AS S1
                      WHERE S1.random_nbr <= Series.seq_nbr);
```

テーブルからランダムな1行を選択するには、次のアプローチを使えばよい。まずは次のようなビューを作る。

```
CREATE VIEW LotteryDrawing (keycol, ..., spin)
AS SELECT LotteryTickets.*,
          (RANDOM(<keycol> + <fractional seconds from clock>))
     FROM LotteryTickets
    GROUP BY spin
   HAVING COUNT(*) = 1;
```

あとは、シンプルなクエリを使えばよい。

```
SELECT *
  FROM LotteryDrawing
 WHERE spin = (SELECT MAX(spin)
                 FROM LotteryDrawing);
```

擬似乱数関数は標準SQLにはないが、非常にポピュラーである。シードにキー列を使えば、おそらく出力にも一意性を保証することができるだろうが、GROUP BY やDISTINCTを使ってもシードの一意性を確保することはできる。システムクロックの小数位を付加するのは、実行するたびに結果が変わるという非決定的な性質を持たせる意味もある。実装によってはこのような動的要素をビュー定義に含めることは構文違反になるかもしれないので、注意してほしい。

いま、0.00 から1.00 の範囲の乱数を返すRANDOMという関数があると仮定しよう。数値列をキーとするテーブルから1行をランダムに

選択したい場合、MAXとMINを使うことでその間にあるランダムな数値を計算することができる。次のクエリはトム・モローが考えた。

```
SELECT L1.*
  FROM LotteryDrawing AS L1
 WHERE col_1
         = (SELECT MIN(keycol)
                    + ((MAX (keycol) - MIN (keycol)) * RANDOM())
              FROM LotteryDrawing AS L2);
```

これと同じことを、COUNT(*)と自己結合を使うことでも実現できる。

```
SELECT L1.*
  FROM LotteryDrawing AS L1
 WHERE CEILING ((SELECT COUNT(*) FROM LotteryDrawing) * RANDOM())
         = (SELECT COUNT(*)
              FROM LotteryDrawing AS L2
             WHERE L1.keycol <= L2.keycol);
```

ここで、0から遠い方向に切り上げているのが重要な点だ。なぜなら、これによって実質的に行を1番からナンバリングすることになるからだ。もっとも、SQLで自己結合を使うのは高コストであるため、この解は実行時間の点ではあまりよくない。

35.3

包含演算子

集合論では部分集合を表す記号は2 種類ある。1 つがA ⊂B である。これは集合A が集合B に含まれることを意味する。このような部分集合Aは真部分集合と呼ばれる。もう1つは、A⊆B である。これはA はBに含まれるかまたは等しいという意味だ。このような集合A は単純に部分集合と呼ばれる。また、こちらの演算子を包含演算子と呼ぶ。

標準SQLは、テーブルに対してこのような包含性や同値性をテストするための演算子を持ったことはない。大学で使うリレーショナルデータベース理論の教科書の中には、CONTAINS述語に言及しているものがあるが、これは標準SQLには存在しない。この述語は、SQLの最初の実験的実装であるIBMのSystem Rには存在していたが、実行コストの問題で以後の実装からは削除されたのだ。

35.3.1　真部分集合演算子

IN 述語は、包含演算子とは異なり、要素が集合に属しているかをテストする述語である。学校の集合論の授業で起きていたなら、この演算子をa∈A のように記述することを覚えているだろう。これはあくまで1つの要素を対象に行うテストなので、集合を対象に行うテストとは異なる点に注意してほしい。

部分集合述語の例として、部署#5における全プロジェクトに関与している社員の名前を選択するクエリを考えよう。System Rの構文を使うとこうなる。

```
SELECT emp_name -- 妥当なSQLではない！
  FROM Personnel
 WHERE (SELECT project_nbr
          FROM JobAssignments
         WHERE Personnel.emp_nbr = JobAssignments.emp_nbr)
           CONTAINS
             (SELECT project_nbr
                FROM Projects
               WHERE dept_nbr = 5);
```

2番目のサブクエリのSELECT文は、部署#5内の全プロジェクトの集合を作っている。最初のサブクエリのSELECT 文は、各社員が働くすべてのプロジェクトの集合を相関サブクエリで作っている。もし最初のサブクエリのSELECT文の作るテーブルが部署#5と等しいか、部署#5を部分集合として含むなら、この述語はTRUEを返す。

このとき注意が必要なのが、両方のテーブルに含まれる重複行の扱いだ。つまり、集合{a, b, c}は重複集合{a, b, b}を包含するのだろうか？ UNIONなどSQLの集合演算子は、重複行を残すか排除するかをオプションで選択する仕様になっている。私の見解としては、CONTAINSにおいては重複行は無視すべきだと思う。つまり、集合{a, b, c}は重複集合{a, b, b}を包含すると考えるべきだ。

社員テーブルの例をもう一度使って、社員のテーブルとボウリングのチームメンバーのテーブルがあるとしよう。後者のテーブルは、社員テーブルの部分集合でなければならない。これは、ボウリングチームのメンバーが社員の集合に包含されるということであり、二重否定で表現すれば「社員でないボウラーはいない」ということだ。我々はすでにこれをSQL で表現する方法を知っている。

```
NOT EXISTS
(SELECT *
  FROM Bowling AS B1
 WHERE B1.emp_nbr NOT IN (SELECT emp_nbr FROM Personnel))
```

このように、包含演算子を要素の所属を調べる述語に還元することも可能だ。

35.3.2 テーブルの相等性

2つのテーブルが互いに等しいかどうかを調べるにはどうすればよいだろう？ これはよくあるプログラミングの問題であり、問題の意味は明らかだ。

2つの集合AとBがあるとき、もし両者が等しければ、以下の性質も満たしている。

- 性質1　どちらも要素数が等しい。
- 性質2　集合Aに存在しない要素は集合Bにも存在しない。
- 性質3　集合Bに存在しない要素は集合Aにも存在しない。
- 性質4　集合Aは集合AとBの交差と等しい。
- 性質5　集合Bは集合AとBの交差と等しい。
- 性質6　集合Bは集合Aの部分集合である。
- 性質7　集合Aは集合Bの部分集合である。

SQLで集合の相等性を調べるのは、思ったより簡単ではない。なぜなら、SQLは重複集合を扱わなければならないからだ。それに加えてSQL にはNULL の問題もある。以下のような重複集合があった場合、互いに等しいのはどのペアだろうか？

```
S0 = {a, b, c}
S1 = {a, b, NULL}
S2 = {a, b, b, c, c}
S3 = {a, b, NULL}
S4 = {a, b, c}
S5 = {x, y, z}
```

S0 = S4であることは誰もが納得するだろう。この2 つは重複要素を含まず、NULLも持たない。互いに完全に一致している。また、S5が他のどの集合とも等しくないことに異論のある人もいないだろう。共通する要素を1つも持っていないのだから。だが、そのほかは難しい。もし重複要素を無視すれば、S0 = S2だ。NULLに対して「疑わしきは罰せず」の原則を適用するなら、S0 = S1 とS0 = S3 も成立する。だが、そうするとS1 = S3 も導かれる。これはつまり、NULLを互いに等しいとみなすことになる。

さらに問題をややこしくするのが、特定のキーだけが一致する2 つの行を等しいとみなすか、それともすべての列について一致しなければならないか、という問題だ。これが現実に問題になるのは、2つの微妙に異なるバージョンのソースから得られたデータを比較する場合だ。たとえば、Joe Celko とJoe F. Celko はたぶん同じ人間を指しているのだろうが、ここにJoseph Frank Celkoが加わると、いささか自信がなくなるだろう。いわゆる「名寄せ」と呼ばれる処理では、常にこのような微妙な揺らぎを持つデータに悩まされる。

キーを使ってマッチングする場合の利点は、きちんとした集合（一意でNULLもない）を対象に行えることだ。こうすると人生は非常にシンプルになる。前ページに挙げた、相等性に関する集合の性質のうち、性質2と性質3を利用すれば、次のようなNOT EXISTS述語を使って書ける。

```
...
WHERE NOT EXISTS
       (SELECT *
          FROM A
         WHERE A.keycol
                 NOT IN (SELECT keycol
                           FROM B
                          WHERE A.keycol = B.keycol))
AND NOT EXISTS
     (SELECT *
        FROM B
       WHERE B.keycol
               NOT IN (SELECT keycol
                         FROM A
                        WHERE A.keycol = B.keycol))
```

これは次のように書いてもいい。

```
...
WHERE NOT EXISTS
       (SELECT *
          FROM A
        EXCEPT [ALL]
        SELECT *
          FROM B
         WHERE A.keycol = B.keycol)
UNION
SELECT *
  FROM B
EXCEPT [ALL]
SELECT *
  FROM A
 WHERE A.keycol = B.keycol))
```

EXCEPTでALL オプションを使うことで、重複行をどう扱うか制御できる。一方、性質1、性質4、性質5を組み合わせて使うなら、次のような解になる。

```
...
WHERE (SELECT COUNT(*) FROM A)
      = (SELECT COUNT(*)
           FROM A INNER JOIN B
             ON A.keycol = B.keycol)
  AND (SELECT COUNT(*) FROM B)
      = (SELECT COUNT(*)
           FROM A INNER JOIN B
             ON A.keycol = B.keycol)
```

このクエリは不一致の値のリストを返す。これらを1つの列にまとめるには次のようにする。

```
SELECT DISTINCT COALESCE(A.keycol, B.keycol) AS non_matched_key
 FROM A FULL OUTER JOIN B
   ON A.keycol = B.keycol
 WHERE A.keycol IS NULL
    OR B.keycol IS NULL;
```

ここまで来れば、結果をINTERSECT [ALL]とUNION [ALL]演算子で扱うことができるようになり、望む形に加工できる。

さて、これまで見てきたサンプルは比較できるキーが存在することを前提としていた。もし2つのテーブルがキーを持っていなかったり、全列を比較しなければならない場合はどうすればよいだろう？

もし片方のテーブルがもう一方のテーブルより列数や行数が多ければ、そもそも相等性の定義からいって比較対象にはならない。2つのテーブルが比較可能であるためには、同じ列数かつ同じ位置の列は同じデータ型でなければならない。だが、行数を数えるのは有用

だ。2 人の子どもがいると想像してほしい。どちらもキャンディーの入った袋を持っている。2つの袋が同一であることを調べるために、一方の子どもが1つずつキャンディーを出しながら「君は赤いキャンディーをいくつ持っているの？」とたずねる。もし2人のキャンディーの数が不一致であれば、袋の中身は不一致であるとわかる。赤いキャンディーの数が一致したら、次に緑色のキャンディーについて数える。この例からわかるように、袋が同じでない場合は、数を数えただけでもわかることがある。だが、数が一致したとしても、まだ袋が同じとは限らない。2人が茶色のキャンディーを1つだけ持っていた場合でも、一方のはM&Mで、もう一方はミルクボールかもしれない。

　さて、この考え方を一般化してみよう。2 つのテーブルを1 つの大きなテーブルに統合する。ただし、`x0`という追加の列を付け加える。これは、行がどちらのテーブルから来たかを示すための列だ。

　次に、オリジナルのすべての列に基づくグループを作る。それぞれのグループには、一方のテーブルから来た行ともう片方のテーブルから来た行が含まれる。もし両者の行数が異なれば、一致しなかった行が存在する証拠だ。このやり方は、テーブルに重複行があってもかまわない。つまり、このやり方ならキーを必要としないのだ。これをクエリにしたのが次のものだ。

```
SELECT x1, x2, ..., xn,
       COUNT(CASE WHEN x0 = 'A'
                  THEN 1 ELSE 0 END) AS a_tally,
       COUNT(CASE WHEN x0 = 'B'
                  THEN 1 ELSE 0 END) AS b_tally
  FROM (SELECT 'A', A.* FROM A
        UNION ALL
        SELECT 'B', B.* FROM B) AS X (x0, x1, x2, ..., xn)
 GROUP BY x1, x2, x3, x4, ... xn
HAVING COUNT(CASE WHEN x0 = 'A' THEN 1 ELSE 0 END)
          <> COUNT(CASE WHEN x0 = 'B' THEN 1 ELSE 0 END);
```

導出テーブルXの中身を変えることでさまざまなバリエーションを得ることができる。たとえば、UNION ALLの代わりにUNIONを使えば、各グループの行数はどちらのテーブルも1になる。SELECTの代わりにSELECT DISTINCTを使えば、各グループにおける1テーブルの行数は1になる。

部分集合の相等性

集合の相等性のうち特に有用なバージョンが、同じテーブル内に等しい部分集合を発見するものだ【訳注3】。この等しい部分集合は、集合論で同値類として知られているパーティションを形成する。

デイトの業者-部品テーブルを使って、まったく同じ部品の集合を供給する業者を見つけよう。

```
CREATE TABLE SupParts
(sup_nbr CHAR(2) NOT NULL,
 part_nbr CHAR(2) NOT NULL,
     PRIMARY KEY (sup_nbr, part_nbr));

INSERT INTO SupParts
VALUES ('S1', 'P1'), ('S1', 'P2'), ('S1', 'P3'),
       ('S2', 'P2'), ('S2', 'P3'),
       ('S3', 'P1'), ('S3', 'P2'), ('S3', 'P3'),
       ('S4', 'P1');
```

2 つの集合が等しいことを証明する通常の手段は、集合A が集合Bを含み、かつ、集合B が集合Aを含むことを証明することだ。これまで見てきた方法ならばどれでも使える。ただし、同じテーブルを使うので別名を使う必要がある。

もう1 つ別のアプローチを考えてみよう。まず1 つの業者を別の業者と共通の部品について結合する。このとき、同じ業者同士を結合するのは無意味なのでそのペアは排除しておく。そのうえで、2つの業者の共通集合が、それぞれの業者の部品数と同じであれば、2つの業者は等しい集合である。

```
SELECT SP1.sup_nbr, SP2.sup_nbr, COUNT(*) AS part_count
  FROM SupParts AS SP1
         INNER JOIN
           SupParts AS SP2
    ON SP1.part_nbr = SP2.part_nbr
   AND SP1.sup_nbr < SP2.sup_nbr
 GROUP BY SP1.sup_nbr, SP2.sup_nbr
HAVING COUNT(*) = (SELECT COUNT(*)
                     FROM SupParts AS SP3
                    WHERE SP3.sup_nbr = SP1.sup_nbr)
   AND COUNT(*) = (SELECT COUNT(*)
                     FROM SupParts AS SP4
                    WHERE SP4.sup_nbr = SP2.sup_nbr);
```

結果

```
 sup_nbr  sup_nbr part_count
============================
 S1       S3               3
```

SupPartsテーブルの業者番号にインデックスがあれば、結合が効率化されるだけでなく、ダイレクトにカウントを得ることができる。

【訳注1】　実装依存の乱数生成機能としては、以下のようなものがある。DBMS_RANDOMパッケージ、またはSAMPLE句（Oracle）、RAND関数（SQL Server、DB2、MySQL）、RANDOM関数（PostgreSQL）。

【訳注2】　S. K. Park and K. W. Miller (1988). "Random Number Generators: Good Ones Are Hard To Find." *Communications of the ACM*, 31 (10): 1192–1201.

【訳注3】　この問題に対しては、ほかにも多くの別解がある。詳しくは『SQLパズル』（第2版）のパズル27を参照。

36

SQLで木と階層構造を扱う

Trees and Hierarchies in SQL

SQL での木構造の扱い方について、私は一冊の本を書いた（『Trees and Hierarchies in SQL for Smarties』Morgan Kaufmann, 2004）。このトピックについては同書で詳細に論じたので、ここではSQLで木や階層構造をモデリングする方法論について、主なアプローチについて概観するにとどめる。

木は有向グラフの特殊例である。グラフとは、ノード（普通は箱や円で表す）をエッジ（普通は矢印付きの線で表す）でつないだデータ構造だ。エッジは2 つのノード間の一方向の関係を表す（リンクと呼ばれることもある）。組織図では、ノードは社員の役職でエッジは「～へ報告する」という関係を表す。部品の設計図では、ノードは各部品を、エッジは「～から作られる」という関係を表す。

木の頂点はルートと呼ばれる。組織図ならば社長、設計図では最終製品だ。あるノードから出ていくエッジの本数を出次数、入ってくるエッジの本数を入次数と呼ぶ。

二分木は、1つの親ノードが最大2 つの子ノードしか持つことのできない木である。より一般的に、n 分木は、出次数が最大n の木である。1 つも子ノードを持たないノードはリーフと呼ばれる。設計図では、これ以上分解できない最小単位の部品がこれにあたる。

木を定義する方法は複数ある。たとえば、木は循環しないグラフである。あるいは、木はルート以外のすべてのノードの入力次数が1 で、ルー

トの入力次数が0 のグラフである。あるいは、エッジをたどることで、ルートから任意のノードに対する経路が必ず1 つ見つけられるグラフ、というのも木の定義だ。

　木構造とノードの集まりは同じ概念ではない。それゆえ、本来はそれぞれ別テーブルとしてモデリングするのが正しい。しかし、本章ではこのルールを破り、図36.1のような抽象的な木を使うことにする。

　この小さな木は、小さいがゆえに本章を読んでいる間も読者の記憶にとどまるだろう。かつ、本章で紹介するさまざまなテクニックを試すにも十分な大きさだ。本章では、子ノード、親ノード、および中間ノードという語を使うが、他のグラフ理論の本では別の用語で呼ばれているかもしれないので注意してほしい。

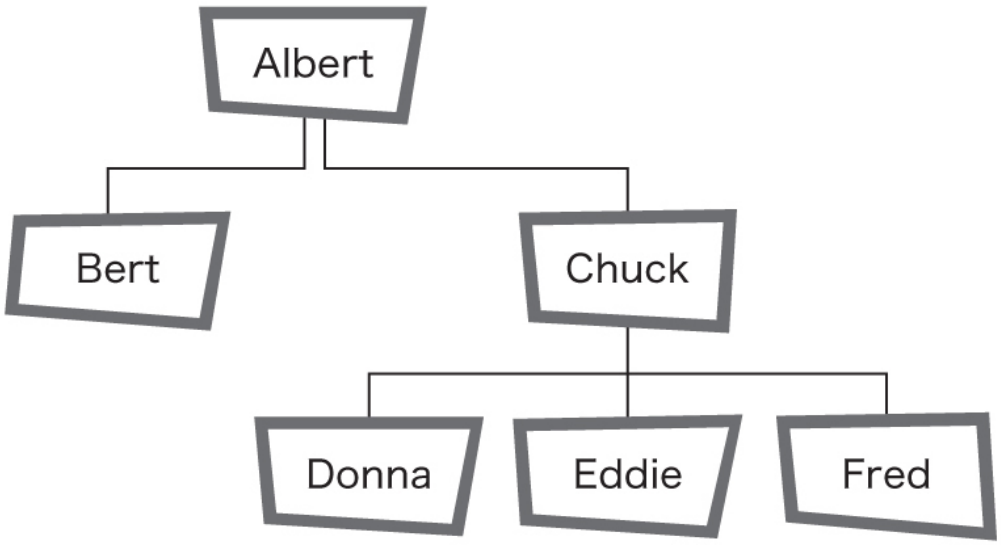


図36.1　抽象的な木

36.1

隣接リストモデル

ほとんどのデータベースは、木構造を扱うために隣接リストモデルを使ってきた。これには2 つの理由がある。1 つは、コッドがリレーショナルモデルの創成期に考えついたモデルであり、これに代わるモデルがその後長く出てこなかったこと。2 つ目の理由は、隣接リストモデルがポインタチェーンを擬似的に表現したもので、この手続き型言語における伝統的なプログラミング技法が、多くのエンジニアにわかりやすかったからだ。このモデルは、エッジを「箱と矢印」の手続き的なダイアグラムとして描写する。サンプルテーブルは次のような形になる。

```
CREATE TABLE AdjTree
(child_node CHAR(2) NOT NULL,
 parent_node CHAR(2), -- NULLはルートを意味する
     PRIMARY KEY (child_node));
```

AdjTree：隣接リストテーブル　　child_node：子ノード　　parent_node：親ノード

AdjTree

child_node	parent_node
A	NULL
B	A
C	A
D	C
E	C
F	C

ルートとリーフを求めるクエリは明らかだ。ルートはparent_node列がNULLの行、リーフは下位ノードを持たないノードだ。各行は、方向を持った隣接エッジを共有する2つのノードを表している。隣接リストモデルは、最もよく使われている方法であると同時に、最悪の方法でもある——一般的なグラフを表すには、これ以上の方法はないのだが。

36.1.1 複雑な制約

隣接リストモデルの最初の問題は、データ整合性を保つために複雑な制約が必要になることだ。実際、現実的な解決策としては、この問題に目をつむり、構造を破壊するような更新が行われないよう祈ることしかない。しかし、もしデータ整合性にこだわるなら、次のような制約が必要になる。

1. ルートは1 つしか存在しない。これを満たす制約を書くのは、大半の実装では難しい。なぜならCHECK制約の中でサブクエリか集約関数を書かねばならないからだ。

```
CREATE TABLE AdjTree
(child_node CHAR(2) NOT NULL,
 parent_node CHAR(2),
     PRIMARY KEY (child_node),
     CONSTRAINT one_root
       CHECK((SELECT COUNT(*)
                FROM AdjTree
               WHERE parent_node IS NULL) = 1)
 ...);
```

2. 循環があってはならない。残念なことに、これはトリガーを使わなくては実装できない。すべての経路を探索して循環をチェックするようなトリガーを書くのである。だが、これよりは弱いが、明らかな単純循環を禁止する制約ならば次のように書くことができる。

```
CHECK (child_node <> parent_node) -- 自分自身の親であることはできない
```

だがこれは、2個以上のノードが関係する循環は見つけられない。

木におけるエッジ数はノード数から1を引いたものである。これは連結グラフの特徴だ。これは次のような制約でチェックできる。

```
CHECK ((SELECT COUNT(*) FROM AdjTree) -1 -- エッジ
     = (SELECT COUNT(parent_node) FROM AdjTree)) -- ノード
```

`COUNT(parent_node)`はルートの`NULL`を除外する。したがって、次の制約によって`NULL`が1つしかないことをチェックできる。

```
CHECK((SELECT COUNT(*) FROM Tree WHERE parent_node IS NULL) = 1)
```

これらの制約は循環がないことの必要条件だが、十分条件にはなっていない。次のデータを見てほしい。`'D'`と`'E'`が循環しているため、これは木構造ではない。

Cycle

child_node	parent_node
A	NULL
B	A
C	A
D	E
E	D

1つのアプローチは、すべてのリーフを削除し、木が空集合になるまで以下のプロシージャを繰り返すことだ。もし木が空集合に行き着かなければ、結合されていない循環が存在する。

```
CREATE PROCEDURE TreeTest
IS
BEGIN
DECLARE
    row_count INTEGER;
    tmp_row_count INTEGER;
BEGIN
    SELECT COUNT(DISTINCT parent_node) + 1 INTO row_count
      FROM AdjTree;

    --一時テーブルにデータをコピー
    INSERT INTO WorkTree
    SELECT child_node, parent_node FROM AdjTree;

    WHILE (row_count > 0) LOOP
        DELETE FROM WorkTree  -- リーフを削除する
         WHERE child_node
                 NOT IN (SELECT T2.parent_node
                           FROM WorkTree T2
                          WHERE T2.parent_node IS NOT NULL);

        row_count := row_count -1;
    END LOOP;

END;
END TreeTest;
```

このプロシージャを実行した結果、`WorkTree`が0行ならば循環がないことがわかる。

36.1.2　手続き的な木の探索

2 つ目の問題は隣接リストモデルにおいては、何らか意味のある問い合わせ、たとえば「キング氏はジョーンズ氏の（直接でないにせよ）上司か？」のような質問に答えたり、集計を行うには、ノードからノードへの探索が必要になることだ。あるのノードの直接の親ノードを求めるクエリは次のようになる。

```
SELECT P1.child_node AS parent_node,
       ' parent_node to ',
       C1.child_node
  FROM AdjTree AS P1, AdjTree AS C1
 WHERE P1.child_node = C1.parent_node;
```

結果

```
parent_node   parent_node to  child_node
=========================================
 A            parent_node to  B
 A            parent_node to  C
 C            parent_node to  D
 C            parent_node to  E
 C            parent_node to  F
```

だがこのクエリだけではすべての問題には対処できない。このクエリは、ノードの直接の`parent_node`を与えるにすぎず、「上司の上司」は教えてくれない。2レベル以上さかのぼるには、より複雑な自己結合が必要になる。

```
SELECT B1.child_node AS parent_of_parent,
       ' parent_node to ',
       E2.child_node
  FROM AdjTree AS B1, AdjTree AS E1, AdjTree AS E2
 WHERE B1.child_node = E1.parent_node
   AND E1.child_node = E2.parent_node;
```

結果

```
parent_node   parent_node to  child_node
=========================================
 A            parent_node to  D
 A            parent_node to  E
 A            parent_node to  F
```

ただまずいことに、一般にどれくらいの深さをたどれば十分かは事前にわからない。しかも、現実問題として、大半のSQLコンパイラは、結合するテーブル数が多くなると、最適化に深刻な問題を抱えている。

別の方法として、カーソルを宣言して手続き型のコードで木を探索する方法がある。これは普通は実用に堪えないほど遅いが、任意の深さの木に対応できるという柔軟さがある。これは、SQLにはない手続き型の利点である。

SQLでも、再帰共通表式を使えば、同様に任意の深さの木をたどることのできるコードを書くことができる。この機能も性能面では、あとで紹介する入れ子集合モデルに比べればほめられたものではない。以下に単純な再帰共通表式のコードを示す。これは各ノードの深さを求める。

```
WITH RECURSIVE PersonnelTraversal
AS -- ルートから始める
(SELECT child_node,
        1 AS hierarchy_level
   FROM AdjTree
  WHERE parent_node IS NULL
 UNION ALL
 SELECT E.child_node, -- 各レベルを追加する
        (X.hierarchy_level + 1) AS hierarchy_level
   FROM AdjTree AS E,
        PersonnelTraversal AS X
  WHERE X.child_node = E.parent_node)
 SELECT child_node, hierarchy_level
   FROM PersonnelTraversal;
```

結果

```
child_node  hierarchy_level
===========================
 A                        1
 B                        2
 C                        2
 D                        3
 E                        3
 F                        3
```

以上が隣接リストモデルにおける基本的な計算のイロハである。

36.1.3　テーブルの更新

隣接リストモデルでは、新しいノードを追加するのは簡単である。単純にINSERT INTO 文を実行して、parent_nodeがテーブルに存在していることをチェックすればよい。

一方、ノードの削除には注意が必要だ。木の中間のエッジを削除すると、複数の分離された木の集合（「森」とでも呼ぼうか）になる。したがって、構造を再構築するためのルールが必要になる。一般的に取られる方法は2つある。

1. 空いたポストに部下を昇進させる。
2. 削除されたノードの親ノードに子ノードをすべて連結させる（孤児は祖父母と暮らす）。

中間のノードが変更された場合は、親ノードと子ノードの両方を変更しなければならない。たとえば、社員「C」が削除されてから、残った部下をルール2に従って上司「A」の直属とするにはこう書く。

```
DELETE FROM AdjTree
 WHERE child_node = 'C';

UPDATE AdjTree
   SET child_node
         = CASE WHEN child_node = 'C'
                THEN 'A'
                ELSE child_node END,
       parent_node
         = CASE WHEN parent_node= 'C'
                THEN 'A'
                ELSE parent_node END
 WHERE 'C' IN (parent_node, child_node);
```

結果

```
child_node parent_node
=====================
A
B           A
D           A
E           A
F           A
```

36.2

経路列挙モデル

SQLで階層構造を表す2つ目のモデルは、経路列挙モデルである。これを初めて論じたのは、ステファン・グスタフソンである。後に、トム・モローとイツァク・ベン-ガンが『Advanced Transact-SQL for SQL Server 2000』（Apress, 2000）で詳細に展開した。このモデルは、ルートから各ノードへの経路を文字列として保存する。

これは厳密に見れば非正規化テーブルである。経路はスカラ値ではないからだ。更新性能が最悪になるのは、ルートを更新するときである。ルートはすべてのノードの経路に含まれるので、木全体の経路を再計算しなければならないからだ。だが、もし上位レベルのノードが頻繁に更新されないのであれば、このモデルは活躍の場がある。テーブル定義は次のようになる。

```
CREATE TABLE PathTree
(node CHAR(2) NOT NULL PRIMARY KEY,
 path VARCHAR (900) NOT NULL);
```

PathTree：経路列挙テーブル　　node：ノード　　path：経路

データのサンプルはたとえば次のようになる。

node	path
a	a/
b	a/b/
c	a/c/
d	a/c/d/
e	a/c/e/

f	a/c/f/

経路とは要するに、ノードの名前をスラッシュを区切り文字として連結したものである。そのため、すべての演算は文字列操作を使うことになる。経路の長さを短く保つためには、ノードの名前も短いものにしておくほうがよい。区切り文字は任意のものが使用できるが、スラッシュがUNIX/Linux系のOSで使用されるため、わかりやすいだろう。

36.2.1　部分木とノードを見つける

経路列挙モデルのクエリでは、LIKE述語を多用する。:mynodeに属する下位の木を見つけるには、以下のクエリを使う。

```
SELECT node
  FROM PathTree
 WHERE path LIKE '%' || :my_node || '%';
```

ルートを見つけるのは、簡単だ。スラッシュを1 つしか含まないノードである。一方、リーフを見つけるクエリはもう少し複雑だ。

```
SELECT T1.node
  FROM PathTree AS T1
 WHERE NOT EXISTS
        (SELECT *
           FROM PathTree AS T2
          WHERE T2.path LIKE T1.path || '_%');
```

結果

```
 node
=====
 b
 d
 e
 f
```

これは、「自らの経路を一部に含むような経路を持つノードが存在しない」という条件を表現している。

36.2.2　木の深さと部分木を見つける

ノードの深さはパスに含まれるスラッシュ（/）の数でわかる。REPLACE関数が使えれば、経路の文字列からスラッシュを取り除いた文字列長と元の経路の文字列長を比較することで、深さを求められる。

```
SELECT node, path,
       CHAR_LENGTH (path)
         - CHAR_LENGTH (REPLACE (path, '/', ''))
  FROM PathTree;
```

結果

```
 node  path  level
====================
 a     a/          1
 b     a/b/        2
 c     a/c/        2
 d     a/c/d/      3
 e     a/c/e/      3
 f     a/c/f/      3
```

直属の親ノードは次のクエリで見つけられる。

```
SELECT T1.node AS child, T2.node AS parent
  FROM PathTree AS T1, PathTree AS T2
 WHERE T2.path =(SELECT MAX(path)
                   FROM PathTree
                  WHERE T1.path LIKE path || '_%');
```

結果

```
child parent
============
 b    a
 c    a
 d    c
 e    c
 f    c
```

36.2.3　ノードと部分木の削除

ノードを削除するとき、ノード配下のすべての経路を更新しなければならないというのは、一見すると奇妙に感じると思う。サンプルのノードbを削除するには次のようにする。

```
BEGIN
UPDATE PathTree
   SET path = REPLACE (path, 'b/', '')
 WHERE POSITION ('b/' IN path) > 0;

DELETE FROM PathTree
 WHERE node = 'b';
END;
```

結果

```
node  path
==============
 a    a/
 c    a/c/
 d    a/c/d/
 e    a/c/e/
 f    a/c/f/
```

:mynodeをルートとする部分木を削除するのはずっと簡単だ。

```
DELETE FROM PathTree
 WHERE path LIKE (SELECT path
                    FROM PathTree
                   WHERE node = :my_node) || '%';
```

36.2.4　整合性制約

もし経路の中に同じノードが2 回現れることがあれば、それは循環が存在する証拠である。これを防止するには次のような制約が有効だ。

```
CHECK (NOT EXISTS
        (SELECT *
           FROM NodeList AS D1, PathTree AS P1
          WHERE CHAR_LENGTH (REPLACE (D1.node, P1.path, ''))
                   < (CHAR_LENGTH(P1.path) - CHAR_LENGTH(D1.node))))
```

ただ残念ながら、制約の中でサブクエリを使うことは、まだ一般的な機能ではない。

36.3

入れ子集合モデル

SQLは集合指向言語である。ここでは、その利点を活かした優れたモデルを紹介する。もし読者がHTML やXMLといったブロック構造の言語を使ったことがあれば、このモデルの基本的なアイデアを理解するのは簡単だ。`lft`列と`rgt`列は、木のXML 表現における「タグ」の数である。

互いに重ならない円が入れ子になっている様子を想像してほしい。ちょうどマークアップ言語の構造を描くときと同じように。この入れ子構造には、クエリを書くうえで利用できる非常に明快な性質がある。この心理的モデルがしっくり来なければ、「箱と矢印」のグラフを入れ子集合に変換してみるといい。小さな虫が木を這っていると想像する。虫は木の頂点から出発し、木を隅々まで1 周して戻ってくる。ノードに到達すると、虫はその箱の訪れた側の脇に番号を打っていく。すると、すべてのノードが左右に1つずつの番号を持つことになる。コンピュータ科学専攻だった人ならば、これが修正先行順の探索アルゴリズムを再構成したものであることに気づくだろう。

```
CREATE TABLE NestTree
(node CHAR(2) NOT NULL PRIMARY KEY,
 lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),
 rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 1),
     CONSTRAINT order_okay CHECK (lft < rgt));
```

NestTree：入れ子集合テーブル　　node：ノード　　lft：左端　　rgt：右端

NestTree

node	lft	rgt
A	1	12
B	2	3

C	4	11
D	5	6
E	7	8
F	9	10

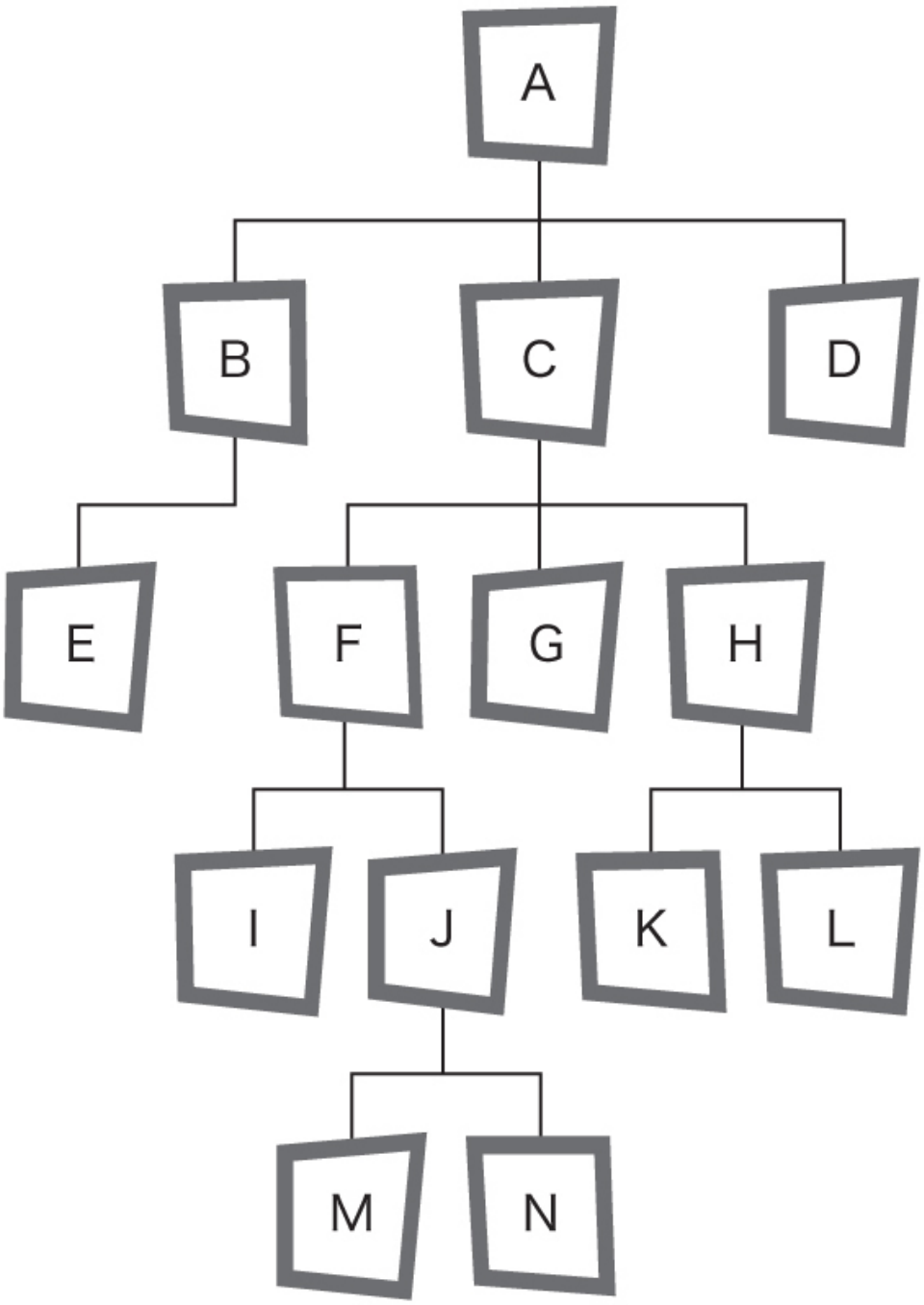


図36.2　伝統的な「箱と矢印」による木の表現

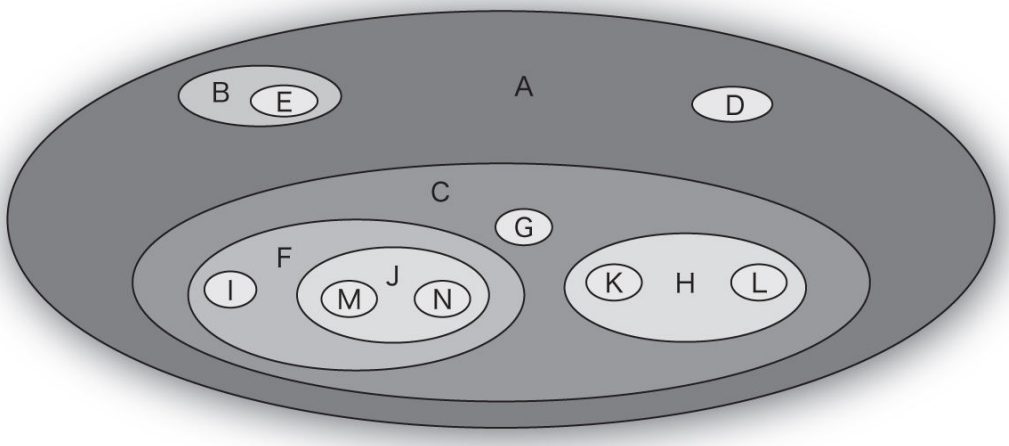


図36.3　入れ子集合による木の表現

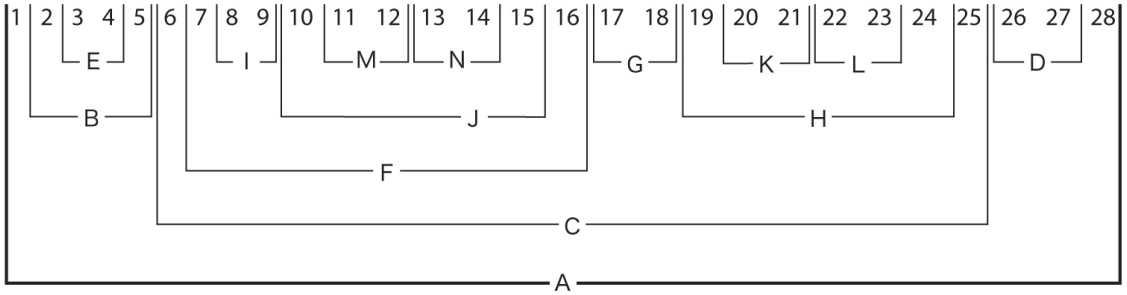


図36.4　線形の入れ子集合による木の表現

このモデルの利点の1つは、すべてのノードがテーブルに一度しか現れないことだ。経路列挙モデルや隣接リストモデルは、多くのノードに対して多くの自己参照を持っており、これが更新を複雑にする要因になっていた。

36.3.1　座標からわかる情報

`lft`と`rgt`の数字は、部分木の位置と性質についての重要な情報を持っている。ルートは常に`(lft, rgt)=(1, 2 *(SELECT COUNT(*) FROM TreeTable))`であり、リーフは常に`(lft + 1 = rgt)`である。

```
SELECT node AS root
  FROM NestTree
 WHERE lft = 1;
```

結果

```
 root
=====
 A
```

```
SELECT node AS leaf
  FROM NestTree
 WHERE lft = (rgt - 1);
```

```
 leaf
=====
 B
 D
 E
 F
```

座標からわかるもう1つの重要な情報は、木の任意のノードが部分木のルートになっているということであり（リーフは部分木を持っていないと思うかもしれないが、これは縮退したケースにすぎない）、そのサイズは`(rgt-lft +1)/2`で求められる。

36.3.2 包含関係

入れ子集合モデルのテーブルでは、あるノードにぶら下がるすべての子ノード、孫ノードたちは、親ノードの`lft`と`rgt`の数字の範囲内に収まる`lft`と`rgt`の値を持つノードを探すことで見つけられる。たとえば、組織図におけるすべての上下関係を明らかにするクエリは次のようになる。

```
SELECT Superiors.node, ' is a boss of ', Subordinates.node
  FROM NestTree AS Superiors, NestTree AS Subordinates
 WHERE Subordinates.lft BETWEEN Superiors.lft AND Superiors.rgt;
```

結果

```
 node is a boss of node
=======================
 A    is a boss of A
 A    is a boss of B
 A    is a boss of C
 A    is a boss of D
 A    is a boss of E
 A    is a boss of F
 B    is a boss of B
 C    is a boss of C
 C    is a boss of D
 C    is a boss of E
 C    is a boss of F
 D    is a boss of D
 E    is a boss of E
 F    is a boss of F
```

BETWEEN 述語を使っているので、自分も自分の上司であるとみなされる。これを除外したい場合は、不等号に変更すれば良い。このシンプルな自己結合は、入れ子集合モデルでのクエリの基礎となる。この包含関係を調べるクエリは`lft`と`rgt`に欠番があっても正しく動く。

ノードの深さは、ルートからノードへのエッジの本数によって求められる。これが大きいほどレベルが深く、ルートから遠いことを意味する。経路は、2 つのノードを直接結ぶエッジの集合である。2つのノード間のエッジの本数を数えるのは簡単だ。たとえば、次のように書けばよい。

```
SELECT T2.node, (COUNT(T1.node) - 1) AS level
  FROM NestTree AS T1, NestTree AS T2
 WHERE T2.lft BETWEEN T1.lft AND T1.rgt
 GROUP BY T2.node;
```

結果

```
 node level
===========
 A        0
 B        1
 C        1
 D        2
 F        2
 E        2
```

(COUNT(*)-1)という式を使う理由は、ノード自身を重複カウントしないようにするためだ。このため、レベルは0から開始される。もし1から開始するほうがよければ、単純にCOUNT関数を使うだけでよい。

入れ子集合モデルでは、あるノードを包含するノードは、包含されるノードより「広い」と考えられる（その広さは(rgt-lft)で求められる）。当たり前だが、ルートは他のどのノードよりも広い。

36.3.3　同レベルのノード間の序列

入れ子集合モデルは通常、同じレベルのノード間に、年齢、序列といった何らかの線形の順序が存在することを前提としている。隣接リストモデルはそのような順序は前提としていない。したがって、隣接リストモデルでは、次のクエリで得られるような結果を得るには、追加の列が必要になる。これは、最も年上の子ノードを見つけるクエリだ（一番「左」に位置するノードを求めている）。

```
SELECT Subordinates.node, ' is the oldest child_node of ', :my_node
  FROM NestTree AS Superiors, NestTree AS Subordinates
 WHERE Superiors.node = :my_node
   AND Subordinates.lft - 1 = Superiors.lft; -- 一番左の子ノード
```

最も年下の子ノードはこうだ。

```
SELECT Subordinates.node, ' is the youngest child_node of ', :my_node
  FROM NestTree AS Superiors, NestTree AS Subordinates
 WHERE Superiors.node = :my_node
   AND Subordinates.rgt = Superiors.rgt - 1; -- 一番右の子ノード
```

入れ子集合モデルを隣接リストモデルに変換するには、次のクエリを使えばよい。

```
SELECT E.node, B.node
  FROM NestTree AS E
         LEFT OUTER JOIN
           NestTree AS B
    ON B.lft
         = (SELECT MAX(lft)
              FROM NestTree AS S
             WHERE E.lft > S.lft
               AND E.lft < S.rgt);
```

結果

```
node node
=========
A
B    A
C    A
D    C
E    C
F    C
```

36.3.4　階層集約

各ノードのレベルを見やすくする方法の1つは、木をインデント付きで表示することだ。

```
SELECT COUNT(T2.node) AS indentation, T1.node
  FROM NestTree AS T1, NestTree AS T2
 WHERE T1.lft BETWEEN T2.lft AND T2.rgt
 GROUP BY T1.lft, T1.node
 ORDER BY T1.lft;
```

結果

```
indentation node
================
           1 A
           2 B
           2 C
           3 D
           3 E
           3 F
```

このクエリをベースに、他の集約関数を使うと、いろいろな意味の結果が得られる。各ノードの重みを保持する別のテーブルNodeWeights があるとしよう。すると階層的な重みの合計は2つのテーブルを結合すれば得られる。

```
SELECT Superiors.node, SUM (Subordinates.weight) AS subtree_weight
  FROM NestTree AS Superiors, NestTree AS Subordinates, NodeWeights AS W
 WHERE Subordinates.lft BETWEEN Superiors.lft AND Superiors.rgt
   AND W.node = Subordinates,node;
```

36.3.5　ノードと部分木を削除する

入れ子集合モデルの面白い性質の1つは、部分木もまた左から右へ線形の順序を持つことだ。言い換えると、parent_nodeは右側にchild_nodeを持ちながら、左側には空白地帯が広がっていることもありうる。この状況では、parent_nodeの最も左側の子ノードを挿入する余地がある。このポジションにいる人間が次に昇格する人物である、というような意味づけも可能になる。

木の内部の単一のノードを削除するのは、部分木を削除するよりも概念的に難しい。ノードを削除すると、その穴をどうやって埋めるかを考えなければならない。これには2つの方法がある。

1つ目は、子ノードの1つを削除されたノードのポストに昇格させる方法――父が死んだら子が事業を引き継ぐ。2 つ目は、子ノードをもう1 つ上のノードに直結する方法――母が死んだら祖母が子を引き取る。ちなみ

にこちらは入れ子集合モデルにおけるデフォルト動作だ。単純にノードを削除すると自然にこういう結果になる。

削除すると、座標に欠番が生じる。実際のところ、欠番が生じても大きな問題はないのだが、これを埋めるには、ナンバリングをやり直す必要がある。

```
UPDATE NestTree
   SET lft = (SELECT COUNT(*)
                FROM (SELECT lft FROM NestTree
              UNION ALL
              SELECT rgt FROM NestTree) AS LftRgt (seq_nbr)
               WHERE seq_nbr <= lft),
       rgt = (SELECT COUNT(*)
                FROM (SELECT lft FROM NestTree
                      UNION ALL
                      SELECT rgt FROM NestTree) AS LftRgt (seq_nbr)
               WHERE seq_nbr <= rgt);
```

もし導出テーブル`LftRgt`のパフォーマンスが悪ければ、一時テーブルやインデックスで性能改善を行うことができる。または、ビューを実体化する方法もある。

```
CREATE VIEW LftRgt (seq_nbr)
AS SELECT lft FROM NestTree
   UNION
   SELECT rgt FROM NestTree;
```

このビューを使うと、木が持つべきすべての`lft`と`rgt`の値を持っているかどうかをチェックできる。

```
SELECT *
  FROM NestTree
 WHERE NOT EXISTS
        (SELECT *
           FROM (SELECT seq_nbr,
                        ROW_NUMBER()
                          OVER (ORDER BY seq_nbr) AS rn
                   FROM LftRgt) AS X
          WHERE seq_nbr <> rn);
```

36.3.6　隣接リストモデルを入れ子集合モデルに変換する

隣接リストモデルのテーブルへデータをロードするのは、ホスト言語のプログラムを使えば比較的簡単である。隣接リストモデルのテーブルから入れ子集合モデルのテーブルにデータを変換するには、大学一年生が習う木の再帰探索アルゴリズムを使うことになる。ストアドプロシージャで記述すると次のようになる。

```
-- 隣接リストモデル
CREATE TABLE Tree
(node CHAR(10) NOT NULL,
 parent CHAR(10));

-- 入れ子集合モデル（最初は空テーブル）
CREATE TABLE Stack
(stack_top INTEGER NOT NULL,
 node CHAR(10) NOT NULL,
 lft INTEGER,
 rgt INTEGER);

CREATE PROCEDURE TreeTraversal
AS
    counter     INTEGER :=2;
    max_counter INTEGER :=0;
    current_top INTEGER :=1;
    current_cnt INTEGER :=0;
BEGIN

    --カウンタの最大値を初期化（1つのノードについて二度巡回するので2倍している）
    SELECT COUNT(*) * 2 INTO max_counter FROM Tree;

    --スタックの初期化
    DELETE FROM Stack;

    --ルートをプッシュ
    INSERT INTO Stack
    SELECT 1, node, 1, max_counter
```

```
      FROM Tree
     WHERE parent IS NULL;

    --プッシュしたノードを元の木から削除
    DELETE FROM Tree WHERE parent IS NULL;

    WHILE counter <= max_counter- 1
    LOOP

            --プッシュしたノードが子を持つか調べる
            SELECT COUNT(*) INTO current_cnt
              FROM Stack S1, Tree T1
             WHERE S1.node = T1.parent
               AND S1.stack_top = current_top;

        IF current_cnt > 0 THEN
            -- ノードが子を持っていれば、子を1つプッシュする
            INSERT INTO Stack
            SELECT (current_top + 1), MIN(T1.node), counter, NULL
              FROM Stack S1, Tree T1
             WHERE S1.node = T1.parent
               AND S1.stack_top = current_top;

            -- プッシュしたノードを元の木から削除
            DELETE FROM Tree
             WHERE node = (SELECT node
                             FROM Stack
                            WHERE stack_top = current_top + 1);

            -- カウンタとスタック・ポインタのインクリメント
            counter := counter + 1;
            current_top := current_top + 1;
        ELSE
            -- スタックをポップして、rgtの値を決める
            UPDATE Stack
               SET rgt = counter,
                   stack_top = -stack_top -- スタックのポップ
             WHERE stack_top = current_top;

            -- カウンタはインクリメントするが、スタック・ポインタはデクリメント
            counter := counter + 1;
            current_top := current_top - 1;
        END IF;
    END LOOP;
END;
```

これは必ずしも最速の変換アルゴリズムではないが、この変換はそれほど頻繁に行うタスクではないだろう。

36.4

木と階層構造を表現するその他のモデル

その他の木構造に対するモデルについては、拙著『Trees and Hierarchies in SQL for Smarties』（Morgan Kaufmann, 2004）で論じた。2分木のような特殊な木に対しては、特殊なモデルを使用することができるが、本章で取り上げた3 つがメジャーどころなのは間違いない。本当の問題は、SQL で階層構造を扱うには、行おうとしているタスクに適切な1つのモデルを選択することである。以下、選択の指針をいくつか挙げる。

指針1

ノード更新は多いが、構造の変更は滅多に行われないケース。たとえば、組織図であれば人の出入りや異動が頻繁だが、組織はそのままであるケース。

指針2

ノード更新は多くないが、構造の変更が多い場合。たとえば、Eメールのメッセージボードでは、既存のノードは変化せず、新しいメールだけが単純に追加されていく。

指針3

ノード更新も構造の変更も多くない場合。たとえばDWHにおける履歴データ。

指針4

ノード更新と構造の変更がともに頻繁なケース。たとえば、中央のディスパッチセンターから、現在最も優先度の高いノードへの最適パスを見つけようとする地理システム。具体的には、消防車が最もひどい火災の現場へ向かうための経路を、リアルタイムに混雑状況を見ながら決めるときがそうだ。

　こうした特定の状況に対して、どのモデルを採用すべきか、という質問にはここでは答えない。あえて答えるならこうだ――「そうですね、それは状況によるので一概には言えませんね……」

37

SQLにおけるグラフ

Graphs in SQL

グラフ理論の用語は直観的にとてもわかりやすい。点を線でつないだこの概念は、とてもシンプルだ。しかしグラフ理論はその見かけに反して極めて強力で、数学的に複雑な性質を持っている。もしこの理論について習ったことがなく、興味がある人は、本章の最後に挙げた参考文献にあたってみてほしい。

グラフが重要なのは、これがさまざまなタイプのデータとその関連を表すことのできる一般的な方法だからだ。たとえば次のようなものがそうだ。

- ソーシャルネットワーク
- 友達の友達
- 距離の次数
- 道路混雑時の経路選択

グラフ理論で使われる用語について、簡単に解説しておこう。これらの用語は、論者によって揺らぎがあるかもしれないので注意してほしい。まず、グラフとはエッジによって連結されたノードからなるデータ構造である。エッジは方向を持っている場合も（一方向にだけ行くことができる）持っていない場合（両方向に行くことができる）もある。方向を持ったエッジを使うグラフを有向グラフ、そうでないものを無向グラフと呼ぶ。あるノードから出ていくエッジの本数を出次数、入ってくるエッジの本数を入次数と呼ぶ。

あるノードから別のノードへたどることのできるエッジの集合を経路と呼ぶ。始点と終点が同じで、かつ途中で交わらない経路のことを閉路と呼ぶ（回路や循環とも呼ぶ）。アルファベットのOは閉路だが、数字の8は閉路ではない。

木は特に重要な性質を持ったグラフであるため、これだけで1 つの研究分野として成立しており、用語や概念も特有なものを持っている。本章では、木以外の一

般的な有向グラフのうち、有用なものに焦点を当てていく。

37.1

グラフを隣接リストモデルで表す

SQLでグラフをモデル化するために最もよく使われる方法は、隣接リストモデルである。グラフのエッジはノードのペアとして表現される。エッジのその他の属性は、そのための列に保持する。以下に、グラフの基本的な隣接リストモデルのスケルトンを示す。

```
CREATE TABLE Nodes
(node_id INTEGER NOT NULL PRIMARY KEY,
 « other attributes of the node »);
```

Nodes：ノードテーブル　　node_id：ノードID

```
CREATE TABLE AdjacencyListGraph
(begin_node_id INTEGER NOT NULL
     REFERENCES Nodes (node_id)
       ON DELETE CASCADE,
 end_node_id INTEGER NOT NULL
     REFERENCES Nodes (node_id)
       ON DELETE CASCADE,
 « other attributes of the edge »,
     PRIMARY KEY (begin_node_id, end_node_id));
```

AdjacencyListGraph：隣接リストグラフテーブル　　begin_node_id：開始ノードID　　end_node_id：終了ノードID

技術的には、`begin_node_id` は`end_node_id` と同じであってもかまわない。また、エッジを持たないノードも存在してもよい。この2つは、図37.1のように非常にシンプルな形になる。

エッジが持つ属性は、通常は重みと呼ばれる。たとえば、地図でいえばノード間の距離や、移動にかかる時間、電気回路で言えば抵抗、ワークフローであれば処理のコストを意味する。この属性は普通は計算しやすいよう数値で表す。

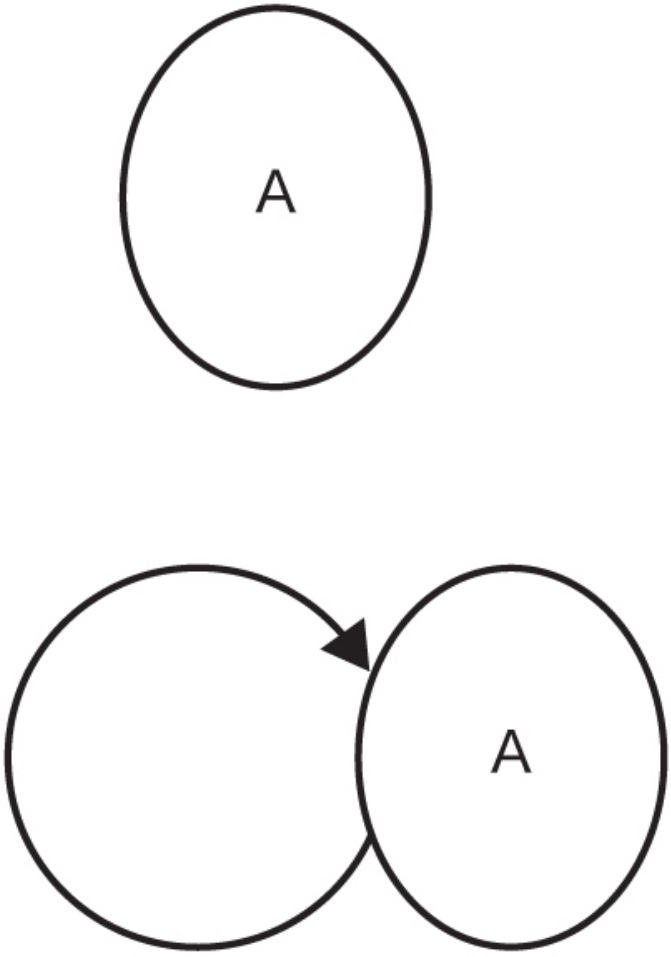


図37.1　エッジを持たないノードと自分へ戻るノード

37.1.1　SQL と隣接リストモデル

グラフを隣接リストモデルで表す場合は、2 つのアプローチしかない。1 つは、手続き型言語とカーソルを使う伝統的な方法。もう1つは再帰共通表式である。

手続き型のアプローチは、数学の教科書に載っているアルゴリズムを直接翻訳することができる。必要な変換は、配列をテーブルに置き換えるぐらいだ。

再帰共通表式はループを置き換えることができる（内部的にはプロシージャを実行しているので、「隠蔽している」と言ったほうが実情に近いが）。だがそのおかげでオプティマイザによる最適化を受けられ、パラレル実行も可能になる。ただし、再帰は一般に遅く、そのコストは指数関数的に大きくなる。ほとんどの

SQL 製品では、再帰がある程度の深さに達するとまったくクエリが返ってこなくなるだろう。再帰共通表式の一般的なスケルトンを示す。

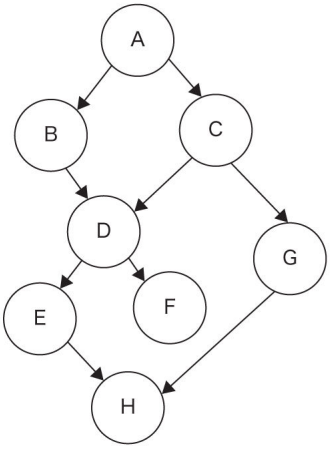
```
WITH RECURSIVE
SolutionGraph (source_node, dest_node, <wgt>, ...)
AS
(SELECT source_node, dest_node,
        <wgt>,
        <other attributes>,
        <possible counts>
   FROM AdjacencyListGraph
 UNION ALL
 SELECT G1.source_node, G2.dest_node,
        <computation on wgt>,
        <computation on other attributes>,
        <increment counts>
   FROM SolutionGraph AS G1, Graph AS G2
  WHERE G2.source_node = G1.dest_node
    AND G2.dest_node <> G1.source_node
    AND NOT EXISTS
         (SELECT *
            FROM Graph AS G3
           WHERE G3.source_node = G1.source_node
             AND G3.dest_node = G2.dest_node
             AND <special conditions>))
SELECT source_node, dest_node,
       <aggregate computation on wgt>,
       <aggregate computation on other attributes>,
       <final counts>
  FROM SolutionGraph
 WHERE <special conditions>
 GROUP BY source_node, dest_node
HAVING <special conditions>;
```

このクエリの意味はこうだ。まず初期のノード集合から出発し、それらが本当に欲しいノードであるかを調べる。もしそうでなければ、再帰的にノードを追加していく。

これ以外にもグラフを構築するアルゴリズムはあるが、これが一般的な設計パターンだ。この方法の難点は、手続き型のプログラムならば最初の解を見つけたところで終わることができるが、再帰共通表式（およびSQL一般）は、どれだけコストがかかろうとも、すべての妥当な解を見つけようとすることだ。

37.1.2　隣接行列モデル

隣接リスト行列は、行で出力ノードを、列で入力ノードを表現する正方形の配列である。セルに1がある場合、2 つのノード間にエッジが存在することを意味する。下のようなグラフを配列で表してみよう【訳注1】。



	A	B	C	D	E	F	G	H
A	1	1	1	0	0	0	0	0
B	0	1	0	1	0	0	0	0
C	0	0	1	1	0	0	1	0
D	0	0	0	1	1	1	0	0
E	0	0	0	0	1	0	0	1
F	0	0	0	0	0	1	0	0
G	0	0	0	0	0	0	1	1
H	0	0	0	0	0	0	0	1

図37.2　単純な有向グラフ

多くのグラフアルゴリズムは隣接行列モデルに基づいており、これはそのままSQLにも翻訳できる（行列についての解説は第33章参照）。たとえば、ノード間の最短経路を求めるダイクストラのアルゴリズムは、この配列を使うと次のようなプロシージャになる。

```
FOR k = 1 TO n
  FOR i = 1 TO n
    FOR j = 1 TO n
      IF a[i,k] + a[k,j] < a[i,j]
      THEN a[i,j] = a[i,k] + a[k,j]
      END IF;
    END FOR;
  END FOR;
END FOR;
```

n個のノードを持つグラフを表すテーブルのサイズはn^2になる。このアルゴリズムの実行コストは、n^3に比例する。このテーブルの利点は、いったんテーブルを作ってしまえば、計算を繰り返す必要がなく、検索だけで済むことだ。たとえば、次のようなデータがあるとしよう。

```
CREATE TABLE AdjacencyListGraph
(source_node CHAR(1),
 dest_node   CHAR(1),
 wgt INTEGER,
     PRIMARY KEY(source_node, dest_node));
```

AdjacencyListGraph：隣接リストグラフテーブル　source_node：出発ノード　dest_node：到達ノード　wgt：重み

```
INSERT INTO AdjacencyListGraph
VALUES ('a', 'd', 1),
       ('d', 'e', 1),
       ('e', 'c', 1),
       ('c', 'b', 1),
       ('b', 'd', 1),
       ('a', 'e', 5);
```

結果は次のようになる。

source_node	dest_node	min_wgt
a	b	4
a	c	3
a	d	1
a	e	2
b	c	3
b	d	1
b	e	2
c	b	1
c	d	2
c	e	3
d	b	3
d	c	2
d	e	1
e	b	2
e	c	1
e	d	3

ダイクストラのアルゴリズムは、配列を使える言語であればおそらくSQLよりかなり速いだろう。

37.2

グラフを入れ子集合モデルで表す

循環しない有向グラフは、入れ子集合モデルで表すことが可能だ。循環しないグラフは、木になるからだ。つまり、これは隣接リストモデルほど一般性のある方法ではなく、特定のタイプのグラフを扱う特殊なトリックである。以下のような入れ子集合のテーブルを使おう。

```
CREATE TABLE NestedSetsGraph
(node_id CHAR(1) NOT NULL REFERENCES Nodes (node_id),
 lft INTEGER NOT NULL CHECK (lft >= 1) PRIMARY KEY,
 rgt INTEGER NOT NULL UNIQUE,
     CHECK (rgt > lft),
     UNIQUE (node_id, lft));
```

NestedSetsGraph：入れ子集合グラフテーブル　node_id：ノードID　lft：左端　rgt：右端

底辺のノードから出発して木をたどり、ノードを分割していく。つまり、もし入次数が1より大きいノードに到達したら、それを上位ノードの下にぶら下がる多くのコピーノードで置き換えていくのだ。これをルートにたどり着くまで繰り返す。そうすると、1つの木（ただし重複ノードを持つ）が得られる。

このモデルの利点は、再帰を使わなくてよいのでパフォーマンスが良いことだ。その代わり、ストレージ容量の消費は多くなる。

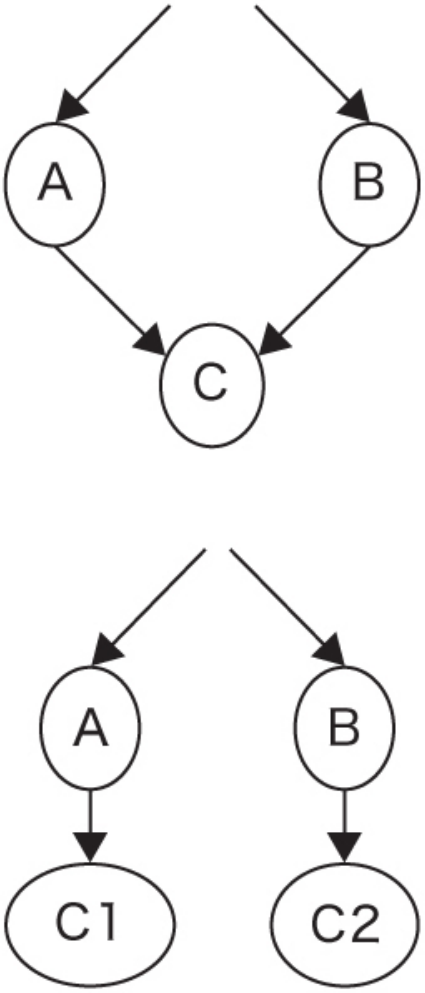


図37.3　ノードを分割する

37.2.1　グラフのすべてのノード

Nodesテーブルのすべてのノードをグラフで使う必要はないし、同じノードが複数回使われることもある。したがって、このモデルではダブルカウントの危険があるため、グラフで使われるノードを一意に保持するビューがあったほうが間違いが少ないだろう。

```
CREATE VIEW GraphNodes (node_id)
AS
SELECT DISTINCT node_id FROM NestedSetsGraph;
```

37.2.2　経路の両端

グラフにおける経路とは、エッジをたどってノードを連続的に探索することだ。1つの経路における終点ノードは、当然のことながら別の経路の始点ノードでもある。経路の長さは、エッジの本数によって決まる。長さ0 のパスでは、経路の両端は同じノードになる——自分に戻ってくるエッジも、グラフにおいてはあってもかまわない。また、1本もエッジを持たない孤立ノードの存在も許されている（隣接リストでは表現できないが）。ただ、そのような孤立ノードに興味を持つ人は少ない。

もし2 つのノード間に1 つ以上の経路が存在する場合、入れ子集合で表すには、それぞれ異なる数字を持つペアとして識別する必要がある。もし2つのノード間に1つしか経路が存在しないが、その経路が他の経路の一部になっている場合、その経路の両端は、より大きな経路の数字のペアとなる。標準的な形としては、最も小さな数字のペアが両端として返される。

```
CREATE VIEW PathEndpoints
(begin_node_id, end_node_id,
 begin_lft, begin_rgt,
 end_lft, end_rgt)
AS
SELECT G1.node_id, G2.node_id,
       G1.lft, G1.rgt, G2.lft, G2.rgt
  FROM (SELECT node_id, MIN(lft), MIN(rgt)
          FROM NestedSetsGraph
         GROUP BY node_id) AS G1 (node_id, lft, rgt)
        INNER JOIN
          NestedSetsGraph AS G2
    ON G2.lft >= G1.lft
   AND G2.lft < G1.rgt;
```

37.2.3　到達可能ノード

もしあるノードが他のノードから到達可能であれば、そのノード間には経路が存在する。当然ながら、どのノードも自分からは到達可能である。

```
CREATE VIEW ReachableNodes (begin_node_id, end_node_id)
AS
SELECT DISTINCT begin_node_id, end_node_id
  FROM PathEndpoints;
```

37.2.4　エッジ

エッジはノードのペアとして表すことができる。エッジEをノードのペア(n0, n1)で表すとすると、n1はn0から到達可能である。

```
CREATE VIEW Edges (begin_node_id, end_node_id)
AS
SELECT begin_node_id, end_node_id
  FROM PathEndpoints AS PE
 WHERE begin_node_id <> end_node_id
   AND NOT EXISTS
        (SELECT *
           FROM NestedSetsGraph AS G
          WHERE G.lft > PE.begin_lft
            AND G.lft < PE.end_lft
            AND G.rgt > PE.end_rgt);
```

37.2.5　入次数と出次数

あるノードに入ってくるエッジの本数を入次数と呼ぶ。全ノードの入次数は次のように求める。ただし、入次数が0の場合は結果を返さない。

```
CREATE VIEW Indegree (node_id, node_indegree)
AS
SELECT N.node_id, COUNT(E.begin_node_id)
  FROM GraphNodes AS N
         LEFT OUTER JOIN
           Edges AS E
    ON N.node_id = E.end_node_id
 GROUP BY N.node_id;
```

あるノードから出ていくエッジの本数を出次数と呼ぶ。全ノードの出次数は次のように求める。ただし、出次数が0の場合は結果を返さない。

```
CREATE VIEW Outdegree (node_id, node_outdegree)
AS
SELECT N.node_id, COUNT(E.end_node_id)
  FROM GraphNodes AS N
         LEFT OUTER JOIN
           Edges AS E
    ON N.node_id = E.begin_node_id
 GROUP BY N.node_id;
```

37.2.6　さまざまなタイプのノードを見つける

ソースノードは、出次数が1以上で、入次数が0のノードである。以下のコードは孤立ノードが存在しないことを前提としている。

```
CREATE VIEW SourceNodes (node_id, lft, rgt)
AS
SELECT node_id, lft, rgt
  FROM NestedSetsGraph AS G1
 WHERE NOT EXISTS
       (SELECT *
          FROM NestedSetsGraph AS G2
         WHERE G1.lft > G2.lft
           AND G1.lft < G2.rgt);
```

同様に、シンクノードは入次数が1以上で、出次数が0のノードである。以下のコードもやはり孤立ノードが存在しないことを前提としている。

```
CREATE VIEW SinkNodes (node_id)
AS
SELECT node_id
  FROM NestedSetsGraph AS G1
 WHERE lft = rgt - 1
   AND NOT EXISTS
       (SELECT *
          FROM NestedSetsGraph AS G2
         WHERE G1.node_id = G2.node_id
           AND G2.lft < G1.lft);
```

孤立ノードは、エッジを持たないノードである。このようなノードをあえてモデルに残しておく意味はない。

```
CREATE VIEW IsolatedNodes (node_id, lft, rgt)
AS
SELECT node_id, lft, rgt
  FROM NestedSetsGraph AS G1
 WHERE lft = rgt - 1
   AND NOT EXISTS
       (SELECT *
          FROM NestedSetsGraph AS G2
         WHERE G1.lft > G2.lft
           AND G1.lft < G2.rgt);
```

内部ノードは入次数と出次数がともに1以上のノードである。つまり、ソースにもシンクにもなりうる。

```
CREATE VIEW InternalNodes (node_id)
AS
SELECT node_id
  FROM (SELECT node_id, MIN(lft) AS lft, MIN(rgt) AS rgt
          FROM NestedSetsGraph
         WHERE lft < rgt - 1
         GROUP BY node_id) AS G1
 WHERE EXISTS
        (SELECT *
           FROM NestedSetsGraph AS G2
          WHERE G1.lft > G2.lft
            AND G1.lft < G2.rgt);
```

37.2.7　非循環グラフを入れ子集合に変換する

まず単純な隣接リストのグラフから始めよう。

```
CREATE TABLE Nodes
(node_id CHAR(1) PRIMARY KEY);

INSERT INTO Nodes (node_id)
VALUES ('a'), ('b'), ('c'), ('d'),
       ('e'), ('f'), ('g'), ('h');

CREATE TABLE AdjacencyListGraph
(begin_node_id CHAR(1),
 end_node_id   CHAR(1),
 path_length   INTEGER,
     PRIMARY KEY(begin_node_id, end_node_id));

INSERT INTO AdjacencyListGraph
VALUES ('a', 'b', 1), ('a', 'c' ,3), ('b', 'd' ,2), ('c', 'd', 1),
       ('c', 'g', 2), ('d', 'e', 5), ('d', 'f', 7), ('e', 'h', 2),
       ('g', 'h', 5);
```

この隣接リストを入れ子集合に変換するには、簡単なスタックアルゴリズムを使えば良い。これは再帰共通表式でも書ける。チャレンジしてみてほしい。

```
-- 深さ優先アルゴリズムでノードをスタックするテーブル
CREATE TABLE NodeStack
(node_id CHAR(1) NOT NULL
     REFERENCES Nodes (node_id),
 distance INTEGER NOT NULL CHECK (distance >= 0),
 lft INTEGER CHECK (lft >= 1),
 rgt INTEGER,
     CHECK (rgt > lft));
```

NodeStack：ノードスタックテーブル　node_id：ノードID　distance：距離　lft：左端　rgt：右端

```
CREATE PROCEDURE AdjacencyToNestedSetsGraph
AS
    path_length INTEGER;
    current_number INTEGER;
    row_num INTEGER;
    node_cnt INTEGER;
BEGIN
    path_length := 0;
    current_number := 0;

    -- テーブルを初期化する
    DELETE FROM NodeStack;

    -- すべてのノードを挿入してスタックを初期化する
    INSERT INTO NodeStack (node_id, distance)
    SELECT DISTINCT G1.begin_node_id, path_length
      FROM AdjacencyListGraph G1
     WHERE NOT EXISTS
            (SELECT *
               FROM AdjacencyListGraph G2
              WHERE G2.end_node_id = G1.begin_node_id);

    SELECT COUNT(*) INTO row_num FROM NodeStack;

    WHILE (row_num > 0)
    LOOP
        current_number := current_number + 1;

        SELECT COUNT(*) INTO node_cnt
          FROM NodeStack
         WHERE distance = path_length;

        IF (node_cnt > 0)
        THEN UPDATE NodeStack
                SET lft = current_number
              WHERE distance = path_length
                AND NOT EXISTS
                     (SELECT *
                        FROM NodeStack S2
                       WHERE distance = path_length
                         AND S2.node_id < NodeStack.node_id);

            INSERT INTO NodeStack (node_id, distance)
            SELECT G.end_node_id, (S.distance + 1)
              FROM NodeStack S,
                   AdjacencyListGraph G
             WHERE S.distance = path_length
               AND S.lft IS NOT NULL
               AND G.begin_node_id = S.node_id;

            path_length := (path_length + 1);
        ELSE path_length := (path_length - 1);
            UPDATE NodeStack
               SET rgt = current_number
```

```
             WHERE lft IS NOT NULL
               AND distance = path_length;

            INSERT INTO NestedSetsGraph (node_id, lft, rgt)
            SELECT node_id, lft, rgt
              FROM NodeStack
             WHERE lft IS NOT NULL
               AND distance = path_length;

            DELETE FROM NodeStack
             WHERE lft IS NOT NULL
               AND distance = path_length;
        END IF;

        SELECT COUNT(*) INTO row_num FROM NodeStack;
    END LOOP;
END;
```

37.3

多角形の中の点

これは正確にはグラフ理論の範疇ではないが、地理的クエリに関連するという点で、本章で扱うのが適切であると思われる。多角形は、(x, y)座標系における頂点の集合として表現できる。この多角形の中にある点が含まれているか否かを調べるクエリを考えたい【訳注2】。以下のアルゴリズムは、ダレル・R・フィンリーによるものだ。この方法の利点は、三角関数を使わなくても標準SQLで実行できることだ。欠点は、凹型の多角形には使えないことだ。この難点の回避策としては、複数の凸型の多角形に分割するという方法がある。

```
CREATE TABLE Polygon
(x FLOAT NOT NULL,
 y FLOAT NOT NULL,
     PRIMARY KEY (x, y));
```

Polygon：多角形テーブル　　x：x座標　　y：y座標

```
INSERT INTO Polygon
VALUES (2.00, 2.00), (1.00, 4.00),
       (3.00, 6.00), (6.00, 4.00), (5.00, 2.00);
```

```
CREATE TABLE Points
(xx FLOAT NOT NULL,
 yy FLOAT NOT NULL,
 location VARCHAR(10) NOT NULL,
     PRIMARY KEY (xx, yy));
```

Points：点テーブル　　xx：x座標　　yy：y座標　　location：位置

```
INSERT INTO Points
VALUES (2.00, 2.00, 'corner'),
       (1.00, 5.00, 'outside'),
       (3.00, 3.00, 'inside'),
       (3.00, 4.00, 'inside'),
       (5.00, 1.00, 'outside'),
       (3.00, 2.00, 'side');
```

```
SELECT P1.xx, P1.yy, p1.location, SIGN(
       SUM (CASE WHEN (polyY.y < P1.yy AND polyY.x >= P1.yy
                   OR polyY.x < P1.yy AND polyY.y >= P1.yy)
                  THEN CASE WHEN polyX.y + (P1.yy - polyY.y)
                        /(polyY.x - polyY.y) * (polyX.x - polyX.x) < P1.xx
                  THEN 1 ELSE 0 END
                  ELSE 0 END))AS flag
  FROM Polygon AS polyY, Polygon AS polyX, Points AS P1
 GROUP BY P1.xx, P1.yy, p1.location;
```

結果

```
xx  yy  location flag
====================
1.0 5.0 outside  0
2.0 2.0 corner   0
3.0 3.0 inside   1
3.0 4.0 inside   1
5.0 1.0 outside  0
3.0 2.0 side     1
```

flag=1 ならば点は内部にある。0 ならば外部だ。多角形の辺上は内部とみなされる。もし頂点を外部とみなしたければ、最初のCASE式を次のように変えればよい。

```
CASE WHEN EXISTS
           (SELECT * FROM Polygon
             WHERE x = P1.xx AND y = P1.yy)
     THEN 1 ...
```

37.4

グラフ理論の参考文献

Berge, Claude. 2001. *The Theory of Graphs*. New York: Dover.（『グラフの理論』C.ベルジュ著，伊理正夫［ほか］共訳，サイエンス社，1976）

Chartrand, Gary. 1984. *Introductory Graph Theory*. New York: Dover.
解説が平易で内容も面白い。

Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. 1990. *Introduction to Algorithms*. McGraw-Hill Companies.（『基礎・ソート・データ構造・数学　第3版』浅野哲夫［ほか］訳，近代科学社，2012）

Even, Shimon. 1979. *Graph Algorithms*. Rockville, MD: Computer Science Press.

Fulkerson, D. R. (ed.). 1975. *Studies in Graph Theory*, Vol. I. American Mathematical Association.

——. 1975. *Studies in Graph Theory, Vol. II*. American Mathematical Association.

Harary, Frank. 1972. *Graph Theory*. Boston: Addison-Wesley.
著者はこの分野の大家である。

Hartsfield, Nora and Ringel, Gerhard. 2003. *Pearls in Graph Theory: A Comprehensive Introduction*. New York: Dover.（『グラフ理論入門』鈴木晋一訳，サイエンス社，1992）

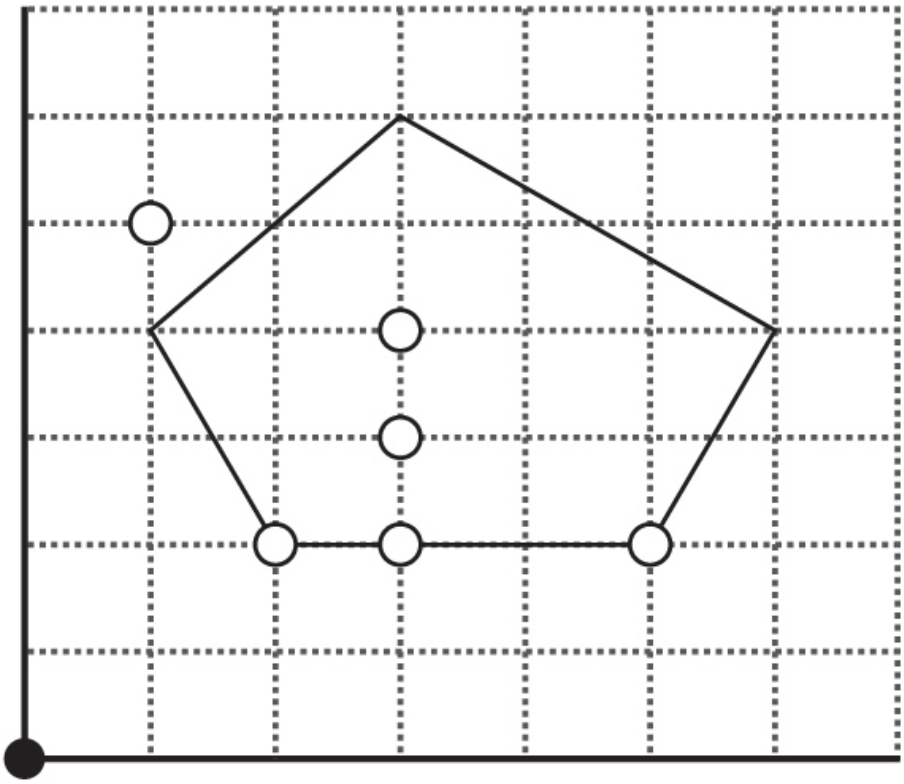
McHugh, James A. 1990. *Algorithmic Graph Theory*. Englewood Cliffs, NJ: Prentice-Hall.

Ore, Oystein (revised by Robin J. Wilson). 1990. *Graphs and Their Uses*. American Mathematical Association.（『やさしくくわしいグラフ理論入門』大石泰彦訳，日本評論社， 1993）
本書は高校生レベルでも理解できるよう書かれた古典である。

Trudeau, Richard J. 1994. *Introduction to Graph Theory, Second Edition*. New York:Dover.

【訳注1】　この配列は自ノードから自ノードへも「1」がセットされており、エッジが存在することを表している。図37.2ではこのような再帰的なエッジは省略されているので注意してほしい。

【訳注2】　この問題のイメージは次のようなものだ。



38

時間を扱うクエリ

Temporal Queries

　時間データは人間が概念的に扱ううえで最も難しいタイプのデータである。時間が難しいのは、他のデータ型が静的であるのに対して、時間型は動的であることに起因する。時間は常に複数の同時生起的な出来事として流れていく。時間は、古来から人々の心を捕らえてきた謎であり、簡単に見えて非常に複雑で奥深い。

38.1

時間の計算

ほぼすべてのSQL 実装がDATE 型を持っている。だが、このデータ型に使うことのできる関数には、実装により違いがある。最も一般的な関数は、整数や文字列から日付を構築する関数と、日付から月、日、年を抽出する関数、および出力を整形するためのいくつかの関数である。出力の整形についてはデータベースの仕事ではないため、本章では触れない。

標準SQLでは、日付の構築子は文字列に適用するCAST(<string expression> AS [DATE |TIME | TIMESTAMP])関数と、次のようなリテラル構築子である。

```
<date literal> ::= DATE <date string>
<time literal> ::= TIME <time string>
<timestamp literal> ::= TIMESTAMP <timestamp string>
```

標準SQLでは、日付の「yyyy-mm-dd」と時刻の「hh:mm:ss.sssss」というISO 8601のフォーマットしか許していない。小数点の桁数は実装依存であるが、FIPS-127は少なくとも秒に関して5 桁を求めている。この形式を採用すると、イギリスで慣習的に使われている「dd/mm/yy」や米国の「mm/dd/yy」を使ったときに起きる混乱（どのパートが年なのか、不慣れな人にはわからない）を避けることができる。ISO 8601には、ほかにもあいまいさを許さない日付と時刻のフォーマットが存在しており、それをSQLで表現することも不可能ではない。たとえば、区切り文字なしの「yyyymmdd」という日付フォーマットを使うこともできる。

時間型から一部の要素を抽出する関数はEXTRACT(<extract field> FROM <extract source>)である。これは次のように定義される:

```
<extract field> ::= <primary datetime field> | <time zone field>
<time zone field> ::= TIMEZONE_HOUR | TIMEZONE_MINUTE
<extract source> ::= <datetime value expression> | <interval value expression>
```

<extract field>に指定できるものを表38.1に挙げておく。

表38.1　抽出要素の一覧

<primary datetime field>の意味	予約語
西暦における年（0001 ～ 9999）	YEAR
年の中の月（01 ～ 12）	MONTH
月の中の日（01 ～ 31）	DAY
時間の中の分（00 ～ 59）	MINUTE
分の中の秒（00 ～ 59.99 …）	SECOND
タイムゾーンにおける時間。正負両方の値を取りうる（-12 ～＋14）	TIMEZONE_HOUR
タイムゾーンにおける分。正負両方の値を取りうる（-59 ～＋59）	TIMEZONE_MINUTE

当たり前のことだが、単位として大きい順に並べると、YEAR、MONTH、DAY、HOUR、MINUTE、SECONDとなる。SECONDだけが小数の値を取りうる。それ以外は日付・時刻として妥当な正の整数しか取ることができない。

標準SQL（ISO/IEC 9075）では、次に示す主要な時間型を3つ定義している。

- **DATE**：YEAR、MONTH、DAYを含む
- **TIME**：HOUR、MINUTE、SECONDを含む
- **TIMESTAMP**：YEAR、MONTH、DAY、HOUR、MINUTE、SECONDを含む

時間型のデータ同士が比較可能なのは、同じ主要なフィールドを持っている場合に限られる。SQLの実装における時間型の基本的な演算は、日付に期間を足し引きすることで新しい日付を出力するものと、日付から別の日付を引いて日数を出力するものである。次に、標準SQLにおける、日付と期間の妥当な演算の組み合わせを示す。

```
<datetime> - <datetime> = <interval>
<datetime> + <interval> = <datetime>
<interval> (* または /) <numeric> = <interval>
<interval> + <datetime> = <datetime>
<interval> + <interval> = <interval>
<numeric> * <interval> = <interval>
```

標準SQLには、ほかにもタイムゾーンなどに関するルールもあるが、直観的にも明らかである。また、読者の使う実装には、システムクロックから現在日付を返す関数があるはずだ。この関数はベンダーによってばらばらで、TODAY、SYSDATE、NOW、CURRENT DATE、getdate()などという名前が付いている。実装によっては、日付から曜日を返す関数もあるかもしれない。これはDOWとかWEEKDAYと呼ばれている。

標準SQL が提供しているのは、CURRENT_DATE、CURRENT_TIME [(<time precision>)]、CURRENT_TIMESTAMP [(<timestamp precision>)]関数のみである。これらは特に説明の必要はないだろう。

38.2

個人的なカレンダー

日付関連のアプリケーションで最もよく使われているのが、ユーザーの将来の予定や行動を記録する個人的なカレンダーを作るものだ。紙のカレンダーに予定を書き込み、それを見て何をすべきかを考えるきっかけにすることは、多くの人がやっているだろう。それの電子版である。

循環的なデータを保持するテーブルを作るのは、思ったよりも簡単である。日付の循環は400 年周期である。つまり、400年分の曜日と日付のセットを用意することで半永久的に使用できる。するとテーブルには20,871 週または146,097日を保持すれば良い。これは今日では大きなデータ量ではない。

このとき便利なトリックが、インデックスを降順ソートしておくことだ。というのも、大半の作業は直近の日付に対して行うだろうから、目的のデータをスキャンの最初のほうで見つけられるよう、物理的なテーブルの配置順序を考慮するのが得策だ。たとえば、営業日は西洋諸国では土日祝日以外を意味するが、イスラム圏では金曜日が週休日のことが多い。

祝日を分類すると、以下のような種類になるだろう。

祝日1　毎年同じ日付

祝日2　復活祭のような移動祝日【訳注1】

祝日3　基本的に同じ日付だが週末と重なるとスライドして3連休になる

祝日4　基本的に同じ日付だが週末または火曜日だと月曜日になる（このタイプはイギリスのボクシング・デーのみ）【訳注2】

祝日5　ある日付（通常は月の最初か最後の月曜日）のあとの数日間（感謝祭など）

祝日6　ギリシャ正教会の復活祭（西方教会とは異なる日付）

祝日7　イスラム暦における毎年同じ日付。ただし、宗教指導者が月の満ち欠けを判断して祝日を判断するケースもあり、厳密な日付が決まらないこともある

祝日8　冬至を起点にした祝日（中国の清明節など）

祝日9　法令により定められる公休日（日本の大喪の礼など）

ざっと見ただけでも、かなりルールが複雑だし、あいまいだと思うだろう。

38.3

期間のシーケンス

現実世界における主要な問題の1つは、特定の順序で同時的に生起する一連の出来事をどのように扱うか、というものだ。最良の方法は、予期せぬ歯抜けや重複が発生しないよう、あらかじめDDLでそれらを排除しておくことだ。アレクサンダー・クズネツォフは、期間の重複を避けるための制約を付与したDDLを考えた。

```
CREATE TABLE Events
(event_id INTEGER NOT NULL,
 some_value INTEGER NOT NULL,
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 prev_end_date DATE,
     PRIMARY KEY(event_id, end_date),
     UNIQUE(event_id, prev_end_date),
     CONSTRAINT no_overlaps
       CHECK (prev_end_date <= start_date),
     CONSTRAINT starts_before_ending
       CHECK (start_date < end_date));
```

Events：イベントテーブル　event_id：イベントID　some_value：何らかの値　start_date：開始日　end_date：終了日　prev_end_date：前回の終了日

もし欠番を完全に防止したければ、`CHECK (prev_end_date <= start_date)`という制約を、より厳しく`CHECK (prev_end_date = start_date)`という制約に変えればよい。

もし非連続な期間を許した状態であれば、その欠けた期間を選択するクエリは次のようになる。

```
SELECT prev_end_date AS gap_start_date,
       start_date AS gap_end_date
  FROM Events
 WHERE start_date > prev_end_date;
```

`end_date`列を変更する場合は、次の期間と整合性を取るために`prev_end_date`列も更新しなければならないだろう。

```
UPDATE Events
   SET end_date
         = CASE
           WHEN end_date = :in_old_end_date
           THEN :in_new_end_date
           ELSE end_date END,
       prev_end_date
         = CASE
           WHEN prev_end_date = :in_old_end_date
           THEN :in_new_end_date
           ELSE prev_end_date END
 WHERE event_id = in_event_id
   AND in_old_end_date IN (end_date, prev_end_date);
```

読者の中には、DDLへON UPDATE CASCADE句を付けるという方法を考えた人もいるかもしれない。だが、SQL製品の中には、CASCADEで自己参照を許していないものもある。

一般には、prev_end_date列を持たないテーブル構造を取ることも多いだろう。今度はそのような(start_time, end_time)のペアだけで期間を管理する方法を考えてみよう。

38.3.1 期間のシーケンスにおける欠落

時間軸は複数の期間に分割できる。そして期間の集合を対象にしたクエリは頻繁に利用される。典型的なのが、採用面接の応募書類で、職歴の欄に空白期間があった場合に面接官が訊く「えーと……、この2年間は何をしていらしたのでしょう？」という質問だ。この質問はえぐい。たいていは単純に職を見つけられなくて無職だった期間だからだ。もし読者が情報システム業界で働いていたなら、「フリーのコンサルタントをしていました」とでも答えるしかないだろう（これは無職のシノニムである）。

さて、次のようなテーブルを考える。候補者ごとの職歴の期間を表示するSQLはどのように書けばいいだろう？

```
CREATE TABLE JobApps
(candidate_name CHAR(25) NOT NULL,
 job_title CHAR(15),
 start_date DATE NOT NULL,
 end_date DATE, -- NULLは雇用中
     CONSTRAINT started_before_ended
       CHECK(start_date <= end_date));
```

JobApps：採用面接テーブル　candidate_name：候補者名　job_title：職歴　start_date：開始日　end_date：終了日

JobApps

candidate_name	job_title	start_date	end_date
John Smith	Vice Pres	2010-01-10	2010-12-31
John Smith	President	2011-01-12	2012-12-31
Bill Jones	Scut Worker	2011-02-24	2011-04-21
Bill Jones	Manager	2012-01-01	2012-01-05
Bill Jones	Grand Poobah	2012-04-04	2012-05-15

現職の`end_date`列は`NULL`で表現する。これはSQLが「永遠」とか「時間の終わり」に相当する時間データを持っていないからだ。可能な最大の値、たとえば「9999-12-31 23:59:59.999999」を使うというのは、正しいモデリングではないため、時間の計算を行う際に問題となる。`NULL`は`COALESCE`関数を使えばコード内で任意の値に変換できるので、柔軟性が高い。この点はあとで見よう。

自己結合を使う必要があることは自明である。無職期間の最初の日は、雇用期間の終了日プラス1である。また、無職期間の終了日は、次の雇用期間の開始日マイナス1である。この開始点と終了点の問題を扱うために、SQLは`OVERLAPS`述語を定義している。

SQLのすべての実装は日付型の演算を定義しているが、残念なことに、そのルールは実装間で統一されていないし、標準SQLにも従っていない。まず最初に考えるクエリは次のようなものだろう。

```
SELECT J1.candidate_name,
       (J1.end_date + INTERVAL '1' DAY) AS gap_start,
       (J2.start_date - INTERVAL '1' DAY) AS gap_end,
       (J2.start_date - J1.end_date) AS gap_length
  FROM JobApps AS J1, JobApps AS J2
 WHERE J1.candidate_name = J2.candidate_name
   AND (J1.end_date + INTERVAL '1' DAY) < J2.start_date;
```

結果

```
candidate_name  gap_start   gap_end     gap_length
================================================
John Smith      2011-01-01  2011-01-11  12
Bill Jones      2011-04-22  2011-12-31  255
Bill Jones      2012-01-06  2012-04-03  90
Bill Jones      2011-04-22  2012-04-03  349  ⬅間違った結果
```

この結果は間違っている。Smith 氏については問題ない。彼の職歴は2 つで、その間に無職期間が1つある。問題はJones氏の職歴だ。彼は3つの職歴を持っており、その間に2つの無職期間があるだけなのに、結果には3つの無職期間が現れている。理由は、このクエリが開始日と終了日の可能なすべての組み合わせについて自己結合を行っていることだ。これは結果としてあり得ない組み合わせを生んでしまう。ここでは、「Scut Worker」の終了日と「Grand Poobah」の開始日のペアである。だが、歯抜けを探すのは、あくまで隣り合う期間同士が対象だ。これはMIN関数と相関サブクエリで見つけられる。正しいクエリはこうだ。

```
SELECT J1.candidate_name,
       (J1.end_date + INTERVAL '1' DAY) AS gap_start,
       (J2.start_date - INTERVAL '1' DAY) AS gap_end
  FROM JobApps AS J1, JobApps AS J2
 WHERE J1.candidate_name = J2.candidate_name
   AND J2.start_date
         = (SELECT MIN(J3.start_date)
              FROM JobApps AS J3
             WHERE J3.candidate_name = J1.candidate_name
               AND J3.start_date > J1.end_date)
               AND (J1.end_date + INTERVAL '1' DAY)
                    < (J2.start_date - INTERVAL '1' DAY);
```

結果

```
candidate_name   gap_start            gap_end
=========================================================
Bill Jones       2011-04-22 00:00:00  2011-12-31 00:00:00
Bill Jones       2012-01-06 00:00:00  2012-04-03 00:00:00
John Smith       2011-01-01 00:00:00  2011-01-11 00:00:00
```

これで無職期間の日数も簡単に求められる。

38.3.2　連続的な期間

任意の期間だけ続くタスクの履歴があるとしよう。このとき、タスクとタスクの間に空白期間が存在していないかどうかを調べるにはどうすればよいだろう？

次のようなタスクの履歴テーブルを考えよう【訳注3】。

```
CREATE TABLE Timesheets
(task_id CHAR(5) NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
     CONSTRAINT started_before_ended
     CHECK (start_date <= end_date));
```

Timesheets：タイムシートテーブル　task_id：業務ID　start_date：開始日　end_date：終了日

```
INSERT INTO Timesheets
VALUES (1, '1997-01-01', '1997-01-03'),
       (2, '1997-01-02', '1997-01-04'),
       (3, '1997-01-04', '1997-01-05'),
       (4, '1997-01-06', '1997-01-09'),
       (5, '1997-01-09', '1997-01-09'),
       (6, '1997-01-09', '1997-01-09'),
       (7, '1997-01-12', '1997-01-15'),
       (8, '1997-01-13', '1997-01-14'),
       (9, '1997-01-14', '1997-01-14'),
       (10, '1997-01-17', '1997-01-17');
```

最も直接的な解としては、連続的な期間の特性をすべて網羅した検索条件を作ることだ。これは最も古い開始日と最も直近の終了日を求めている。

```
SELECT T1.start_date, MAX(T2.end_date) AS end_date
  FROM Timesheets AS T1, Timesheets AS T2
 WHERE T1.start_date <= T2.end_date
   AND NOT EXISTS
       (SELECT *
          FROM Timesheets AS T3, Timesheets AS T4
         WHERE T3.end_date < T4.start_date
           AND T3.start_date >= T1.start_date
           AND T3.end_date <= T2.end_date
           AND T4.start_date >= T1.start_date
           AND T4.end_date <= T2.end_date
           AND NOT EXISTS
               (SELECT *
                  FROM Timesheets AS T5
                 WHERE T5.start_date
                       BETWEEN T3.start_date AND T3.end_date
                   AND T5.end_date
                       BETWEEN T4.start_date AND T4.end_date))
 GROUP BY T1.start_date
HAVING t1.start_date = MIN(t2.start_date);
```

結果

```
start_date  end_date
======================
1997-01-01  1997-01-05
1997-01-06  1997-01-09
1997-01-12  1997-01-15
1997-01-17  1997-01-17
```

　別の方法を考えてみよう。どの仕事が連続的な期間を構成している（重なっている）かを調べるクエリだ。

```
SELECT T2.start_date,
       MAX(T1.end_date) AS end_date,
       MAX(T2.task_id || ' to ' || T1.task_id) AS task_id_pair
  FROM Timesheets AS T1, Timesheets AS T2
 WHERE T2.task_id <> T1.task_id
   AND T1.start_date BETWEEN T2.start_date AND T2.end_date
   AND T2.end_date BETWEEN T1.start_date AND T1.end_date
 GROUP BY T2.start_date;
```

結果

```
start_date  end_date    task_id_pair
==================================
1997-01-01  1997-01-04  1 to 2
1997-01-02  1997-01-05  2 to 3
1997-01-06  1997-01-09  4 to 6
1997-01-09  1997-01-09  6 to 5
1997-01-13  1997-01-14  8 to 9
```

　3つ目の解は、期間の重複するタスクを1つとみなして、その日数を数えるものだ。

```
SELECT T1.start_date,
       MIN(T2.end_date) AS end_date,
       MIN(T2.end_date + INTERVAL '1' DAY)
          - MIN(T1.start_date) AS duration -- find any (T1.start_date)
  FROM Timesheets AS T1, Timesheets AS T2
 WHERE T2.start_date >= T1.start_date
   AND T2.end_date >= T1.end_date
   AND NOT EXISTS
         (SELECT *
            FROM Timesheets AS T3
           WHERE (T3.start_date <= T2.end_date
             AND T3.end_date > T2.end_date)
                 OR (T3.end_date >= T1.start_date
             AND T3.start_date < T1.start_date))
 GROUP BY T1.start_date;
```

結果

```
start_date  end_date    duration
================================
1997-01-01  1997-01-05  5 days
1997-01-06  1997-01-09  4 days
1997-01-12  1997-01-15  4 days
1997-01-17  1997-01-17  1 day
```

次の方法は、補助カレンダーテーブル（詳細は38.8 節参照）を使って、他の期間によってカバーされる期間とカバーされない期間を見つけるものだ。`covered_date_flag`と`cal_date`は、データの範囲ではなく単一の日について調べる他のクエリで使うこともできる。

```
SELECT C1.cal_date,
       SUM(DISTINCT CASE
                    WHEN C1.cal_date BETWEEN T1.start_date
                                         AND T1.end_date
                    THEN 1 ELSE 0 END) AS covered_date_flag
  FROM Calendar AS C1, Timesheets AS T1
 WHERE C1.cal_date BETWEEN (SELECT MIN(start_date) FROM Timesheets)
                       AND (SELECT MAX(end_date) FROM Timesheets)
 GROUP BY C1.cal_date;
```

これはかなり速いだろう。というのは、`WHERE`句は静的なスカラサブクエリであり、カレンダーテーブルの主キーである`cal_date`を指定するので、インデックスを利用できるからだ。

この問題を少し改変したバージョンとして、連続的な単位を基準にして、期間をグループ化するというものがある。次のようなテーブルを考えよう。

```
CREATE TABLE Calibrations
(start_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL PRIMARY KEY,
 end_time TIMESTAMP NOT NULL,
     CONSTRAINT started_before_ended
       CHECK (end_time = start_time + INTERVAL '1' MINUTE),
 cal_value INTEGER NOT NULL);
```

Calibrations：目盛りテーブル　start_time：開始時刻　end_time：終了時刻　cal_value：目盛りの値

Calibrations

start_time	end_time	cal_value

2014-05-11 02:52:00.000	2014-05-11 02:53:00.000	8
2014-05-11 02:53:00.000	2014-05-11 02:54:00.000	8
2014-05-11 02:54:00.000	2014-05-11 02:55:00.000	8
2014-05-11 02:55:00.000	2014-05-11 02:56:00.000	8
2014-05-11 02:56:00.000	2014-05-11 02:57:00.000	8
2014-05-11 02:57:00.000	2014-05-11 02:58:00.000	9
2014-05-11 02:58:00.000	2014-05-11 02:59:00.000	9
2014-05-11 02:59:00.000	2014-05-11 03:00:00.000	9
2014-05-11 03:00:00.000	2014-05-11 03:01:00.000	9
2014-05-11 03:01:00.000	2014-05-11 03:02:00.000	9
2014-05-11 03:02:00.000	2014-05-11 03:03:00.000	8
2014-05-11 03:03:00.000	2014-05-11 03:04:00.000	8
2014-05-11 03:04:00.000	2014-05-11 03:05:00.000	8

これを次のようにグループ化したい。

```
start_time               end_time                 cal_value
============================================================
2014-05-11 02:52:00.000  2014-05-11 02:57:00.000  8
2014-05-11 02:57:00.000  2014-05-11 03:02:00.000  9
2014-05-11 03:02:00.000  2014-05-11 03:05:00.000  8
```

選択される結果は、目盛りの値が新しく追加されるたびに更新されていく。目盛りの値は1分ごとに更新される。目盛りの値の開始と、変更前の終了の値を合計するクエリは次のようになる。

```
SELECT MIN(start_time) AS start_time,
       MAX(end_time) AS end_time,
       cal_value
  FROM (SELECT C1.start_time, C1.end_time, C1.cal_value,
               MIN(C2.start_time)
          FROM Calibrations AS C1
                 LEFT OUTER JOIN
                   Calibrations AS C2
            ON C1.start_time < C2.start_time
           AND C1.cal_value <> C2.cal_value
         GROUP BY C1.start_time, C1.end_time, C1.cal_value)
            AS T (start_time, end_time, cal_value, x_time)
 GROUP BY cal_value, x_time;
```

38.3.3　連続的な期間における失われた時間

次のような単純なテーブルを考える。これは、失われた時間をどう扱うかという問題だ。

```
CREATE TABLE Events
(event_id CHAR(2) NOT NULL PRIMARY KEY,
 start_date DATE,
 end_date DATE
     CONSTRAINT started_before_ended
       CHECK (start_date < end_date));
```

Events：イベントテーブル　　start_date：開始日　　end_date：終了日

```
INSERT INTO Events
VALUES ('A', '2014-01-01', '2014-12-31'),
       ('B', '2015-01-01', '2015-01-31'),
       ('C', '2015-02-01', '2015-02-28'),
       ('D', '2015-02-01', '2015-02-28');
```

いま、やむを得ない事情により`end_date`列は`NULL`を格納することがあると想定する。たとえば、`('B', '2015-01-01', NULL)`のようなデータもありうる。

```
INSERT INTO Events
VALUES ('A', '2014-01-01', '2014-12-31'),
       ('B', '2015-01-01', NULL),
       ('C', '2015-02-01', '2015-02-28'),
       ('D', NULL, '2015-02-28');
```

1つの合理的な解は、失われた`end_date`列の値を、次の期間の開始日マイナス1日と考えることだ。これは非常に簡単だ。

```
UPDATE Events
   SET end_date
         = (SELECT MIN(E1.start_date) - INTERVAL '1' DAY
              FROM Events AS E1
             WHERE E1.start_date > Events.start_date)
 WHERE end_date IS NULL;
```

さて、終了日と同じやむを得ない事情により、start_date 列もNULL を許すと仮定しよう。すると、('B', NULL, '2015-01-31')のような行を許すことになる。

この場合も同じロジックを使って、終了日プラス1日を開始日とみなすようにすれば話は簡単だ。

```
UPDATE Events
   SET start_date
         = (SELECT MIN(E1.end_date) + INTERVAL '1' DAY
              FROM Events AS E1
             WHERE E1.end_date < Events.end_date)
 WHERE start_date IS NULL;
```

このクエリは対称的であり、1 つのUPDATE 文にまとめることができる。そうすれば、余計なテーブルスキャンをせずに済む。

```
UPDATE Events
   SET end_date
         = CASE WHEN end_date IS NULL
                THEN (SELECT MIN(E1.start_date) - INTERVAL '1' DAY
                        FROM Events AS E1
                       WHERE E1.start_date > Events.start_date)
                ELSE end_date END,
       start_date
         = CASE WHEN start_date IS NULL
                THEN (SELECT MIN(E1.end_date) + INTERVAL '1' DAY
                        FROM Events AS E1
                       WHERE E1.end_date < Events.end_date)
                ELSE start_date END
 WHERE start_date IS NULL
    OR end_date IS NULL;
```

結果

```
event_id  start_date  end_date
================================
A         2014-01-01  2014-12-31
B         2015-01-01  2015-01-31
C         2015-02-01  2015-02-28
D         2015-01-01  2015-02-28
```

本当の問題は、開始日も終了日もわからず、どこに位置付けてよいかわからないイベントの場合だ。たとえば、次のようなデータである。

```
INSERT INTO Events
 VALUES ('A', '2014-01-01', '2014-12-31'),
        ('B', NULL, NULL),
        ('C', '2015-02-01', '2015-02-28'),
        ('D', '2015-02-01', '2015-02-28');
```

このデータに先ほどのUPDATEを実行しても、NULLを埋めることはできない。いずれにせよ、NULLを他の値に変換するルールは、その場の状況に応じて考えるしかない。とはいえ、このような期間のペアをもとにしたアプローチは常にうまくいくとは限らない。このやり方を使うときのルールを考えてみよう。

ルール1　もしstart_dateが非NULL かつend_dateも非NULL であれば、何も変更の必要はない。

ルール2　もしテーブルがあまりに多くのNULL を持っている場合は、修正を諦めて上司に手は尽くしたと報告する。

ルール3　start_dateがNULLでend_dateが非NULLの場合、start_dateにend_dateより前の日付を設定する。

ルール4　start_date が非NULL でend_date がNULL の場合、end_date に次の既知のstart_dateよりも前の日付を設定する。

ルール5　start_dateとend_dateにともにNULLが存在する場合、最小のstart_dateまたはend_dateを探すために先行と後続のイベントを検索する。

「ルール5」のやり方は、結局あとで解決すべき歯抜けを残すことになる。たとえば、次の例を使って説明しよう。

データの初期状態が次のようになっているとする。

```
('A', '2014-01-01', '2014-12-31'),
('B', '2015-01-01', NULL),
('C', NULL, '2015-02-28'),
('D', '2015-02-01', '2015-02-28');
```

これが次のようになる。

```
('A', '2014-01-01', '2014-12-31'),
('B', '2015-01-01', NULL),
('C', '2015-02-28', '2015-02-28'),  ⬅ルール3
('D', '2015-02-01', '2015-02-28');
```

さらに次のようになる。

```
('A', '2014-01-01', '2014-12-31'),
('B', '2015-01-01', '2015-02-27'),  ⬅ルール4
('C', '2015-02-28', '2015-02-28'),
('D', '2015-02-01', '2015-02-28');
```

次はこういうデータで考えてみよう。

```
('A', '2014-01-01', '2014-12-31'),
('B', NULL, NULL),
('C', '2015-02-01', '2015-02-28'),
('D', '2015-02-01', '2015-02-28');
```

これは次のようになる。

```
('A', '2014-01-01', '2014-12-31'),
('B', '2015-01-01', '2015-01-31'),  ⬅ルール5
('C', '2015-02-01', '2015-02-28'),
('D', '2015-02-01', '2015-02-28');
```

では最後に、次のデータに対してはどうなるだろうか。考えてもらいたい。

```
('A', '2014-01-01', '2014-12-31'),
('B', NULL, NULL),
('C', NULL, '2015-02-29'),
('D', '2015-02-01', '2015-02-29');
```

38.3.4　日付の引き算

この問題は、最初に聞いたときは簡単に思えるかもしれない。だが、なかなか難しいところがある。私が最初にこの問題を知ったのは、『Explain』誌上だった。後に、ボストンで開かれたデータベースワールドカンファレンスで、この問題を考えたルディ・ライムバックに会う機会があった。問題は、今日から45日以内に誕生日を迎える社員のリストを印刷するというものだ。社員のテーブルには誕生日の列が存在する。この問題の解は、実装がどのような日付関数を持っているかによって変わる（ルディ自身はDB2を使っていた）。

```
CREATE TABLE Personnel
(emp_name CHAR(16) PRIMARY KEY,
 dob DATE);
```

Personnel：社員テーブル　emp_name：社員名　dob：誕生日

```
INSERT INTO Personnel
VALUES ('Adam',  '1979-02-17'),
       ('Beth',  '1981-03-17'),
       ('Celine','1970-04-03'),
       ('Dave',  '1969-04-04'),
       ('Eva',   '1989-10-01');
```

今日の日付は、「2013 年2 月17 日」だとしよう。今日から45 日以内に誕生日を迎えるのは、Adam、Beth、Celineの3人だ。

この問題が興味深いのは、考え方の出発点を間違えやすいことだ。まず、最初に多くの人がとっかかりとして考えるクエリは、検索条件の中で日付の一部を取り出して使おうとするものだろう。たとえば、同月内のすべての誕生日を探すなら、次のようなクエリになる。

```
SELECT emp_name, dob, CURRENT_DATE
  FROM Personnel
 WHERE EXTRACT(MONTH FROM CURRENT_DATE)
         = EXTRACT(MONTH FROM dob);
```

結果

```
 emp_name  dob         current_date
===================================
 Adam      1979-02-17  2013-02-17
```

だが、このアプローチを拡張しようとする試みは、かなりたいへんだ。なぜなら、45日間という期間は、3か月にまたがる可能性があるし、それどころか次の年までまたがる可能性もあるし、閏日まで扱わなければならない。すぐに関数の呼び出し回数が増えてロジックが複雑になる。

そこで次に考えるアプローチは、誕生日が今日から45日以内に存在するかを調べる単純な検索条件を作ることだ。これは、社員の誕生日のうち、月と日を今年の年と連結することで可能になる。

```
SELECT emp_name, dob, CURRENT_DATE
  FROM Personnel
 WHERE CAST(EXTRACT (YEAR FROM CURRENT_DATE) || '-' ||
            EXTRACT (MONTH FROM dob) || '-' ||
            EXTRACT (DAY FROM dob) AS DATE)
         - CURRENT_DATE <= 45;
```

だが、これだと閏日の誕生日を持つ社員の場合に例外を返すだろう。1976年2月29日生まれの人は、2013年2月29日という日付に変換されるが、このような日付は存在しない。

さらに、このやり方で問題になるのは、年をまたがる場合だ。たとえば、今日が2013 年12月だとしたら、1月の誕生日の人も含めなければならないが、こうした人々を含めることができない。

標準SQL では、`(date2-date1) YEAR` は、`INTERVAL` 値を返す。これを使うと答えが極め てシンプルになる【訳注4】。

```
SELECT emp_name, dob, CURRENT_DATE
  FROM Personnel
 WHERE INTERVAL (CURRENT_DATE - birthday) YEAR
          < INTERVAL (CURRENT_DATE - birthday + INTERVAL 45 DAYS) YEAR;
```

これを日本語で表現するなら、もし社員の年齢が45日以内ならば、その社員は45日以内に生まれた、ということである。

38.3.5　開始日と終了日

日付をデータベースに格納する方法はいろいろある。そのため、与えられた日付に対して、それが月の最初の日か月の最後の日かを判断するこれがベストとい

う方法はない。また、そのような標準関数も存在しない。

月の最初の日を次の式で得られることは明らかだ。

```
SELECT CAST(EXTRACT (YEAR FROM CURRENT_DATE) || '-' ||
            EXTRACT (MONTH FROM CURRENT_DATE) || '-01' AS DATE);
```

前月の月の最終日を得るには、上で求めた日付から1日引けばよい。

```
SELECT CAST(EXTRACT (YEAR FROM CURRENT_DATE) || '-' ||
            EXTRACT (MONTH FROM CURRENT_DATE) || '-01' AS DATE)
        - INTERVAL '1' DAY;
```

38.3.6　開始時間と終了時間

時間に関する問題の1 つに、時間データに関する標準であるISO 8601 など絶対に読まない人々が、「24:00:00」という時刻を使って、これは翌日に属する時刻だと言い張ることだ。だがそうだとすると、「2014-01-02 00:01:00」も「2014-01-01 24:01:00」に変換しなければなるまい。DB2は`TIMESTAMP` 型の時間要素に00と24 の両方を許可しており、これが問題をさらにややこしくしている。このおかげで、`EXTRACT`関数はいまでは必要以上に複雑な仕様になっている。

人々にISO 8601を読むよう強制できないとすれば、最良の方法は、入力時に時刻の文字列を正しい形式に変換しておくことだ【訳注5】。

38.4

ユリウス通日

DATE型はすべてのSQL製品が持っているが、内部的にどのように実装するべきかについては特に取り決めがない。年、月、日を2桁の整数で保持するという方法もあるが、文字列として保持してもかまわないし、ユリウス通日を使ってもよい。プログラマとしては、DATE型として正しく動作してくれさえすればそれでよい。

なお、ここで言うユリウス通日は、ユリウス暦とは異なるので注意してほしい。ユリウス通日は、天文学でよく使われる用語で、紀元前4713 年1月1日を起点とした日数で日付を管理する方法だ（一方のユリウス暦は文字どおりユリウス・カエサルが採用した太陽暦の暦法である）。この暦法に従うと、現在は2億日を超えたあたりだ。これは年をまたいだ2点の間隔を計算するのに便利なので、天文学者がよく使うが、一般にはあまり使われない。だがコンピュータ企業はこの用語をもう少し広義にとって、特定の日付や時刻を開始点として通時的な時間体系を採用している【訳注6】。ISO標準では年日付というものもあり、これは一年の中でその日付がどこに位置するかを示す。したがって1から365または366までの値を取る。これはよく卓上カレンダーのページの端に印刷されている。

現在の日付がユリウス通日で何日目かを知る通常の方法は、各月の日数を配列で保存して、現在の月の経過日数を足す単純なプログラムを書くことである。注意点は、閏年の場合は2 月に1日足すのを忘れないようにする点ぐらいだ。グレゴリオ暦からユリウス通日へ変換する簡単な関数を以下に示す。このアルゴリズムを最初に書いたのはロバート・タンツェンである（ACM, 1980）。

```
CREATE FUNCTION Julianize
(greg_day IN INTEGER, greg_month IN INTEGER, greg_year IN INTEGER)
RETURN INTEGER
IS
    century INTEGER;
    yearincentury INTEGER;

    tmp_month INTEGER;
    tmp_year INTEGER;
BEGIN
    tmp_month := greg_month;
    tmp_year  := greg_year;

    IF (tmp_month > 2)
    THEN tmp_month := tmp_month - 3;
    ELSE tmp_month := tmp_month + 9;
         tmp_year := tmp_year - 1;
    END IF;

    century := tmp_year/100;
    yearincentury := tmp_year - 100 * century;

    RETURN ((146097 * century)/4
    + (1461 * yearincentury)/4
    + (153 * tmp_month + 2)/5 + greg_day + 1721119);
END;
```

　このアルゴリズムで行われる除算はすべて整数の除算である。これは引数が整数であるから当然だ。

　ユリウス通日をグレゴリオ暦に変換するアルゴリズムは、ピーター・マイヤーによる。

```
CREATE FUNCTION JulDate (julian IN INTEGER)
RETURN INTEGER
IS
    z INTEGER;
    r INTEGER;
    g INTEGER;
    a INTEGER;
    b INTEGER;
    c INTEGER;
    greg_year INTEGER;
    greg_month INTEGER;
    greg_day INTEGER;
BEGIN
    z := FLOOR(julian - 1721118.5);
    r := julian - 1721118.5 - z;
    g := z - 0.25;
    a := FLOOR(g/36524.25);
    b := a - FLOOR(a/4.0);
    greg_year := FLOOR((b + g)/365.25);
    c := b + z - FLOOR(365.25 * greg_year);
    greg_month := TRUNC((5 * c + 456)/153);
    greg_day := c - TRUNC((153 * greg_month - 457)/5) + r;

    IF greg_month > 12
    THEN greg_year := greg_year + 1;
         greg_month := greg_month - 12;
    END IF;

    RETURN (greg_year * 1000) + (greg_month * 100) + greg_day;
END;
```

これらのアルゴリズムには、2つポイントがある。1つは、ユリウス通日は正午から始まることだ。これは天文学者というのは昼間寝て夜働くからだ（我々の感覚で1日が夜中に始まるのと逆の感覚である）。2つ目は、使用する整数が大きくなりがちなため、丸め誤差が大きくなり、実装依存の浮動小数点の精度では表せないかもしれないことだ。

38.5

その他の時間関数

標準SQLには存在しない時間関数としては、曜日を扱うものがある。たとえば、PostgreSQL はDOWという入力の日付に対する曜日を返す関数を持っている。2〜8の値を返し、8 が日曜日である。これを持っていない実装でも、ツェラーの公式を使えば、日付から曜日へ変換できる。

```
CREATE FUNCTION Zeller (z_year IN INTEGER, z_month IN INTEGER, z_day IN INTEGER)
RETURN INTEGER
IS
    m INTEGER;
    d INTEGER;
    y INTEGER;
BEGIN
    y := z_year;
    m := z_month - 2;
    IF (m <= 0)
    THEN m := m + 12;
    y := y - 1;
    END IF;

    RETURN (MOD((z_day + (13 * m - 1)/5
             + 5 * MOD(y, 100)/4 - 7 * y/400), 7) + 1);
END;
```

38.6

週

週を単位として扱う関数は、SQL には含まれていない。だがISO 8601 の一部ではある。米国ではそれほど一般的ではないが、北欧やEUでは多くの商業的アプリケーションが週を時間の単位として使っていることが多い。

「週01」は、その年の木曜日を持っている最初の週として定義される。これは1月の最初の4日を含むということと同義だ。また別の言い方をすると、新年最初の週とは、新年の最初の曜日の過半数を持っている週、ということでもある。「週01」はまた、前年の日を含むこともある。そのため、週と年は完全には同期しない。余談だが、米国のカレンダーは日曜日を左端の列に配置する。一方、ヨーロッパなどでは日曜日を右端の列に配置する。ISO 標準では、日曜日が最後の日とされている。

標準表記では、続く2桁の数字が週番号を表すために、文字Wを使う。年と週番号は「1999-W01」のようにハイフンやスペースで区切ることもできるが、必ずしも必須ではないので、「1999W01」という表記でもよい。さらに曜日を1から7で表して、「2010-01-05」（火曜日）を「2010-W01-2」や「2010W012」とする表記もある。

ISO標準では、週番号の範囲は明示的に示されていない。だが少し考えれば、範囲は01～52 または53であることがわかる。ただし、このルールには1つだけ例外がある。グレゴリオ暦が導入されたのは1753年だが、この年は日数が365日よりも少なかったため、週の数も52未満だった。

SQL Serverユーザーはこの点に慎重でなければならない。なぜなら、SQL Serverの週番号の振り方はISO 標準に従っていないからだ。この問題を回避するために、SQL Server は`ISO_WEEK`というオプションを持っている。

さらに、2つの日付の間の週数を計算するのも簡単ではない。たとえば、開始日として「2012-12-06」、終了日として「2012-12-19」を考えよう。この場合、17日は火曜日ではないため、答えとしては2 週とみなすべきだろう。この問題を解く考え方は数多くあり、また解は何をもって「週」とみなすかによって変わる。考える際のヒントとして、以下に、2012 年12月のカレンダーを示そう。

Su	Mo	Tu	We	Th	Fr	Sa
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

1. 6日から13日までを1 週目、それ以降から20日までを2 週目とみなす。この考え方だと、19日は20日に届いていないので、答えは1週となる。
2. 含まれる日数が14日なので、14/7＝2週とみなす。
3. 開始日または終了日のどちらかをカウントしない。実際これは、年のときに使うルールだ。たとえば、1999年と2007年の間に何年あるかと聞かれたら、多くの人は9年ではなく8年と答えるだろう。単純に大きいほうから小さいほうを引いている。これを日にも適用すれば、19–6＝13となる。13/7＝1.857142…となり、切り捨てで1になる。
4. 上記3.の結果を四捨五入して2とする。
5. 日曜から土曜までの完全な週の数で数える。すると9日から15日までの1つしかない。このルールに従うと、たとえば3日が開始日で終了日が21日だとしても、18日（または19日）が含まれるにもかかわらず、やはり1 週しか含まないという、いささか奇妙な結果が生じる。

ちょっと考えただけでも、複雑さにうんざりするだろう。標準SQLが週ではなく日を単位とするのを好む理由がよくわかる。

38.6.1　曜日によるソート

次のトリックは、クレイグ・S・マリンズが考えた。曜日の名前を保持するテーブルがあるとする。

```
CREATE TABLE Foobar
(day_name CHAR(3) NOT NULL
     CHECK (day_name
       IN ('SUN', 'MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT')));

INSERT INTO Foobar
VALUES ('SUN'), ('MON'), ('TUE'), ('WED'), ('THU'), ('FRI'), ('SAT');
```

　このテーブルのデータを正しくソートするにはどうすればよいだろう？ つまり、日曜日が最初で、以下、月曜、火曜、水曜……、という順序にソートしたいのだ。おそらく多くの人が最初に思いつくのは次のようなクエリだろうが、これはうまくいかない。

```
SELECT day_name
  FROM Foobar
 ORDER BY day_name;
```

　このクエリはアルファベット順にソートしてしまうので、次のような結果になる。

```
day_name
========
FRI
MON
SAT
SUN
THU
TUE
WED
```

　もちろん、1つの解決策は数値型の列をテーブルに追加して、それでソートすることだ。だが、ここではそれ以外の方法を考えよう。なかなか見事なもので、データベースに対する変更が必要ない。

```
SELECT day_name,
       POSITION(day_name IN 'SUNMONTUEWEDTHUFRISAT') AS day_nbr
  FROM Foobar
 ORDER BY day_nbr;
```

結果

```
day_name day_nbr
================
SUN             1
MON             4
TUE             7
WED            10
THU            13
FRI            16
SAT            19
```

曜日の名前から曜日の番号に変換したいならば、次のように型変換を行えばよい。

```
SELECT day_name,
  CAST (POSITION(day_name IN 'SUNMONTUEWEDTHUFRISAT')/3 AS INTEGER) + 1 AS day_nbr
  FROM Foobar
 ORDER BY day_nbr;
```

結果

```
day_name day_nbr
================
SUN             1
MON             2
TUE             3
WED             4
THU             5
FRI             6
SAT             7
```

お気づきのように、このトリックは同じく3文字で略す月名についても適用できる。

38.7

時間データのモデリング

時間は本質的に連続体であり、ISOでは半開区間として定義されている。最良のアプローチは、イベントの履歴を(start_time, end_time)のペアで表すことだ。これは、列で表されるイベントが与えられた期間において成立していることを意味する、データの状態遷移モデルである。このモデルをうまく使うには、(start_time, end_time)ペアが重複しないよう適切な制約を付与する必要がある。

end_time がNULL であることは、「終わっていないイベント」を表す。例としては、ホテルの宿泊客がまだチェックアウトしていない、といったものだ。したがって、あるエンティティの履歴にはNULLは存在しても1つだけであることは明らかである。

```
CREATE TABLE FoobarHistory
(foo_key INTEGER NOT NULL,
 start_date DATE DEFAULT CURRENT_DATE NOT NULL,
     PRIMARY KEY (foo_key, start_date),
 end_date TIMESTAMP, -- NULLは現在進行中を意味する
 foo_status INTEGER NOT NULL,
     CONSTRAINT started_before_ended
       CHECK(start_date < end_date),
     CONSTRAINT end_time_open_interval
       CHECK (end_date = CAST(end_date AS DATE)
                + INTERVAL '23:59:59.999' HOUR TO SECOND),
     CONSTRAINT no_date_overlaps
       CHECK (NOT EXISTS
                    (SELECT *
                       FROM FoobarHistory AS H1, Calendar AS C1
                      WHERE C1.cal_date BETWEEN H1.start_date
                        AND H1.end_date
                      GROUP BY foo_key
                     HAVING COUNT(*) > 1)),
    CONSTRAINT only_one_current_status
       CHECK (NOT EXISTS
               (SELECT *
                  FROM FoobarHistory AS H1
                 WHERE H1.end_date IS NULL
                 GROUP BY foo_key
                HAVING COUNT(*) > 1)));
```

このDDLで使用しているCalendarテーブルについては、次節で説明する。テーブルレベルのCHECK制約はまだ多くの実装では一般的ではない。したがっていまは、同じ制約をトリガーで施行する必要があるだろう。

だが、このDDLにおける本当のポイントはそこではない。start_dateがDATE型であるため、TIMESTAMP型に変換した場合に「00:00:00.00000」がセットされることだ。一方、end_timeはTIMESTAMP であるため、次の日の直前の値を入れることができる。こうすることの利点は、次節で見るように、BETWEEN 述語を使った検索が可能になることだ。これをビューに入れて、両方の日付をDATE型として保持し、end_timeの時間を保持するための列を追加する方法も考えられる。

実際のところ、このコードが動作するかどうかはかなり実装依存なので、トリガーで(start_time, end_time)の状態を正しく保つほうが現実的かもしれない。

38.7.1 期間のペア

もしテーブルが(start_time, end_time)のペアを持っていなかった場合、それらを作るために自己結合のクエリを使うか、あるいは状態遷移が必ず正しい順序で行われることが前提のデータを格納する必要がある。たとえば、以下のテーブルがあった場合、現在の状態コードが10のプロジェクトをすべて選択するクエリを書くにはどうすればよいだろう？

```
CREATE TABLE Projects
(project_id INTEGER NOT NULL PRIMARY KEY,
 project_name CHAR(15) NOT NULL);

INSERT INTO Projects
VALUES (1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'),
       (5, 'E'), (6, 'F'), (7, 'G');

CREATE TABLE ProjectStatusHistory
(project_id INTEGER NOT NULL
     REFERENCES Projects(project_id),
 project_date DATE DEFAULT CURRENT_DATE NOT NULL,
 project_status INTEGER NOT NULL,
     PRIMARY KEY (project_id, project_date));

INSERT INTO ProjectStatusHistory
VALUES (1, '2012-01-01', 9), (1, '2012-02-02', 10),
       (2, '2012-09-22', 8), (2, '2012-09-30', 10), (2, '2012-11-11', 11),
       (3, '2012-12-24', 8), (3, '2013-02-03', 9);
```

次の解はデヴィッド・ポータスによるものだ。

```
SELECT P.project_id, P.project_name
  FROM Projects AS P
 WHERE EXISTS
       (SELECT 1
          FROM ProjectStatusHistory AS H
         WHERE H.project_id = P.project_id
        HAVING MAX(CASE WHEN H.project_status = 10
                        THEN project_date END) = MAX(project_date));
```

結果

```
project_id  project_name
=======================
         1  A
```

指定した日付（:mydate）において状態コードが10だったプロジェクトを選択するクエリは、次のようになる。

```
SELECT X.project_id
  FROM (SELECT P1.project_id, P1.project_date AS start_date,
               MIN(P2.project_date) AS end_date
          FROM ProjectStatusHistory AS P1
                 LEFT OUTER JOIN
                   ProjectStatusHistory AS P2
            ON P1.project_id = P2.project_id
           AND P1.project_date < P2.project_date
         WHERE P1.project_status = 10
         GROUP BY P1.project_id, P1.project_date)
            AS X(project_id, start_date, end_date)
 WHERE :mydate BETWEEN X.start_date
                       AND COALESCE (X.end_date, CURRENT_DATE);
```

導出テーブルXは、プロジェクトの(start_time, end_time)ペアを作っている。COALESCE関数は、NULLは終了日が未定であることを想定したものだ。状況によっては、CURRENT_DATEの代わりに入力した日付を条件に使いたいこともあるだろう。

```
WHERE :my_date BETWEEN X.start_date
  AND COALESCE (X.end_date, :my_date)
```

38.8

カレンダー補助テーブル

補助テーブルは、SQLの限られた計算力では計算が難しい演算を行うための補助ツールである（詳細は第8章参照）。このテーブルはモデルの一部ではないため、ER図には現れない。

カレンダーは非常に不規則なデータであるため、計算だけで対応するのは難しい。感謝祭やラマダン、その他の太陽暦／太陰暦の祝祭日の日付を求めるアルゴリズムをちょっと調べてみれば、その複雑さに頭を抱えるだろう。

米国の証券取引委員会は、仲介業務は3 営業日以内に完結すること、というルールを定めている。この営業日というのは、日曜日、土曜日、およびニューヨーク証券取引所が指定している祝日を含まない日のことだ。日曜日と土曜日の日付を求めるだけなら、多くのSQL製品の関数ライブラリにあるだろう。だがそれでは祝日は求められない。しかも実際には、ニューヨーク証券取引所は国家の非常事態のときに閉まることがある。

カレンダー補助テーブルには、2つの様式がある。最初のものは、単一の日付を1行として保持するタイプだ。

```
CREATE TABLE Calendar
(cal_date DATE NOT NULL PRIMARY KEY,
 julian_day INTEGER NOT NULL
     CONSTRAINT valid_julian_day
       CHECK (julian_day BETWEEN 1 AND 366),
 business_day INTEGER NOT NULL
     CHECK (business_day IN (0, 1)),
 three_business_days DATE NOT NULL,
 fiscal_month INTEGER NOT NULL
     CONSTRAINT valid_month_nbr
       CHECK (fiscal_month BETWEEN 1 AND 12),
 fiscal_year INTEGER NOT NULL);
```

Calendar：カレンダーテーブル　cal_date：日付　julian_day：ユリウス通日　business_day：営業日　three_business_days：3営業日目の日付　fiscal_month：会計月　fiscal_year：会計年

これは静的テーブルになる可能性が高いので、10年から20年の期間のデータを保持してもいいだろう（20年でも7000行程度の小さなテーブルだ）。

2番目の様式は、期間を1行として保持するテーブルだ。これは前節で見た制約のいくつかが必要になる。

```
CREATE TABLE EventCalendar
(event VARCHAR(30) NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 end_date TIMESTAMP NOT NULL,
     CONSTRAINT started_before_ended
       CHECK(start_date < end_date),
     CONSTRAINT end_time_open_interval
       CHECK (end_date = CAST(end_date AS DATE)
         + INTERVAL '23:59:59.99999' HOUR),
     CONSTRAINT no_date_overlaps
       CHECK (...),
     CONSTRAINT only_one_current_status
       CHECK (...));
```

EventCalendar：イベントカレンダーテーブル　event：イベント　start_date：開始日
end_date：終了日

イベントは流動的になる傾向があるので、制約を多く付与しておくほうがよい。カレンダーテーブルにロードするデータは、スプレッドシートを使って作るのが便利だ。スプレッドシートは、SQLよりも多くの時間関数を持っていることが多い。

38.8.1　イベントと日付

カレンダーテーブルを使うと、複数のイベントの重なりを見つけるのは簡単だ。図38.1を見ればこのクエリの意味は明らかだろう。

```
SELECT event_id
  FROM Events
 WHERE :my_date BETWEEN start_date AND end_date;
```

終了期間を含まないようにしたいなら、次のように書けばよい。

```
SELECT event_id
  FROM Events
 WHERE :my_time >=start_time
   AND :my_time < end_time;
```

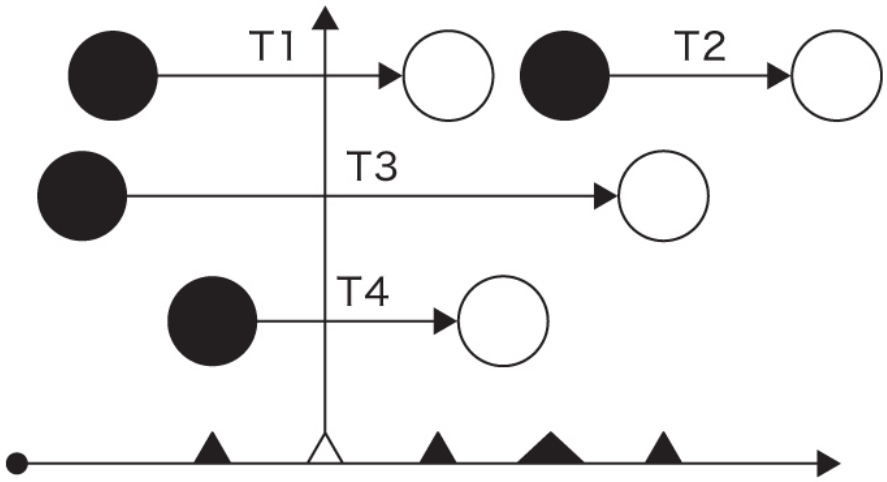


図38.1

38.9

2000 年問題

2000 年にまつわる特殊な問題が、かつてシステム業界をにぎわした。これは、本書においても一節を割いて言及するに値する問題である。もちろん、いまでは「Y2K危機」や「千年に一度のバグ」は終わった。だが、これはいまだに一考の価値のある問題なのだ。

コンピュータシステムにおける2000年問題は、主に次の4つの問題として表れた。以下では、お粗末すぎる実装に起因する問題4を除く3つについて解説する。

問題1　2000年が多くの0を持っている
問題2　2000年が閏年である
問題3　2000年が前世紀の最後の年だった
問題4　多くの日付フィールドが実際には日付ではなかった

38.9.1　ゼロ

問題1（2000年が多くの0を持っている）を「多すぎるゼロ問題」、または「走行計問題」と呼びたい。というのも、これはハードウェアレベルの問題だったからだ。実際にはこれは千年に一度の問題というより、単に日付の計算が不適切なために起きるものだ。もし年を1 世紀しか持たないようなフォーマットで設計していると、2000年を迎えたときにメーターが「1周」して、0 に戻ってしまう（ちょうど車の走行計のように）。すると、アプリケーションからは1900年（または2000年以外の年）と認識されてしまうのだ。

この問題はハードウェアとOSに関連したシステムクロックという、ユーザーから目に見えないところで起きている。こうした問題に対する情報は不完全なので、特定製品の新しいリリースが出るたびに自分でチェックすることになる。

この「ゼロ問題」は別の微妙な形をとって現れることもある。ハッシュや乱数のアルゴリズムは、内部的にシステム日付を入力に使っていることがある。この入力に0 が渡されると、0除算によってプログラムがエラーになることがある。

また、メインフレームで実際に起きたことだが、ユニシス2200は、1996年の最初の日を迎えるとエラーになった。これは、年フィールドの8ビット目（符号付き整数）が1になったからだ。このときは、幸運なことにベンダーは対策をとること

ができたが、ほかにも同じ作りになっている製品が何か知りたくはないだろうか？ 本当に怖いのは古いインテルベースのPC だ。「走行計」が1周すると、DOSは多くの場合は1980年（ときには1984年。BIOSチップによる）にジャンプする。Windows 3.1は普通1900 年にジャンプする。もし古いPCを持っていれば、自分で試してみてほしい。システム日付を「2005-12-31 at 23:59:30」に設定して、30秒後にどこへ飛ぶか見てみるだけだ。おそらく、時計は「12:00 AM」を指し、日付は「01/01/00」を指すだろう。ところが、新しく作ったファイルの日付は、驚いたことに1984年か1980年になるはずだ。

この問題は、アプリケーションプログラムに対しても影響を与えたが、その影響は少し意外な形で現れた。IBM PCのMS-DOS 6上で動くQuicken Version 3を例に考えよう【訳注7】。直接「2000-01-01」という日付を入力すると、予想どおり1980年または1984 年と解釈される。だが妙なことに、「2005-12-31」とセットしてからしばらく時間が経つと、2000 年と解釈される。Quickenは、この変化を1900年ではなく1901年の1月1日になったと考えたのだ。このように、古いソフトウェアの2000年問題に対する反応は非常に奇妙なものがある。

38.9.2　閏年

問題2（2000 年が閏年である）に遭遇した人は必ず驚く。1 年が365日と¼日であることは小学校でも習うだろう。そのため、4年に一度端数が積み上がって閏年が生じることになる。だがこの話には続きがある。1年は厳密には365.2422日なのだ。したがって、まず100年ごとに失われる1日（365.25–365.2422）を調整するために、100 年周期で平年とする。だがこれでもまだ1年につき0.0022の誤差が残るため、400年周期で閏年とする。この結果、閏年は次の3つのルールで定義される。

ルール1　西暦年が4で割り切れる年は閏年
ルール1　ただし、西暦年が100で割り切れる年は平年
ルール1　ただし、西暦年が400で割り切れる年は閏年

ある年が閏年かどうかを調べるユーザー定義関数は次のものだ。

```
CREATE FUNCTION Test_Leapyear (my_year IN INTEGER)
RETURN CHAR
IS
BEGIN
    RETURN CASE WHEN MOD(my_year, 400) = 0
        THEN 'Yes'
        WHEN MOD(my_year, 100) = 0
        THEN 'No'
        ELSE CASE WHEN MOD(my_year, 4) = 0
            THEN 'Yes' ELSE 'No'
        END
    END;
END;
```

もう少しコンパクトなものがよければフィル・アレクサンダーが考えた次のインラインコードを検索条件で使ってもよい。

```
( MOD(my_year, 400) = 0
  OR (MOD(my_year, 4) = 0
  AND NOT (MOD(my_year, 100) = 0)))
```

このシンプルなアルゴリズムを知らない人々によって、多くのプログラムが書かれてきた。といっても別に読者の会社のレガシーなCOBOLプログラムのことではない。私が言いたいのは、結構な値段のする有料パッケージ製品のことだ。Lotus、Excel、Quattro Proの最初のリリースの日付関数は「2000-02-29」を正しく扱えなかった。Lotusは単純にエラーを吐いた。その他はいずれも「Lotus互換」をうたっていた。Microsoft Excel for Windowsは、バージョン4になって、「2000-02-28」の次は「2000-02-29」と正しい値を返すようになった。だが、「1900-02-28」の次は閏年でないにもかかわらず相変わらず2 月29 日を返していた。Microsoft Excel for Macintoshは1900年から1903年までを扱えなかった。

読者は自分の使っている入力ソフトやスプレッドシート、デスクトップデータベース、カレンダーといったソフトウェアに2000 年問題がないか調べたことはあるだろうか？ 試しに2000-02-29と打ち込んで計算を行ったとき、どのような挙動になるか見てほしい。

ネットワークで相互に結ばれたシステムにおいては、問題は一層悲惨だった。ネットワークにつながれている1つのノード上で2000 年を閏年として扱わないプログラムが動いていれば十分だった。それだけでネットワーク全体が一日使いものにならなかった。後日あらためて実行するのでは遅いトランザクションがあっ

た場合は目も当てられなかった。北米とヨーロッパだけでATMネットワークに何台のノードが動いていたか見当もつかない。

38.9.3　千年に一度

問題3（2000 年が前世紀の最後の年だった）を最後に持ってきたのは、これが最もメディアで取り上げられて有名になったからだ。プログラマは長い間、日付を完全な形式では保持してこなかった。実際は、年を2桁で保持していたことが多い。これは前世紀の最終年を扱えなかった。西暦の最初の千年は2000 年で終わり、次の千年は2001 年から始まる――アーサー・C・クラークがこの年を本のタイトルに使った――のはこのためだ。

もしプログラマがストレージの節約のために正確さを犠牲にしようとしなければ、この問題を回避するためISO標準のフォーマットを使っただろう。だが現実には、年に対する比較がおかしなことになるプログラムが続出した。1992 年に組んだ30 年ローンが終わるのは2022 年だが、年を2桁で保持していると、返済期間を求める計算は以下のようになる。

22 – 92 = –70年

返す前から完済しているではないか！ こんなローンがあったら誰でも入りたいだろう。

この問題にCOBOLプログラマがどのように対応したか、知りたい人もいるかもしれない。その方法はこうだ。2 桁の年が30未満であれば、2000を足し、そうでなければ1900を足して4 桁の年にしたのだ。この閾値は、プログラムが書かれた時期と、いつまで動く予定だったかによって20、30、40 年と変化する。COBOLがシステムの世界でどれだけの長命を保っているかを知れば、これは無視できない問題だ。

リサーチ会社Ovumの主任研究員グレイ・バーネットは2005 年に、COBOLはすべての金融トランザクションの90%を占めており、一般のトランザクションの75%を占めているというレポートを出している。COBOLについての情報は以下で見つけられる。

The COBOL Center
http://www.infogoal.com/cbd/

在庫管理のプログラムは、保管期間の切れた物品を倉庫から除外する。たとえば自動車業界では10年程度の保管期間が求められる。サービススケジュールと製造の日付が正しく求められないと、製品の保証期間が正しく設定できない。クーパーズ＆ライブランドLLPのベニー・ポペックによれば（Xenakis, 1995）、「この問題は非常に大きいため、我々はこれを原因とするバグは通常のソフトウェア保守契約の範囲外であると考える。この問題に対しても我々が責任を負うべきだと考えるクライアントには、免除条項を発動し、アウトソーシング契約を終了するつもりだ。」ポペックはさらにこう続けている。「我々が見た限り、多くのクライアントがこの問題への対応を行わなかった。我々が話したCIOの一人は、来年が定年なのでこの問題に対処する必要はないのだと言っていた」

この問題の全貌をつかむには、単に日付型のデータを探すだけでは不十分だ。タイムスタンプは、エンコーディングスキーマの中に埋め込まれていることが多いからだ。もし2桁の年がシリアルナンバリングの上2 桁に使われていたら、そのシリアルナンバーを使うすべてのプログラムが失敗する。磁気テープのライブラリを使っている人は、テープのラベルを見てもらいたい。年が2 桁で記述されていないだろうか。多くのメインフレーム企業では、日付に5 桁のコードが使われており、テープ管理ソフトウェアは、もしテープを永年保管する場合は保管期間のラベルに99365 を設定するという規約を持っていた。この方法は、2000 年が始まり保管ラベルに00001を使わざるを得なくなったときに破綻する。

38.9.4　レガシーデータにおける奇妙な日付

レガシーデータにおける日付の問題についてはランドール・L・ヒッチンスの論文で議論されている（Hitchens, 1991）。また、これは私がかつて論じたテーマでもある（Celko, 1981）。ヒッチンスは非標準的な日付フォーマットについて論じているが、この問題は彼が考えているより複雑だ。日付は日付フィールドだけではなく、他の場所にも隠れている。一般的なのは、シリアルナンバーとコンピュータが生成する識別子だ。

1960 年代の前半、アトランタの小さな保険会社が、さらに規模の小さな、南部で貧しい人々を対象に埋葬保険を扱う会社を買収した。その約款では、契約者は週または月単位で数ドルの保険料を払うことで、埋葬費用が保証された。営業店は地元の葬儀会社だった（こういう営業は現在は違法だ）。その埋葬保険の会社は、約款の識別子に、西暦の下2桁で始まる8桁の数字を使っていた。年の計算をするときは、その2桁を対象に行っていた。

やがて月日は流れ、その会社はシンプルなファイルシステムから階層型データベースを使うようになった。識別子はレコードの一意キーとして使われるようになった。システムは単純に新しい連番を生成し、相変わらず約款のバージョン番号として使っていた。数十年は、それでも何も起きなかった。

20 年が経過した頃、約款のバージョンは100 ほどになっていた。その間、重複キーをチェックするプログラムを書いた人はいなかった。問題が起きなかったからだ。そしてある日、保険会社は初めての重複キーを手にすることになった。ある場合はデータベースがクラッシュし、ある場合は子レコードが間違った親レコードにぶら下がることになった。後者のケースのほうが厄介だった。間違った受取人に保険料を支払うという致命的な間違いが起きたからだ。この会社が幸運だったのは、古い約款が使われていることに比較的早く気づく人がいたことだ。問題の解決には数か月を要した。

このケースでは、日付が一意な識別子の一部に使われていた。同様のスキームを利用したプログラムは数多い。たとえば2 桁の年をシーケンスに使っている場合は、1999より大きな値を採番できない。

38.9.5　2000 年問題の影響

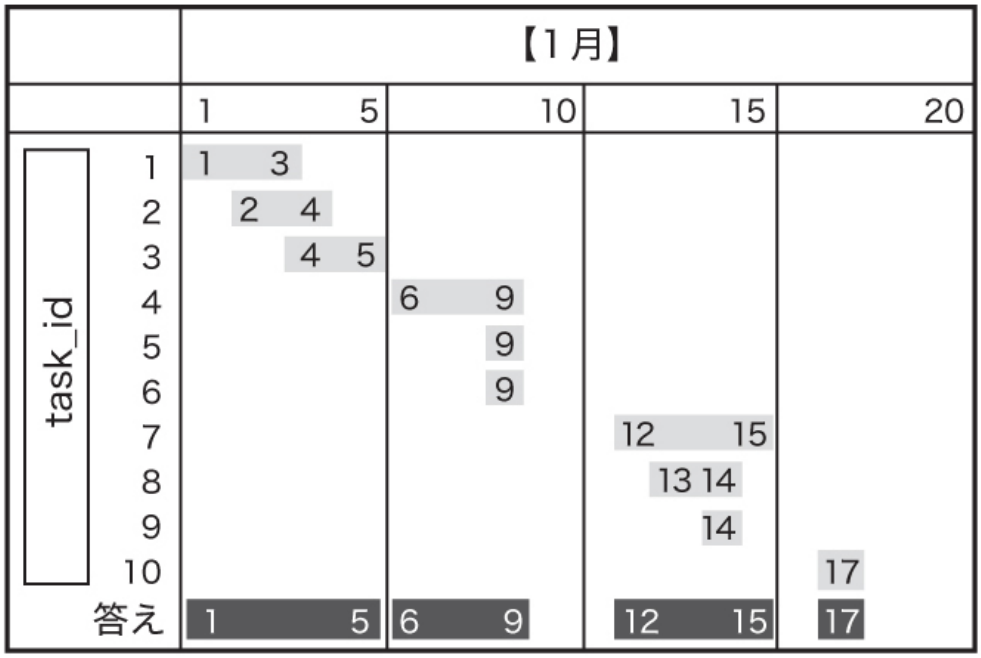
2000 年問題は終わった。その修正にかかったコストについては、さまざまな試算が行われているが、政府や企業による総支出は2000 億ドルから5000 億ドルと言われている。2000 年問題の結果、多くのISO 標準（SQLも含む）がISO 8601フォーマットの日付をサポートすることになった。もう1つの良い副作用は、人々が自分たちのデータを見直し、その品質を意識するようになったことだ。多くの企業は、この問題が頭をよぎらなければ、わざわざデータの洗い直しはしなかった

だろう。標準SQLにとってもまた、時間型の見直しに時間を費やすきっかけとなった。

【訳注1】　復活祭はもともと太陰暦に基づいて決められた祝日のため、太陽暦では年によって日付が変わる（だいたい3月から5月の間）。また現代においても、「祝日7」で触れているように、ユリウス暦を用いる地域とグレゴリオ暦を用いる地域で日付がずれる。

【訳注2】　クリスマスの翌日で、クリスマスのプレゼントの箱（box）を開ける日という意味なので、スポーツのボクシングとは関係ない。

【訳注3】　この問題は、『SQLパズル』の「パズル59 期間を結合する」でも取り上げられている。サンプルデータをガントチャート風に表すと以下のようになる。



【訳注4】　このクエリはPostgreSQL、Oracleなどではまだ動作しない。

【訳注5】　PostgreSQLでは、`TIMESTAMP`型の列に時間フィールドが24のデータを`INSERT`すると、自動的に0に変換される。また、Oracleでは`INSERT`時にエラーになる。

【訳注6】　1970年1月1日を起点とした経過秒数で日時を表すUNIX時間などが代表例である。

【訳注7】　Quickenは、Intuit社が開発する個人向け会計ソフト。確定申告などに使用する。

39

SQLの最適化

Optimizing SQL

　すべてのSQL製品で常に最適化できるようなコーディングのルールは存在しない。オプティマイザは下位のアーキテクチャに依存しており、その動作は端的に言って千差万別だ。とはいえ、一般的に言えることがまったくないわけでもない。ただ、ここで述べることを鵜呑みにはせず、自分でテストするのを忘れないでほしい。1 つの実装でパフォーマンスを改善する方法が、他の実装では効果がなかったり、逆効果だったりすることもある。あるいは同じ実装であっても、次のリリースで動作が変わることもある。

　オプティマイザにはコストベースとルールベースの2 種類がある。ルールベースのオプティマイザ（たとえば、バージョン7以前のOracle)は、書かれた順番でクエリをパースした。もちろん、ある程度の同値な構文変換を行いながら。基本的には、ここにオプティマイザの出番はない。ルールベースはいまとなっては過去の遺物だが、小規模のインメモリDB では統計情報と余計なコードのオーバーヘッドを許容できないため、いまだルールベースを採用しているものもある。もしデータベース全体がメモリに載るならば、I/Oコストを減らすための伝統的な最適化がたいして必要ないのも事実だ【訳注1】。

コストベースのオプティマイザはクエリだけでなく統計情報やキャッシュ内のデータも考慮してクエリを実行する。その際、どのようにデータへアクセスし、どのようなデータをメモリに乗せるか、どのようなソート方法を使うか、といったことが判断される。（常にではないが）たいていの場合は、限られた情報しか持たない人間よりも良い実行計画を立てる。

ルールベースの時代は、FROM 句のテーブルは左からサイズの昇順に並べるのが鉄則だった。ルールベースではこの順番で結合の駆動表が決まったからだ（「駆動表は小さく」がチューニングの原則である）。現在では、テーブルの結合順序は統計情報からオプティマイザが判断するので、基本的にこのようなルールに従う必要はない。ただし、テーブル数が増えて組み合わせの数が増えると最適な順序で結合されなくなることもある。

39.1

アクセス方法

この議論の前提として、何らかのディスクストレージを使用しているとする。数年以内に、SSDがブレイクスルーを果たしてこの状況は劇的に変わるだろう【訳注2】。

ディスクにおいては、4つの基本的なデータへのアクセス方法がある。

方法1 テーブルのフルスキャンまたはシーケンシャルリード
方法1 ツリーインデックス経由のアクセス
方法1 ハッシュインデックス経由のアクセス
方法1 ビットベクトルインデックス経由のアクセス

39.1.1 シーケンシャルアクセス

テーブルスキャンはテーブルの全データを物理的な格納順序でシーケンシャルに読み出す。ほとんどのデータベースは削除された行を物理的には残しているので、この方法で読み込まれるデータはしばしば必要なデータよりも大きくなる。削除された行を物理的にも削除するには、データベースのユーティリティやコマンドを実行するか、定期実行するよう設定する必要がある。そのようなデータベースの再編成を行うと、フルスキャンの性能が良くなる。

今日では、多くのSQL 製品がテーブルをパーティション化することができる。その場合は、アクセスをパラレル化することも可能だ。これはANSI/ISO 標準の機能ではないので、詳細は製品マニュアルを読んでほしい。テーブルがパーティション化されていると、オプティマイザはクエリが必要とするパーティションだけをスキャン

することが可能になる。この機能は、カラム指向データベースでは利用できない。これは、データが行ではなく列単位で格納されるため、行による分割が意味をなさないためだ。

39.1.2　ツリーインデックス

インデックスアクセスは、一度のアクセスで1行または1ページ（1ブロック）を返す。多く使われているインデックスはBツリー、B+ツリー、T ツリーといったあたりだ。もちろん、テーブルにインデックスが存在しなければ、インデックスアクセスを利用することはできない。インデックスが存在する場合でも、条件の形式やオプティマイザの判断によってはインデックスが使われないこともある。

インデックスには、シーケンシャルなものと非シーケンシャルなものがある。シーケンシャルなインデックスは、テーブルの行が物理的にソートされたテーブルを持つ。当然のことながら、シーケンシャルなインデックスは、テーブルに1つしか存在しえない。テーブルがソートされていることの副作用として、スキャンの結果がその順序でソートされていることが多い。また、シーケンシャルなインデックスは、列の重複値を物理的に同じ領域に配置する。したがって集約関数も高速化される。このようなシーケンシャルなインデックスの例としては、Sybase/SQL Serverのクラスタ化インデックスがある。Oracleは「クラスタ化」という語を複数のテーブルにマッチする行を含む単一のデータ領域という意味で使っているので、注意してほしい。

最初にシーケンシャルインデックスが実装されたのは、IBM の懐かしいISAM（Indexed Sequential Access Method）ファイルシステムの最初のバージョンであった。懐かしいと言ってもこれはまだメインフレーム上で使われているのだが。

テーブルサイズが大きくなるにつれて、インデックスの木も深くなり、目的のデータを見つけるのに時間がかかるようになる。これはツリー状のインデックス全般に該当する。

39.1.3　ハッシュインデックス

ハッシュ関数を書くのは簡単ではない。基本的な考え方としては、入力の値を受け取って物理ストレージのアドレスを返せば良いのだが、もし複数の入力に対して同じハッシュ値が返されると（「衝突」や「コリジョン」と呼ぶ）、2 つの値が同じアドレスに存在することになってしまう。これを避けるには別のハッシュ関数でもう一度変換をかけるなどの必要がある。

非常に単純な例を挙げよう。整数で表現されるキーがあるとする。これを5 で割った余りをハッシュに使う。ハッシュ値は{0, 1, 2, 3, 4}のどれかになる。キーの値が42、35、120であれば、ハッシュ値は2、0、0になる。キーが42の行については問題ない。物理的なアドレスが2のデータを探しにいけば見つかる。だが、キーが35 の行を探しにいくと、キーが120 の行も見つかってしまう。

コリジョンを解決する方法は2 つある。1 つは、ハッシュテーブルの要素をポインタチェーン化して、本当に欲しいデータにたどり着くまでそれをたどっていく方法だ。もう1つが、別のハッシュ関数を適用して別の場所に格納する方法。ハッシュ関数は高速で、呼び出

しコストも小さい。1つのキー値を変換するのに5回以上の呼び出しが必要なことはほとんどない。上のシンプルな例であれば、6か7の余りを使えばコリジョンは回避できる。

　もし一意な列があった場合、理想的なのは最小完全ハッシュ関数が使える場合だ。キー値から一意な物理的なアドレスを導くことができ、ハッシュテーブルにも無駄が生じない（最小完全ハッシュについては第23章参照）。次に良いのは、完全ハッシュ関数が使える場合だ。キー値から一意な物理的なアドレスを導くことができるが、ハッシュテーブルには使われない領域が生じる。

　一意でない列に対するハッシュ関数は、メモリに収まるくらいの小ささを持つバケットを対象にハッシュしなければ性能が悪い。Teradata のエンジンはハッシュを基礎にしているが、すべての行が最大2回の探索で見つかり、90%以上のアクセスが一度の探索で見つかる。

39.1.4　ビットベクトルインデックス

　あるエンティティの特定の行の特定の列が特定の値を持っていることは、ベクトルまたは配列における単一のビットで表現できる。検索条件は、その配列に対する真理値のビット演算で制御できる。これらのテクニックは、大規模データに対する検索を高速化する優れた方法である。

39.2

インデックス設計

インデックスには2種類ある。主キーのインデックスと二次インデックスだ。主キーのインデックスは、文字どおり主キーまたは一意制約のある列に対して付与されるインデックスである。実装によっては主キー制約や一意制約を作ると暗黙にインデックスが作られるものもある。二次インデックスは、パフォーマンス改善のために任意の列に付与されるインデックスだ。最適なインデックスの組み合わせを求める問題はNP完全であることが知られている（NP完全については第5章参照）。詳細が知りたければ、以下の論文を読んでほしい。

Comer, D. (1978).“ The Difficulty of Optimum Index Selection.” *ACM Transactions on Database Systems*, 3(4): 440–445.

Paitetsky-Shapiro, G. (1983).“ The Optimal Selection of Secondary Indexes in NP-Complete.” *SIGMOD Record*, 13(2): 72–75.

といっても、特定のデータベースにおける特定のクエリの集合に対する最適なインデックスを見つけることができないわけではない。すべてのデータベースのすべてのインデックスについて最適なインデックスの組み合わせを見つけるプログラムを書くのが難しいというだけだ。

テーブルにインデックスをベタベタと張りすぎると、挿入、更新、削除といった操作が遅くなる。これは、基底テーブルが変更されたらインデックスも変更しなければならないため、オーバーヘッドになるからだ。また、インデックスが多すぎるとオプティマイザ

が混乱して、どれを選べばいいのかわからなくなってしまうという弊害もある。

DML文を高速化するTipsはいくつかあるが、すべてのヒューリスティクスがそうであるように、あらゆる製品のあらゆるケースに対応できるものではない。だが、勝算の高い選択肢には低い選択肢よりも多くの賭け金を払ってもよいだろう。公平を期して言うならば、今日ではほとんどのオプティマイザは、そうした高速化の判断を内部的に行う程度には賢くなっている。

39.2.1　シンプルな検索条件

可能な限り検索条件はシンプルにすべし。複雑な式や関数呼び出しを含む検索条件にどのインデックスを使うべきか迷うのは、人間だけではない。

関数は呼び出されるたびに評価されるので、関数が使用された列にインデックスを使用することはできない。例外は、実装が関数インデックスを持っている場合だけだ【訳注3】。インデックスの構文は実装によって異なるが、たとえば以下のような単純なインデックスがあるとする。

```
CREATE INDEX Foobar_idx ON Foobar (x);
```

三角関数を適用したクエリを考えてみよう。

```
SELECT x, sin(x), a, b, c
  FROM Foobar
 WHERE sin(x) = 0.08715; --x = 5
```

この場合、オプティマイザはインデックスFoobar_idxを使えない。だが、もし次のように関数インデックスを宣言したら話は別だ。

```
CREATE INDEX Foobar_Sin_idx ON Foobar (sin(x));
```

このインデックスは使われるだろう。しかし今度は、関数インデックスは次のようなクエリに対しては使われない。

```
SELECT x, sin(x), a, b, c
  FROM Foobar
 WHERE x = 5;
```

列が数式や文字列式の一部に含まれた場合、オプティマイザはインデックスを使うことができない。それ以外にも、否定条件でもインデックスは使われないことが多い。返却する行数が多すぎて、オプティマイザがシーケンシャルスキャンのほうが有利だと判断するのだ。これは常にそうというわけではない。たとえば、アイルランドに住む非カトリック教徒を求めるクエリは、普通こう書くだろう。

```
SELECT last_name, first_name, religion_name
  FROM Ireland
 WHERE religion_name <> 'Catholic';
```

この不等号を分解すると次と同値である。

```
SELECT *
  FROM Ireland
 WHERE religion_name < 'Catholic'
    OR religion_name > 'Catholic';
```

この書き方をした場合は、インデックスが使われるかもしれない。

インデックスで注意が必要なもう1 つの点は、IS [NOT] NULL 述語を避けることだ。というのも、NULLを保持する方法は実装によって異なるが、必ずしも物理的に列の値として格納されているとは限らないからだ。たとえば、CHAR(3)の列があるとして、この列がNULL 以外のデータを見つけようとする場合、普通ならば次のように書くだろう。

```
SELECT sale_nbr, alpha_code
  FROM Sales
 WHERE alpha_code IS NOT NULL;
```

だが、これはインデックスが利用されない可能性が高い【訳注4】。

39.2.2　シンプルな文字列式

同様に、文字列式はコストの高い検索条件になる。SIMILAR TOよりもLIKEのほうがコストが低いため、これをうまく利用するのがコツだ。正規表現は複雑で、毎回計算しなければならない（なお、正規表現を最適化する方法については、本書では扱わない）。

LIKE 述語のパターンマッチングでは、極力「%」を避けて「_」を使うことだ。たとえば、CHAR(5)の列を持つテーブルを考えよう。

```
SELECT student_first_name, student_last_name, homeroom_nbr
  FROM Students
 WHERE homeroom_nbr LIKE 'A-1Q_';
```

このクエリでインデックスが使われるかどうかは断言できないが、もし最後の1文字が数値であることが判明している場合、次のように書き換えることができる。

```
SELECT student_first_name, student_last_name, homeroom_nbr
  FROM Students
 WHERE homeroom_nbr BETWEEN 'A-1Q0' AND 'A-1Q9';
```

このクエリはhomeroom_nbr列に対するインデックスを使用できるだろう。注意が必要なのは、このトリックが使えるのは、VARCHAR(5)ではなくCHAR(5)の場合に限られることだ。もしVARCHAR(5)だと、2 番目のクエリは'A-1Q'という行を選択してしまうが、もとのLIKE 述語は選択しないので、同値性が崩れてしまう。文字列の同値性とBETWEEN述語は、短いほうの文字列の右側に空白を埋めてから比較する。LIKE述語はこのような動作はしない。

39.2.3　シンプルな時間式

複雑な文字列や数値の式を避けるべきなのと同じ理由で、時間の計算も極力避けるべきである。西暦カレンダーとバビロニアの60進時間モデルの変則的な性質が、データベース内部での日付計算を極めて複雑なものにしているからだ。

たとえば、仕事の開始日と終了日を持つテーブルがあるとしよう。ここから、終わらせるのに3日かかる仕事を見つける。

```
SELECT task_nbr
  FROM Tasks
 WHERE (finish_date - start_date) = INTERVAL '3' DAY
   AND start_date >= CAST('2015-01-01' AS DATE);
```

だが、多くのレポートで使うのは終了日であるため、この列にインデックスがあるとしよう。すると上のクエリは、次のように書いたほうが速いということだ。

```
SELECT task_nbr
  FROM Tasks
 WHERE finish_date = (start_date + INTERVAL '3' DAY)
   AND start_date >= CAST('2015-01-01' AS DATE);
```

39.3

その他の注意事項

人間ならわかることがオプティマイザにはわからないことがある。クエリに多くの情報があるほど、オプティマイザはより良い実行計画を立てられる。最も有用な情報はDDLの制約である。オプティマイザは自ら制約を見に行って実行計画を立てるときの参考にする。

だが、制約がない場合、ユーザーはコード中でそれを明示しなければならない。たとえば、次のようなクエリがあるとする。

```
SELECT sale_nbr, alpha_code
  FROM Sales
 WHERE sale_amt >= 500.00;
```

もしDDLで`alpha_code`列が次のような制約を持っていたとしよう。

```
CREATE TABLE Sales
(...
 alpha_code CHAR(3) NOT NULL
     CHECK (alpha_code SIMILAR TO 'A[:UPPER:]C'),
 ...);
```

すると、上のクエリは実質的にこう書いたのと同じだ。

```
SELECT sale_nbr, alpha_code
  FROM Sales
 WHERE sale_amt >= 500.00;
   AND alpha_code SIMILAR TO 'A[:UPPER:]C';
```

このことと、論理的に冗長な条件とを混同しないでほしい。冗長な条件は、オプティマイザが本当にどちらも計算してしまって二度

手間になる可能性がある。たとえば、次のようなクエリは悪い例だ。

```
SELECT sale_nbr, alpha_code
  FROM Sales
 WHERE sale_amt >= 500.00
   AND alpha_code SIMILAR TO 'A[:UPPER:]C'
   AND alpha_code BETWEEN 'AAA' AND 'ZZZ';
```

これでは、BETWEEN述語が各行に対して計算されかねない。しかも、SIMILAR TO述語の条件がTRUEならばBETWEEN述語の条件もTRUEになるので、まったく絞り込みの役に立っていない。

この制約によってオプティマイザに与えられる情報を、ヒントなどの実行計画を制御する機能とも混同しないでほしい。ヒントはオプティマイザに特定の実行計画を選ばせるよう指示する機能だが、構文やできることの範囲は実装によって異なる【訳注5】。ヒントの問題点は、それが永久にコードの中に残ることだ。統計情報、テーブル構造、インデックスなどが変化したとしても、ヒントは変わらない。誰もヒントを取り除く勇気のある者はいない。なぜそのヒントが追加されたか覚えている者はいないからだ。プログラマの世界には「一時的」に作られたパッチが最も長生きするという格言があるが、それはSQLにも当てはまる。

39.4

複数列に対するインデックスは慎重に

物理的制約から明らかなことだが、テーブルにはシーケンシャルインデックスは1つしか作れない。したがって、どの列をインデックスに含むかはパフォーマンス上重要な判断になる。

主キーはシーケンシャルインデックスでなくてはならない、というのは、ISAMファイルと階層型データベースやネットワークデータベースで仕事をした経験のある年輩のDBAにありがちな迷信だ。この迷信が生じた理由は、昔のファイルシステムでは、ファイルはキーでソートしなければならなかったからだ。すべての検索はそのキーで行われた。だがこれはSQLには当てはまらない。多くの主キーの一意性はユニークインデックスで担保されるが、それは別にシーケンシャルインデックスである必要はない。たとえば、社員を一意な社員IDで管理するテーブルがあるとする。社員情報の更新は社員IDをキーに行われるが、一方で社員ID順に表示するレポートは少ないだろう。そう考えると、社員情報の更新は、インデックスがシーケンシャルだろうとそうでなかろうと、いずれにせよ非常に高速である。どちらでも木の構造は同じで、リーフが指す物理的アドレスが違うだけだからだ。だが、レポートにおける表示の単位は、社員個人というより部署のほうが会社にとっては意味を持つだろう。そうすると、社員IDにシーケンシャルインデックスが存在し、行が社員ID順にソートされていたとしても、部署単位の集約を行ううえではたいした利点はない。むしろ、社員IDより、社員の姓でソートされていたほうが、まだレポートには有用だ。

逆に、シーケンシャルインデックスが（一意でない）部署コードの列に存在していたとすれば、同じ部署に所属する社員は、同じ物理的な領域に格納される可能性が高くなり、検索に必要な読み出し量が減ることが期待できる。

最近の実装では、ANDやORで連結された条件に対しては、複数のインデックスを利用することができる。また、もしインデックスがクエリで必要とされるすべての列をカバーしていれば、基底テーブルへのスキャンを行う必要がなく、インデックスのみで検索を完結させる機能（インデックスオンリースキャン）を持っている実装も多い。

39.5

IN 述語には要注意

IN 述語は、OR で連結された同値性テストの短縮形である。IN 述語には2 つの使い方がある。1つが、明示的な値のリストを与える方法。もう1つが、そのようなリストを作るサブクエリを与える方法である。

データベースエンジンは、定数のリストに含まれる値の相対頻度については何の統計も持っていないので、左から現れた順に使っていく。我々人間は、リストをアルファベットや大きさの順に並べるのを好むが、パフォーマンスを考えれば、出現頻度の高い順に記述するほうがよい。なお、リストの中に重複値を記述するのは意味がないので避けてほしい。IN 述語は最初にマッチした値を見つけたらTRUEを返して評価を打ち切る。この場合、重複値の2個目が参照されることは永遠にない。かつ、もし重複値の最初がFALSEに評価されたら、2個目もFALSEに決まっているので、CPUを浪費しても仕方がない。

IN述語にサブクエリが与えられると、多くの実装では、最初にサブクエリを実行して、定数のリストを持つ一時テーブルに変換してからIN 述語の評価が行われる。これはコストが高くなりやすい。たとえば、次のクエリを見よう。

```
SELECT P1.first_name, P1.last_name
  FROM Personnel AS P1, BowlingTeam AS B1
 WHERE P1.last_name
         IN (SELECT last_name
               FROM BowlingTeam AS B1
              WHERE P1.emp_nbr = B1.emp_nbr)
                AND P1.first_name
                      IN (SELECT first_name
                            FROM BowlingTeam AS B2
                           WHERE P1.emp_nbr = B2.emp_nbr);
```

このクエリは、BowlingTeam テーブルに対してスキャンを2 回行う必要がある。これよりも、次のクエリのほうが速い。

```
SELECT P1.first_name, P1.last_name
  FROM Personnel AS P1
 WHERE first_name || last_name
         IN (SELECT first_name || last_name
               FROM BowlingTeam AS B1
              WHERE P1.emp_nbr = B1.emp_nbr);
```

これならば、BowlingTeamテーブルへのスキャンは一度で済む。さらに速くするならば、サブクエリを非相関にしよう（ただし、この変換が可能なのは姓名で人物を一意に特定できる場合だ）。

```
SELECT P1.first_name, P1.last_name
  FROM Personnel AS P1
 WHERE first_name || last_name
         IN (SELECT first_name || last_name
               FROM BowlingTeam);
```

標準SQLの行式を使って書けば、次のようになる。

```
SELECT P1.first_name, P1.last_name
  FROM Personnel AS P1
 WHERE (first_name, last_name)
          IN (SELECT first_name, last_name
                FROM BowlingTeam);
```

これはおそらく最速だ。なぜなら、「first_name || last_name」のように文字列式の中に列を入れてしまうと、この列に対するインデックスは使われないが、(first_name, last_name)ならば、オプティマイザはインデックスを使えると判断するかもしれないからだ。

このように、同じ結果を得られるクエリの形式が複数ある場合、オプティマイザは、内部的に同値変換を行うこともある。たとえば、次の2 つのクエリは同値であるため、オプティマイザは内部的に変換可能とみなすだろう。

```
SELECT *
  FROM Warehouse AS W1
 WHERE onhand_qty IN (SELECT order_qty FROM Sales);

SELECT *
  FROM Warehouse
 WHERE onhand_qty = ANY (SELECT order_qty FROM Sales);
```

同じ結果を返すクエリであっても、どのような演算子を使うかは大きなパフォーマンス上の違いをもたらすことがある。この例だと、IN述語は次のようにさらに結合へと同値変換できる。

```
SELECT W1.*
  FROM Warehouse AS W1, Sales AS S1
 WHERE W1.onhand_qty = S1.order_qty;
```

これは、結合キーのインデックスが使えれば、一時テーブルを作る必要がないため、3つの中では最も速いだろう【訳注6】。

39.6

UNIONを避ける

UNIONは、内部的には2 つの結果セットをマージして、重複行を削除するためにソートする、という実装がされていることが多い。複数のテーブルをスキャンするのはコストが高いので、UNIONを避けられないか、常に検討するべきだ。たとえば、次のクエリを考える。

```
SELECT *
  FROM Personnel
 WHERE work_city_name = 'New York'
UNION
SELECT *
  FROM Personnel
 WHERE home_city_name = 'Chicago';
```

これは次と同値であるが、このほうが速い。

```
SELECT DISTINCT *
  FROM Personnel
 WHERE work_city_name = 'New York'
    OR home_city_name = 'Chicago';
```

もう1つのトリックは、重複行が発生しないことがわかっている、または重複行が結果に含まれてもかまわない場合は、UNION の代わりにUNION ALLを使うことだ。UNION ALL はソートを行わないため、そのぶんUNIONより高速だ。もっとも、テーブルスキャンを複数回繰り返すコストに比べればその程度のコストは雀の涙、ということも多いが。

39.7

サブクエリを避けて結合を使う

サブクエリは最適化が難しい。そこでオプティマイザはこれを「フラット化」して結合に変換しようとする。次のようなサンプルを考えよう。

```
CREATE TABLE Authors
(author_nbr INTEGER NOT NULL PRIMARY KEY,
 author_name CHAR(50) NOT NULL);
```

Authors：著者テーブル　author_nbr：著者番号　author_name：著者名

```
CREATE TABLE Titles
(isbn CHAR(10)NOT NULL PRIMARY KEY,
 book_title CHAR(50) NOT NULL,
 advance_amt DECIMAL(8,2) NOT NULL);
```

Titles：書籍テーブル　isbn：ISBN book_title：書名　advance_amt：支払額

```
CREATE TABLE Authorship
(author_nbr INTEGER NOT NULL REFERENCES
     Authors(author_nbr),
 isbn CHAR(10)NOT NULL REFERENCES Titles(isbn),
 royalty_rate DECIMAL(5,4) NOT NULL,
     PRIMARY KEY (author_nbr, isbn));
```

Authorship：著者書籍テーブル　author_nbr：著者番号　isbn：ISBN book_title：書名　royalty_rate：印税

次のクエリは、印税が50%未満の著者を見つけるものだ。

```
SELECT author_nbr
  FROM Authors
 WHERE author_nbr
         IN (SELECT author_nbr
               FROM Authorship
              WHERE royalty_rate < 0.50);
```

同じことは次のクエリでもできる。

```
SELECT DISTINCT Authors.author_nbr
  FROM Authors, Authorship
 WHERE Authors.author_nbr = Authorship.author_nbr
   AND royalty_rate < 0.50;
```

SELECT DISTINCT は、複数の本を書いている著者がいた場合に重複を除外するためだ。IN 述語の場合は、著者名が複数返されたとしても、INの評価時に重複は除外されるが（39.5節参照）、結合の場合は結果に重複が残るために必要となる。

39.8

SQL 文の数を減らせ

SQLとデータベースを扱うときの鉄則の1 つが、発行するSQL 文の数を極力減らすことだ。SQL文の数が少なければ少ないほど、アクセスが必要なデータ量は少なくなる【訳注7】。SQLの数が少ないということは、保守するコードも少ないということだ。

このようなSQL 削減の代表例は、MERGE 文である。読者には、同じテーブルへのINSERTとUPDATEが含まれているストアドプロシージャがないか、コードをよくチェックしてほしい。それはきっと1つのMERGE文で実行可能なものだ。

ソートを避ける

SELECT DISTINCT、GROUP BY句、ORDER BY句は多くの製品で内部的にソートを発生させる。したがって、本当に必要なとき以外は使うべきでない。もし重複行を排除したり、結果セットが一意であることを保証したい場合だけ使うべきだ。

UNION、INTERSECT、EXCEPTもまた重複行を排除するためにソートを行う。例外は、ソートをスキップできるようなインデックスが存在する場合だけである。重複行を排除しなくてよい場合は、ALLオプションを使うべきだ。

GROUP BY句がない場合でも、集約関数やHAVING句が使われていればやはりソートが必要だ。ウィンドウ関数も同様にソートを発生させる。

ソートにかかるコストはアルゴリズムにもよるが、おそらく(n*log2(n))に比例する。

SELECT DISTINCTは、しばしばEXISTS述語で変換できる。これは「サブクエリを避ける」という原則に反するように見えるかもしれないが、EXISTS述語は例外だ。たとえば、科学専攻の学生を見つけるクエリは、普通に考えれば次のようになる。

```
SELECT DISTINCT S1.name
  FROM Students AS S1, ScienceDepts AS D1
 WHERE S1.dept = D1.dept;
```

これは次のように書いたほうが速い可能性がある。

```
SELECT S1.name
  FROM Students AS S1
 WHERE EXISTS
       (SELECT *
          FROM ScienceDepts AS D1
         WHERE S1.dept = D1.dept);
```

もう1 つの問題は、テーブルにおいて最低限必要な一意性制約だけ宣言されていて、他の一意性制約が存在しない場合だ。次のような授業のスケジュールを管理するテーブルを考える。

```
CREATE TABLE Schedule
(room_nbr INTEGER NOT NULL,
 course_name CHAR(7) NOT NULL,
 teacher_name CHAR(20) NOT NULL,
 period_nbr INTEGER NOT NULL,
     PRIMARY KEY (room_nbr, period_nbr));
```

Schedule：スケジュールテーブル　room_nbr：教室番号　course_name：コース名　teacher_name：教師名　period_nbr：学期番号

この主キーから、もし教室と学期がわかれば、教室とコースが一意に定まる。たとえば「第3 学期の教室101を使うのは、ジョーンズ先生の英語初級」というように。だが、このテーブルにおいて一意なキーはこれだけではない。たとえば、`UNIQUE (teacher, period)`という条件も付与することが可能だ。というのも、ジョーンズ先生は一学期に1人しか存在しないからだ。こうした冗長な一意性が宣言されていないと、オプティマイザはそこから得られる情報をクエリのパース時に使うことができない。同様に、もし主キーが`PRIMARY KEY (room_nbr, course_name, teacher_name, period_nbr)`とだけ宣言されていたら、やはりオプティマイザはこのキーを個々の候補キーに分解することはできない。

SELECT 句とWHERE 句ですべてのことが可能ならば、あえてHAVING 句やGROUP BY 句を使うべきではない。たとえば、次の2つのクエリは同値だが、後者のほうが速い。

```
SELECT project_type, AVG(cost)
  FROM Tasks
 GROUP BY project_type
HAVING project_type = 'bricklaying';

SELECT 'bricklaying', AVG(cost)
  FROM Tasks
 WHERE project_type = 'bricklaying';
```

HAVING 句のクエリは、まずテーブルをフルスキャンして、全行を対象にソートを行いながらHAVING句によるフィルタを行う。2番目のクエリは、不要な行を最初に除外するので、ソートに渡される行数が少なくなる。

なお、GROUP BY句を避ける方法として、SELECT句でスカラサブクエリを使うという方法があるが、これはあまり速くはない。たとえば、次のような在庫管理のテーブルを考えよう。

```
CREATE TABLE Requisitions
(req_date DATE NOT NULL,
 req_qty INTEGER NOT NULL
     CONSTRAINT non_zero_qty
       CHECK (req_qty <> 0));
```

Requisitions：在庫管理テーブル　req_date：入出庫日　req_qty：入出庫量

いま、各日付における在庫数を返すクエリを考える。欲しい結果のイメージは次のようなものだ。

req_date	req_qty	onhand_qty

2013-07-01	100	100
2013-07-02	120	220
2013-07-03	-150	70
2013-07-04	50	120
2013-07-05	-35	85

古典的な方法ではこうなる。

```
SELECT R1.req_date, R1.req_qty, SUM(R2.req_qty) AS onhand_qty
  FROM Requisitions AS R1, Requisitions AS R2
 WHERE R2.req_date <= R1.req_date
 GROUP BY R1.req_date, R1.req_qty;
```

これと同じ結果を相関スカラサブクエリを使って書くことができる。

```
SELECT R1.req_date, R1.req_qty,
       (SELECT SUM(R2.req_qty)
          FROM Requisitions AS R2
         WHERE R2.req_date <= R1.req_date) AS onhand_qty
  FROM Requisitions AS R1;
```

だがこれはGROUP BYは削除できたものの、結局サブクエリ内で集約関数を使っており、ソート自体は残っている。しかも相関スカラサブクエリは高コストだ。結局、この問題を解く最も良い方法は、ウィンドウ関数を使うことだろう。これも内部的にソートを使うが、結合がないためパフォーマンス的に有利だ。

```
SELECT req_date, req_qty,
       SUM(req_qty) OVER(ORDER BY req_date)
  FROM Requisitions;
```

39.10

クロス結合を避ける

次のような3つのテーブルを結合するクエリを考えよう。

```
SELECT P1.paint_color
  FROM Paints AS P1, Warehouse AS W1, Sales AS S1
 WHERE W1.onhand_qty + S1.sold_qty = P1.gallons/2.5;
```

結合条件に含まれる列がすべて式の中にあるため、インデックスがあったとしても一切使用できない。こういう場合、SQL エンジンはクロス結合を選択し、その後で結合条件によるフィルタを行うことがある【訳注8】。このような問題を避けるためにも、次のように少なくとも1つの列は式に含めない「裸」の状態にしておくべきだ。

```
SELECT P1.paint_color
  FROM Paints AS P1, Warehouse AS W1, Sales AS S1
 WHERE W1.onhand_qty = P1.gallons/2.5 - S1.sold_qty;
```

汝、自らのオプティマイザを知れ

自分の使う製品のオプティマイザの癖を知っておくことは重要だ。チューニングのためにオプティマイザの癖に合わせた特殊な書き方が必要とされることも、残念ながらたびたびある。たとえば、次のような隣接リストモデルを扱うクエリを考えよう。

```
CREATE TABLE Tree
(node_id CHAR(2) NOT NULL,
 parent_node_id CHAR(2), -- NULLはルートを意味する
 creation_date DATE DEFAULT CURRENT_TIMESTAMP NOT NULL);
```

Tree：木テーブル　node_id：ノードID　parent_node_id：親ノードID
creation_date：作成日

親ノードとそのノードに従属する子ノードのうち、最も直近に作られたノードを選択するクエリは、普通に考えれば次のようになる。

```
SELECT node_id, T1.parent_node_id, T1.creation_date
  FROM Tree AS T1
 WHERE NOT EXISTS
        (SELECT *
           FROM Tree AS T2
          WHERE T2.parent_node_id = T1.parent_node_id
            AND T2.creation_date > T1.creation_date);
```

EXISTS 述語のサブクエリは、当該ノードよりもあとに作られた子ノードは存在しない、という意味である。これは、外部結合を使って次のようにも書ける。

```
SELECT T1.node_id, T1.parent_node_id, T1.creation_date
  FROM Tree AS T1
         LEFT OUTER JOIN
           Tree AS T2
    ON T2.parent_node_id = T1.parent_node_id
   AND T2.creation_date > T1.creation_date
 WHERE T2.node_id IS NULL;
```

後者のクエリはSQL Server 2000では、内部的にいったん集約したビューを作るように変換されていた。つまり、次のようなクエリになる。これは、他の方法よりもはるかに速かった。

```
SELECT T1.node_id, T1.parent_node_id, T1.creation_date
  FROM Tree AS T1
         INNER JOIN
           (SELECT parent_node_id, MAX(creation_date)
              FROM Tree
             GROUP BY parent_node_id)
              AS T2 (parent_node_id, creation_date)
    ON T2.parent_node_id = T1.parent_node_id
   AND T2.creation_date = T1.creation_date;
```

オプティマイザの仕様は同じ製品においてもリリースのたびに変更されるが、こうした最適化のトリックは、いったんコードに取り入れられたら、その後は大きな改修がない限り製品の機能としてとどまるはずだ。フェビアン・パスカルは『Database Programming and Design』誌（vol. 1, #12, December 1998）に書いた記事「SQL Redundancy and DBMS Performance」の中で、いくつかの論理的に同値なクエリが、主要な製品の間で大きく性能が異なると述べた。これはつまり、オプティマイザは年々進化していくとはいえ、長期的に持続する特有の癖と、特定のタイプのクエリの好みを持っているということである。

39.12

スキーマが変更されたら静的SQLをリコンパイルする

多くの実装では、静的SQLはホストプログラムにおいてコンパイルされ、固定的な実行計画が割り当てられる。データベースオブジェクトが変更されると、それに基づいていた実行計画も変わらなくてはならない。

古い実装では、スキーマオブジェクトが削除されると、プログラマはそれを参照しているクエリをリコンパイルする必要があった。SQL エンジンは何らチェックすることを要求されていなかったので、大半の実装ではチェックされなかった。代わりに、ユーザーにはエラーが返された。最悪の場合は、エラーすら返らなかった。

ステップ1　テーブルAを作る
ステップ2　テーブルAに基づくビューVAを作る
ステップ3　ビューVAを使う
ステップ4　テーブルAを削除
ステップ5　新しいテーブルAを作る
ステップ6　ビューVAを使う

「ステップ6」で何が起きるだろうか？ 答えは実装依存だが、良い結果にはならない。「ステップ3」で作られたVAは「ステップ6」で作られたVAと同値ではないが利用できるというのが一番マシな結果だろう。多くの場合はエラーが発生する。

標準SQL では、テーブルを削除するDROP 文にCASCADE か RESTRICT のオプションを付けられる。デフォルトはRESTRICT

で、他のオブジェクトから参照されているオブジェクトを削除することを許さない。たとえば、ビューや参照整合性制約から参照されているテーブルを削除することはできない。CASCADEは依存するオブジェクトも合わせて削除する。

注意が必要なのは、X/Openトランザクションモデルでは、CASCADE がデフォルトであり、製品によっては、これをシステムパラメータで切り換えられることだ【訳注9】。安全側に倒すならば、スキーマを破壊してしまうような暗黙のデフォルト設定は使わず、必要なときにオプションを明示的に指定することだ。

インデックスが削除されると自動的に静的SQLをリコンパイルする製品もある。だが、インデックスが追加されたときに静的SQLをリコンパイルする製品は少ない。さらに、統計情報が更新されたときにリコンパイルする製品もほとんどない。普通DBAは、統計情報を明示的に更新する場合は、リコンパイルも要求しなければならない。開発者は、自分が必要とするインデックスが作成されたら自分のプログラムをリコンパイルするだけだ。その新しいインデックスがシステム全体のパフォーマンスにとって吉と出るか凶と出るかは容易に判断できない。

異なる開発者が同じ列の組み合わせで同じ順番のインデックスを異なる名前で作成することは、避けなければならない。実装の中には、このようなインデックスを検知しないものもある。このような重複は時間と領域の無駄だ。変更するときも2 つを忘れずに更新しなければならない。こうした重複を防ぐためにも、スキーマオブジェクトはDBAのもとで一元管理する必要がある。

39.13

一時テーブルも時には役に立つ

性能という観点から見ると、一時テーブルはアンチパターンの1 つだ。一時テーブルは実体化する必要がある。ということは、もし中間結果をメモリに持つことができるなら、一時テーブルを使うのは無駄だし、中間結果が大きくてメモリには持てないならば、オプティマイザは自分で判断してディスクなどに書き出す。いずれにせよわざわざ一時テーブルを作るのは余計なことだ【訳注10】。

だが、一時テーブルが役に立つケースもある。たとえば何度も同じ計算を繰り返したり、クロス結合を避けるために使うケースだ。ANSI/ISOの一時テーブルでは、基底テーブルが持っているすべてのオプションを持っている。違うのは、データが自動的に消えることだけだ。実体化されたビューもまた一時テーブルの一種だが、データが自動的に消えることはない。たとえば、銀行口座の最近の収支を計算する問題を考えよう。

`Payments`テーブルには、毎回の支払い明細が記録されているとする。支払額の合計は時間が経つほど大きくなる。`Accounts`テーブルには、口座番号と、毎回の支払いが終わったあとの残額が記録されている。

クエリは次のようなものになるだろう。

```
SELECT SUM(A1.acct_balance)
  FROM Accounts AS A1, Payments AS P1
 WHERE P1.acct_nbr = A1.acct_nbr
   AND P1.payment_nbr
         = (SELECT MAX(payment_nbr)
              FROM Payments AS P2
             WHERE P2.acct_nbr = A1.acct_nbr);
```

これは相関サブクエリと集約関数を使っているため、速いとは言いがたい。一時テーブルや実体化されたビューを作ったほうが速いだろう。

```
CREATE TABLE LastPayments
(act_nbr INTEGER NOT NULL,
 last_payment_nbr INTEGER NOT NULL,
 payment_amt DECIMAL(8,2) NOT NULL,
 payment_date DATE NOT NULL,
 ... );

CREATE INDEX LPX ON LastPayment(act_nbr, payment_nbr);

INSERT INTO LastPayments
SELECT act_nbr, payment_nbr, MAX(payment_nbr)
  FROM Payments
 GROUP BY act_nbr, payment_nbr;

SELECT SUM(A1.balance) -- 最終的な答え
  FROM Accounts AS A1, LastPayments AS LP1
 WHERE LP1.acct_nbr = A1.acct_nbr
   AND LP1.payment_nbr = A1.payment_nbr;

DROP TABLE LastPayments;
```

次のような3つのテーブルを使うクエリを考えよう。これは、商品とパッケージの組み合わせについて価格の10%を保証料に取る保証プランを出力する。すべての製品がすべてのパッケージに入れることができ、どのような保証プランも適用できると仮定する。

```
SELECT I1.item
  FROM Inventory AS I1, Packages AS P1, Warranty AS W1
 WHERE I1.price + P1.box = W1.plancost * 10;
```

この検索条件では、すべての列が式の中に現れるため、インデックスを使うことができない。下手をするとクロス結合が実行される

かもしれない。これは次のようにインデックスを持つ一時テーブルに分解することができる。

```
CREATE TABLE SellingPrices
(item_name CHAR (15) NOT NULL,
 sell_price DECIMAL (8,2) NOT NULL);

-- インデックスを付与する
CREATE INDEX SPX ON SellingPrices (sell_price);

-- 保証プランの計算を行う
INSERT INTO SellingPrices (item_name, sell_price)
SELECT DISTINCT I1.item_name, (P1.box + I1.price) * 0.1
  FROM Inventory AS I1, packages AS P1;

-- 最後に結合を実行する
SELECT DISTINCT SP1.item_name
  FROM SellingPrices AS SP1, Warranty AS W1
 WHERE SP1.sell_price = W1.plancost;
```

Sybase/SQL Serverファミリーでは、プログラマが使い捨ての一時テーブルを作ることができる。これは標準SQLにおける一時テーブルとはかけ離れたものだ。標準SQL では、USER 権限のユーザーにはスキーマオブジェクトを作る権限は与えられていない。それはADMIN 権限を持ったユーザーの仕事だ。

このような「アドホック」なオブジェクトは、往々にしてインデックスや制約を持つことができず、オプティマイザはこれを扱うのに苦労する。これは、我々が手続き型の世界で中間結果セットをいちいちファイルに書き出していた時代の遺物だ。

いまではこのようなアドホックなオブジェクトの代わりに、クエリの中で導出テーブルを使うことが多い。導出テーブルの定義はパーサによって解析され、制約やインデックスの情報を考慮して実行計画の中に取り込まれる。

39.14

統計情報を更新する

当たり前のことに聞こえるかもしれないが、オプティマイザは妥当な統計情報がなければ正しく動作しない。以下の前兆が見えたら統計を更新するべきだ。

前兆1 同じ構造を持つ2つのクエリに異なる実行計画が割り当てられる。これは普通、データの分散が変化したにもかかわらず、片方のSQL が古い情報をもとに動いていることを意味する。

前兆2 大量データのロードが行われた。製品によっては、テーブルのデータ量が大きく変動すると自動的に統計情報が更新される。

前兆3 大量のデータが削除された。残念ながら、ロードと違って削除のタイミングで統計情報の更新が自動的に行われる製品は少ない。

前兆4 クエリの混合率が変化した。たとえば、月末になるとレポート作成のために集約データを保持するビューを使った重いクエリが集中するシステムは多い。

新機能は信用しない

新機能は格好良く見えるものだ。だが、それらはまだ十分に最適化されていない危険がある。

たとえば、SQLにウィンドウ関数が実装されたとき、それは自己結合を代替するうまい手段のように思われた。だが、初期の実装では無駄なソートが行われることで、単純な自己結合よりも性能が悪いことがしばしばあった。簡単な例を考えよう。一般的な定義の社員テーブルがあるとしよう。ここから、各部門で一番高給取りの社員を見つける。

```
CREATE TABLE Personnel
(emp_name VARCHAR(20) NOT NULL PRIMARY KEY,
 salary_amt DECIMAL (8,2) NOT NULL,
 dept_name VARCHAR(20) NOT NULL)
```

解1

「伝統的な」やり方は、自己結合を使うものだ。つまり、部署の中で最高の給与を求め、それと一致する給与の社員を選択する。これが多くの人が考えるコードだ。

```
SELECT P1.emp_name, P1.salary_amt, P1.dept_name
  FROM Personnel AS P1
 WHERE P1.salary_amt
        = (SELECT MAX(salary_amt)
             FROM Personnel AS P2
            WHERE P2.dept_name = P1.dept_name)
```

解2

あるいは、相関サブクエリの代わりに`NOT EXISTS`を使う方法もある。これは、その社員より部署内で給与の高い社員が存在しない

社員を求める。

```
SELECT P1.emp_name, P1.salary_amt, P1.dept_name
  FROM Personnel AS P1
 WHERE NOT EXISTS
       (SELECT *
          FROM Personnel AS P2
         WHERE P2.salary_amt > P1.salary_amt
           AND P2.dept_name = P1.dept_name);
```

もし部署名の列にインデックスがあれば、このクエリはかなり速い。EXISTSはインデックスを利用するので、サブクエリのように一時的なテーブルを作らないからだ。だが、このクエリは見た目が複雑だ。新機能を使えばもっと簡潔に書ける。

解3

共通表式とウィンドウ関数を使う。

```
WITH P_Max (emp_name, salary_amt, dept_name, salary_amt_max)
AS
(SELECT emp_name, salary_amt, dept_name,
        MAX(salary_amt) OVER (PARTITION BY dept_name)
   FROM Personnel)
SELECT emp_name, salary_amt, dept_name
  FROM P_Max
 WHERE salary_amt = salary_amt_max;
```

部署ごとの最高の給与を求め、それと一致する社員をメインのクエリでフィルタしている。賢いオプティマイザなら、パーティションごとの最大値を求める際にGROUP BYと同じ動作をするだろう。その点で、実はこの解は最初の解と考え方が似ている。

解4

これはつまり、別にウィンドウ関数を使わなくてもGROUP BYでいいということだ。これは「解3」と同じ実行計画になるはずだ。読者

の実装で確認してほしい。

```
WITH P2 (dept_name, salary_amt_max)
AS
(SELECT dept_name, MAX(salary_amt)
   FROM Personnel
  GROUP BY dept_name)
SELECT P1.emp_name, P1.salary_amt, P1.dept_name
  FROM Personnel AS P1, P2
 WHERE P1.dept_name = P2.dept_name
   AND P1.salary_amt = P2.salary_amt_max;
```

解5

ということは、ウィンドウ関数は無駄なのだろうか？ もちろんそうではないが、魔法でもない。たとえば、次の解はオプティマイザが賢ければ他の解よりもパフォーマンスが良いはずだ。順序関数で割り振った連番をWHERE句の検索条件にするという方法はポピュラーなので、最適化の恩恵を受けられるだろう。

```
WITH P1 (emp_name, dept_name, salary_amt, rk)
AS
(SELECT emp_name, dept_name, salary_amt,
        DENSE_RANK()
          OVER(PARTITION BY dept_name
                   ORDER BY salary_amt DESC)
   FROM Personnel)
SELECT emp_name, dept_name, salary_amt
  FROM P1
 WHERE rk = 1;
```

【訳注1】 データベースの性能問題のほとんどすべてはI/Oがボトルネックになって引き起こされる。したがって、データベースにおけるチューニングは、つまるところ「いかにしてI/Oコストを減らすか」に腐心することにな

る。本章を読む際はこの点を大前提として理解しておいてほしい。

【訳注2】 SSDによるI/O革命は2013年の時点ですでに現実のものになっていることは、読者も承知のとおりである。

【訳注3】 関数インデックスは、Oracle、PostgreSQL、SQL Serverなどがサポートしている。

【訳注4】 NULLに対するインデックスは実装によって扱いが異なるので注意が必要だ。DB2ではIS NULLに対してもインデックスを使った検索が行われる。一方、Oracleではインデックスは使われない。

【訳注5】 ヒントの実装状況については、DB2のように原則持たないものから、Oracleのように何でもできるものまで千差万別である。

【訳注6】 このクエリは結果にWarehouseテーブルの列しか含まないため、さらにEXISTS述語で書き換えることもできる。これもよく利用するチューニング手段である。

【訳注7】 この文の含意は明らかだと思うが、念を押しておくと、複数のSELECT文をUNIONでマージするようなやり方でSQLの発行数を減らしても意味がない。UNIONは内部的に複数のSELECT文を実行しているので、アクセスするデータ量を減らす効果はない。

【訳注8】 クロス結合が生じやすいケースとしては、いわゆる「三角結合」もある。これは、A、B、Cという3つのテーブルを結合するとして、AとB、BとCの間には結合条件が

あるが、AとCの間には結合条件がないパターンだ。

```
SELECT A.a, B.b1, C.c
  FROM A, B, C
WHERE A.a = B.b1
  AND B.b2 = C.c;
```

この場合、素直にまずAとBを結合し、その結果とCを結合するという順番で結合すればクロス結合は発生しない。だが、いきなりAとCを結合すると、クロス結合が発生する。オプティマイザがなぜクロス結合を選択するのかはブラックボックスだが、AとCの件数が少なく、クロス結合してもリスクが少ないと判断するケースにおいて発生するのだろう。これを避けるには、冗長ではあってもAとCの間に（もし存在するならば）結合条件を記述すればよい。ヒントで結合順序を指定するという方法は、本書のすすめるものではない。

【訳注9】　X/Openは、ヨーロッパのUNIXベンダーが中心となって設立された標準化コンソーシアム。

【訳注10】　一時テーブルのパフォーマンス上の最大の難点は、統計情報の取得タイミングの判断が難しいことだ。しばしば空っぽの状態で取得されたまま、大量データがロードされてパフォーマンス問題を引き起こす。

参考文献

一般

Adams, D. (2004). *The Hitchhiker's guide to the galaxy, 25th anniversary edition*. Crown.

論理学

Bolc, L., & Borowik, P. (1992). *Many-valued logics*. New York: Springer-Verlag.［本書は3値論理に一章を割いている。］

Boole, G. (1854/1951). *An investigation of the laws of thought, on which are founded the mathematical theories of logic and probabilities*. New York:Macmillan and Co./New York:Dover Books.［初版はコレクター垂涎の貴重なアイテムだ。通常われわれが読むことができるのは、1854年の再版である。］

Celko, J. (1992). SQL explorer: Voting systems. *DBMS Magazine*, 11.

数学

Gardner, M. (1983). Wheels, life, and other mathematical amusements. New York: W. H. Freeman.（『アリストテレスの輪と確率の錯覚：ガードナーの数学ゲーム・コレクション』マーチン・ガードナー著，一松信訳，日経サイエンス社，1993）

Huff, D., & Geis, I. (illustrator). (1993, reissue). How to lie with statistics. New York: Norton.（『統計でウソをつく法：数式を使わない統計学入門』ダレル・ハフ著，高木秀玄訳，講談社，1968）

Knuth, D. (1997a). *The art of computer programming, volume 1: Fundamental algorithms* (3rd ed.). Reading, MA: Addison-Wesley.（『The Art of Computer Programming Volume 1 Fundamental Algorithms Third Edition 日本語版』Donald E. Knuth 著，有澤誠／和田英一監訳，アスキー，2004）

Knuth, D. (1997b). *The art of computer programming, volume 2: Semi-numeral algorithms* (3rd ed.). Reading, MA: Addison-Wesley.（『The Art of Computer Programming Volume 2 Seminumerical Algorithms 日本語版』Donald E. Knuth 著，有澤誠／和田英一監訳，アスキー，2004）

Knuth, D. (1998). *The art of computer programming, volume 3*: Sorting and searching (3rd ed.). Reading, MA: Addison-Wesley.（『The Art of Computer Programming Volume 3 Sort

ing and Searching Second Edition 日本語版』Donald E. Knuth 著，有澤誠／和田英一監訳，アスキー，2006）

Knuth, D. (2009). *The art of computer programming, volume 4, fascicles 0-4* (5 volume set).（『The Art of Computer Programming Volume 4, Fascicle 0–4 日本語版』（5 分冊）Donald E. Knuth著，和田英一訳，アスキー・メディアワークス，2006–2010）

Melzak, Z. A. (1983). *Bypasses: A simple approach to complexity*. New York: John Wiley & Sons.

Mrdalj, S., Vujovic, B., & Jovanovic, V. (1996, August). SQL matrix processing. *Database Programming and Design*.

乱数

Bays, C., & Sharp, W. E. (1992). Improved random numbers for your personal computer or workstation. Geobyte, 7(2), 25.

Carta, D. G. (1990). Two fast implementations of the "minimal standard"random number generator. Communications of the ACM, 33(1), 87.

Chambers, W. G., & Dai, Z. D. (1991). Simple but effective modification to a multiplicative congruential random-number generator. IEEE Proceedings: Computers and Digital Technology, 138(3), 121.

Chassing, P. (1989). An optimal random number generator Zp. Statistics and Probability Letters, 7(4), 307.

Elkins, T. A. (1989). A highly random-number generator. Computer Language, 6(12), 59.

Hulquist, P. F. (1991). A good random number generator for microcomputers. Simulation, 57(4), 258.

Kao, C. (1989). A random number generator for microcomputers. OR: The Journal of the Operational Research Society, 40(7), 687.

Knuth, D. (1997). The art of computer programming, volume 2: Semi-numeral algorithms (3rd ed.). Reading, MA: Addison-Wesley.（前掲書）

Komo, J. J. (1991). Decimal pseudo-random number generator. Simulation, 57(4), 228.

Leva, J. L. (1992a). Algorithm 712: A normal random number generator. ACM Transactions on Mathematical Software, 18(4), 454.

Leva, J. L. (1992b). A fast normal random number generator. ACM Transactions on Mathematical Software, 18(4), 449.

Macomber, J. H., & White, C. S. (1990). An n-dimensional uniform random number generator suitable for IBM-compatible microcomputers. Interfaces, 20(3), 49.

Maier, W. L. (1991). A fast pseudo random number generator. Dr. Dobb's Journal, 17(5), 152.

Marsaglia, G., Narasimhan, B., & Zaman, A. (1990). A random number generator for PCs. Computer Physics Communications, 60, 345–349.

Marsaglia, G., & Zaman, A. (1990). Toward a universal random number generator. Statistics and Probability Letters, 8, 35–39.

Morton, M. (1985). A digital dissolve for bit-mapped graphics screens. Dr. Dobb's Journal, #48.

Park, S. K., & Miller, K. W. (1988). Random number generators:Good ones are hard to find. Communications of the ACM, 31(10),1201.

Press, W. H., et al. (1989). Numerical recipes in Pascal: The art of scientific computing (Rev. ed.). Cambridge, UK: Cambridge University Press.

Sezgin, F. (1990). On a fast and portable uniform quasi-random number generator. Simulation Digest, 21(2), 30.

スケールと測度

Celko, J. (2009). Joe Celko's data, measurements and standards in SQL (Morgan Kaufmann series in data management systems).

Crocker, L., & Algina, J. (2006). Introduction to classical and modern test theory. Wadsworth Pub Co.

失われた値

Codd, E. F. (1975). Understanding relations. FDT, 7, 3–4.

Grahne, G. (1989). Horn tables—an efficient tool for handling incomplete information in databases. ACM SIGACT/SIGMOD/SIGART Symposium on Principles of Database Systems, 75–82.

Grant, J. (1977). Null values in relational databases. Information Processing Letters, 6(5).

Grant, J. (1979). Partial values in a tabular database model. Information Processing Letters, 6(5).

Honeyman, P. (1980). Functional dependencies and the universal instance property in the relational model of database systems. Doctoral Dissertation, Princeton University.

Lien, Y. E. (1979). Multivalued dependencies with null values in relational databases. VLDB V, Rio De Janeiro. ACM/IEEE.

Lipski, W. (1981a). On semantic issues connected with incomplete information. ACM Transactions on Database Systems, 262–296.

Lipski, W. (1981b). On databases with incomplete information. Journal of the ACM, 28(1), 41–47.

McGoveran, D. (1993). Nothing from nothing part I (or, what's logic got to do with it?). Database Programming and Design, 6(12), 32.［SQLのNULLの代わりにインジケータを使うべきだと論じた4回連載の第1回］

McGoveran, D. (1994a). Nothing from nothing part II: Classical logic: Nothing compares 2 U. Database Programming and Design, 7(1), 54.

McGoveran, D. (1994b). Nothing from nothing part III: Can't lose what you never had. Database Programming and Design, 7(2), 42.

McGoveran, D. (1994c). Nothing from nothing part IV: It's in the way that you use it. Database Programming and Design, 7(3), 54.

Rozenshtein, D. (1981). Implementing null values in relations. Unpublished.

Rozenshtein, D. (1995). Optimizing transact SQL. SQL Forum Press.

SPARC Study Group on Database Management Systems. (1975). Interim report 75-02-08 to the ANSI X3. FDT-Bulletin ACM SIGMOD, 7(2).

Vassiliou, Y. (1979). Null values in database management—a denotational semantics approach. ACM SIGMOD Conference, 162–169.

Vassiliou, Y. (1980). Functional dependencies and incomplete information. VLDB VI, Montreal. ACM/IEEE.

Walker, A. (1979). A universal table relational database model with blank entries. Unpublished manuscript.

Zaniolo. (1984). Database relations with null values. Journal of Computer and System Sciences, 28(1), 142–166.

正規表現

Friedl, J. E. F. (2006). Mastering regular expressions (3rd ed.). Cambridge, MA: O'Reilly Media.（『詳説正規表現　第3版』Jeffrey E. F. Friedl著，長尾高弘訳，オライリー・ジャパン，2008

Goyvaerts, J., & Levithan, S. (2009). Regular expressions cookbook. Cambridge, MA: O'Reilly Media.（『正規表現クックブック』Jan Goyvaerts／Steven Levithan著，長尾高弘訳，オライリー・ジャパン，2010）

Stubblebine, T. (2007). Regular expression pocket reference: Regular expressions for Perl, Ruby, PHP, Python, C, Java and .NET (2nd ed.). Sebastopol, CA: O'Reilly Media.（邦訳は第1版『正規表現デスクトップリファレンス』Tony Stubblebine著，歌代和正監訳，三島純子訳，オライリー・ジャパン，2004）

Forta, B. (2004). Sams teach yourself regular expressions in 10 minutes. Indianapolis, IN: Sams.

Good, N. A. (2005). Regular expression recipes: A problem-solution approach. Berkeley, CA:Apress.

グラフ理論

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). Introduction to algorithms. McGraw-Hill Companies.（『アルゴリズム・イントロダクション　第3版』（全2巻）T．コルメン／C．ライザーソン／R．リベスト／C．シュタイン著，浅野哲夫／岩野和生／梅尾博司／山下雅史／和田幸一訳，近代科学社，2012）

Even, S. (1979). Graph algorithms. Rockville, MD: Computer Science Press.［絶版］

Harary, F. (1992). Graph theory. Westview Press.［オンデマンド印刷］

Ore, O. (1996). Graphs and their uses (revised by Robin J. Wilson). Providence, RI: American Mathematical Association.（『やさしくくわしいグラフ理論入門』O. オア著，大石泰彦訳，日本評論社，1993）

Rozenshtein, D. (1997). Tree and graph processing in SQL. Peer to Peer Communications.［絶版］

SQL 基礎

Brouard, F. (2001). SQL. CampusPress.［SQL:2003標準をカバーしている。］

Chappell, D., & Trimble, H., Jr. (2001). A visual introduction to SQL (2nd ed.). Wiley.

Date, C. J. (2003). An introduction to database systems (8th ed.). Reading, MA: Addison-Wesley.（『データベースシステム概論原著第6版』C. J. Date著，藤原譲監訳，丸善，1997）

Date, C. J., & Darwen, H. (1996). A Guide to SQL standard (4th ed.). Reading, MA: Addison-Wesley Professional.（『標準SQLガイド』C. J. Date／Hugh Darwen共著，Quipu LLC訳，アスキー，1999）

Groff, J., & Weinberg, P. (2002). SQL: The complete reference (2nd ed.). McGraw-Hill Osborne Media.

Gulutzan, P., & Pelzer, T. (1999). SQL-99 complete, really. Gilroy, CA: CMP Books.

Gulutzan, P., & Pelzer, T. (2002). SQL performance tuning. Boston, MA: Addison-Wesley.

Hellerstein, J. M., & Stonebraker, M. (Eds.). (2005). Readings in database systems (4th ed.). Cambridge, MA: The MIT Press.

Limeback, R. (2008). Simply SQL. SitePoint at www.sitepoint.com.

Pascal, F. (1989). SQL and relational basics. Redwood City, CA: M&T Books.［絶版］

Pascal, F. (1993). Understanding relational databases. John Wiley. ［絶版］

Takahashi, M., & Azuma, S. (2008). The Manga guide to databases (1st ed.). San Francisco: No Starch Press.（『マンガでわかるデータベース』高橋麻奈著，あづま笙子作画，トレンド・プロ制作，オーム社，2005）

Tropashko, V. (2007). SQL design patterns: Expert guide to SQL programming (1st ed.). Rampant Techpress.［本書は入門書ではない。かなり高度な数学を使う。］

van der Lans, R. (2006). Introduction to SQL: Mastering the relational database language (4th ed.). Addison-Wesley Professional.

Viescas, J. L., & Hernandez, M. J. (2007). SQL queries for mere mortals (2nd ed.). Addison-Wesley Professional.

クエリ最適化

Gulutzan, P., & Pelzer, T. (1994). Optimizing SQL: Embedded SQL in C. R&D Technical Books.

Shasha, D. E., & Bonnet, P. (2002). Database tuning. San Francisco: Morgan Kaufmann.

時間データと2000 年問題

ANSI INCITS 30-1997. (R2008). Representation of calendar date and ordinal date for information interchange.

Arnold, R. S. (1995a, January). Millennium now: Solutions for century data change impact. Application Development Trends.

Arnold, R. S. (1995b). Resolving year 2000 problems in legacy software. Presentation at Software Quality Week, San Francisco.

Associated Press. (1995, May 25). Troubled time. Celko, J., & McDonald, J. (1981, February). Father time software secrets allows updating of dates. Information Systems News.［この記事は後に1999 年7月に『ヴァニティ・フェア』に引用された（ロバート・サム・アンソン「Y2Kの悪夢」）。］

Cini, A. (1995, January). System bug of the apocalypse. Internetwork.

Cohn, M. B. (1994, November 21). No need to fear the year 2000. Computerworld, 28(47), 35.

Fine, D. (1995, April 10). Companies brace for millennium. Infoworld.

Furman, J., Marotta, A., & Candiotti, C. (1995, April). Party when it's 1999. Software Magazine.

Hayes, B. (1995, January–February). Waiting for 01-01-00. American Scientist, 83.

Hitchens, R. L. (1991, January 28). Viewpoint. Computerworld.

ISO 8601. (1988). Data elements and interchange formats—information interchange—representation of dates and times.

ISO 8601:2004 (Revises ISO 8601:2000). Data elements and interchange formats—information interchange—representation of dates and times.

Murray, J. T., & Murray, M. J. (1984). Computers in crisis. Princeton, NJ: Petrocelli Books.［2000 年問題に対するコードレベルでの解決法を論じたもの］

Ross, N. (1995, April). The end of the century is nearer than you think. Application Development Trends.

Rubin, H., & Woodward, J. Millenium: A billion dollar software crisis.［コンピュータチャンネルによるビデオテープ］

Sullivan, R. L. (1995, June 19). Ghosts in the machines. Forbes.

Tantzen, R. G. (1980). Algorithm 199: Conversion between calendar date and Julian day number. Collected Algorithms from the ACM (Association for Computing Machinery).

Xenakis, J. (1995, July). The millennium bug: The fine de Siécl computer virus. CFO Magazine.

Zerubavel, E. (1989). The seven day circle. Chicago: University of Chicago Press.

SQLプログラミングテクニック

Celko, J. (1992, March). Implementing T-joins in SQL queries. DBMS Magazine.

Celko, J. (1993, September). Views: More than meets the eye. Database Programming and Design.

Celko, J., & Date, C. J. (1993, September). Access path: Lauds and lectures, kudos and critiques. Database Programming and Design.

古典

Bernstein, P. A. (1976). Synthesizing third normal form relations from functional dependencies. ACM Transactions on Database Systems, 1(4), 277–298.

Codd, E. F. (1970). A relational model of data for large shared data banks. Communications of the ACM (Association for Computing Machinery), 13(6), 377–387.

Codd, E. F. (1979). Extending the database relational model to capture more meaning. ACM Transactions on Database Systems, 4(4), 397–434.

Comer, D. (1978). The difficulty of optimum index selection. ACM Transactions on Database Systems, 3(4), 440–445.

Damerau, F. J. (1964). A technique for computer detection of correction of spelling errors. Communications of the ACM, 7(3).

Date, C. J. (1992, February). According to date: Shedding some light. Database Programming and Design, 15–17.

Dijkstra, E. (1968). Go to statement considered harmful. Communications of the ACM, 11(3), 147–148. [Addresses old versus new JOIN syntax.]

Goodman, N. (1990). VIEW update is practical. InfoDB, 5(2).

Information Systems Week (ISW). (1987, December). Code overload plagues NYC welfare system. Information Systems Week.

Limeback, R. (n.d.). SQL workshop challenge. Explain, (18).

Lorie, R. A., & Daudenarde, J.-J. (1991). SQL and its applications. Englewood Cliffs, NJ: Prentice-Hall.

Martinis, M. (1992, May). Letters column. DBMS Magazine.

Melton, J., & Simon, A. R. (1993). Understanding the new SQL: Complete guide. San Francisco:Morgan Kaufmann.（『SQL92 完全ガイド』J. メルトン／A. R. サイモン著，郡司芳昭訳，トッパン，1998）

Murchison, R. (n.d.). SQL workshop challenge. Explain, (16).

Paitetsky-Shapiro, G. (1983). The optimal selection of secondary indexes in NP-complete. SIGMOD Record, 13(2), 72–75.

Palmer, R. (2007). The bar code book (5th ed.). Bloomington, IN: Trafford Publishing.

Philips, L. (1990). Hanging on the metaphone (a phonetic text-retrieval algorithm better than soundex). Computer Language, 7(12), 38.

Rozenshtein, D., Abramovich, A., & Birger, E. (1993). Loop-free SQL solutions for finding continuous regions. SQL Forum, 2(6).

Smithwick, T. (1991). Pascal version of Lawrence Philips' 1990 hanging on the metaphone (a phonetic text-retrieval algorithm better than soundex). Computer Language, 7(12), 38.

SQL Forum. (1993/1994). See articles by Anatoly Abramovich Yelena Alexandrova, and Eugene Birger (July/August 1993, March/April 1994).

Stevens, S. S. (1957). On the psychophysical law. Psychological Review, 64, 153–181.

Tillquist, J., & Kuo, F.-Y. (1989). An approach to the recursive retrieval problem in the relational database. Communications of the ACM, 32(2), 239.

van der Lans, R. (1990/1991). SQL portability. The Relational Journal, 2(6).

Verhoeff, J. (1969). Error detecting decimal codes. Mathematical Centre Tract #29. Amsterdam:The Mathematical Centre.

Vicik, R. (1993, July/August). Advanced transact SQL. SQL.

更新可能なビュー

Codd, E. F. RV-6 view updating. The relational model for database management, version 2.

Date, C. J. (1986). Updating VIEWs. Relational database: Selected writings. Reading, MA: Addison-Wesley.

Date, C. J., & Darwen, H. (1991). Role of functional dependencies in query decomposition. Relational database writings—1989–1991. Reading, MA: Addison-Wesley.

Goodman, N. (1990). VIEW update is practical. InfoDB, 5(2).

Umeshar, D., & Bernstein, P. A. (1982). On the correct translation of update operations on relational VIEWs. ACM Transactions on Database Systems, 7(3).

理論、正規化、および高度なトピック

Bernstein, P. A. (1976). Synthesizing third normal form relations from functional dependencies. ACM Transactions on Database Systems, 1(4), 277–298.

Codd, E. F. (1990). The relational model for database management: Version 2. Reading, MA:Addison Wesley.

Date, C. J. (1990). Relational database writings, 1985–1989. Reading, MA: Addison-Wesley.

Date, C. J. (1992). Relational database writings, 1989–1991. Reading, MA: Addison-Wesley.

Date, C. J. (1995). Relational database writings, 1991–1994. Reading, MA: Addison-Wesley.

Date, C. J. (1998). Relational database writings, 1994–1997. Reading, MA: Addison-Wesley.

Date, C. J. (2006). Date on database: writings 2000–2006. New York: Apress.

Dutka, A., & Hanson, H. (1989). Fundamentals of data normalization. Reading, MA: Addison-Wesley.

Fagin, R. (1979). Normal forms and relational database operators. Proceedings ACM SIGMOD International Conference on Management of Data.［第5正規形に関する決定版］

Fleming, C. C., & von Halle, B. (1989). Handbook for relational database design. Reading, MA:Addison-Wesley.

Kent, W. (1983). A simple guide to five normal forms in relational database theory. Communications of the ACM, 26(2).

Maier, D. (1983). The theory of relational databases. Redwood City, CA: Computer Science Press.［絶版だが、今でも読むに値する］

Teorey, T. J. (1994). Database modeling and design (2nd ed.). San Francisco: Morgan Kaufmann.

SQL-92、SQL-99

Cannan, S., & Otten, G. (1992). SQL: The standard handbook. McGraw-Hill.［絶版］

Gulutzan, P., & Pelzer, T. (1999). SQL-99 complete, really. Gilroy, CA: CMP Books.

Melton, J., & Simon, A. (1993). Understanding the new SQL: A complete guide. San Francisco:Morgan Kaufmann.（前掲書）

標準および関係組織

2003年2月10日、旧ANSI X3委員会はINCITS（International Committee for Information Technology Standards）という新しい組織に統合された。

- ANSIおよびISO標準

  American National Standards Institute

  5 West 43rd Street, 4 floor

  New York, NY 10036

  Tel:1-212-642-4900

  Fax:1-212-398-0023

  http://www.ansi.org/

- 業界標準の情報源

  IHS Headquarters

  15 Inverness Way East

  Englewood CO 80112

  United States

  Phone:303-397-7956 or 800-854-7179

  Fax:303-397-2740

  E-mail:http://www.ihs.com

- その他コンソーシアム

  NIST (National Institute for Science & Technology)

  Standards Information Center

  100 Bureau Drive, Stop 2100,

  Gaithersburg, MD 20899-2100

E-mail:ncsci@nist.gov

Phone:(301)975-4040

- TPC（Transaction Processing Council）

  Presidio of San Francisco

  572B Ruger St. (surface)

  P. O. Box 29920(mail)

  San Francisco, CA 94129-0920

  Voice:415-561-6272

  Fax:415-561-6120

  E-mail:webmaster@tpc.org

SQL に関係するWebサイト

Whitemarsh Information Systemsのマイケル・M・ゴーマンは、NCITS H2技術委員会の創設時から運営に関わっており、彼のWebサイト（http://www.wiscorp.com/）は標準SQL のドラフトに関する多くの情報を得られる。

統計

米国統計学会（American Statistical Association：ASA）は、資料をオンラインに掲載しており、さまざまな統計計算についての記事を閲覧できる。

www.amstat.org/publications/tas/

www.amstat.org/publications/technometrics/

Chan, T. F., Golub, G. H., & LeVeque, R. J. (1983). Algorithms for computing the sample variance: Analysis and recommendations. The American Statistician, 37(3), 242–247.

Welford, B. P. (1962). Note on a method for calculating corrected sums of squares and products. Technometrics, 4, 419–420.

時間データベース

Date, C., Darwen, H., & Lorentzos, N. A. (2002). Temporal data and the relational model. San Francisco: Morgan Kaufmann.

Jensen, C. S., Clifford, J., Elmasri, R., Gadia, S. K., Hayes P., & Jajodia, S. (Eds.). (1994). A glossary of temporal database concepts. ACM SIGMOD Record, 23(1), 52–64.

Johnston, T., & Weis, R. (2010). Managing time in relational databases:How to design, update and query temporal data. Morgan Kaufmann.

Ozsoyoglu, G., & Snodgrass, R. T. (1995). Temporal and real-time databases:A survey. IEEE Transactions on Knowledge and Data Engineering, 7(4), 513–532.

Snodgrass, R. T. (Ed.), Ahn, I., Ariav, G., Batory, D., Clifford, J., Dyreson, C. E.,Elmasri, R., Grandi F., Jensen C. S., Kaefer W., Kline, N., Kulkarni, K., Leung, T. Y. C., Lorentzos, N., Roddick, J. F., Segev, A., Soo, M. D., & Sripada, S. M. (1995). The temporal query language TSQL2. Norwell, MA: Kluwer Academic Publishers.

Snodgrass, R. T., Boehlen, M. H., Jensen, C. S., & Steiner, A. (1996a). Adding valid time to SQL/temporal. Change proposal, ANSI X3H2-96501r2, ISO/IEC JTC 1/SC 21/WG 3 DBL-MAD-146r2.［以下で閲覧可能 ftp://ftp.cs.arizona.edu/tsql/tsql2/sql3/mad146.pdf］

Snodgrass, R. T., Boehlen, M. H., Jensen, C. S., & Steiner, A. (1996b). Adding transaction time to SQL/temporal. Change proposal, ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/WG3 DBL MAD147r2.［以下で閲覧可能 ftp://ftp.cs.arizona.edu/tsql/tsql2/sql3/mad147.pdf］

Snodgrass, R. T., & Jensen, C. (1999). Temporal databases. San Francisco:Morgan Kaufmann.［本書は絶版だが、アリゾナ大学のWebサイトで閲覧可能である。］

Tansel, A., Clifford, J., Gadia, S. K., Jajodia, S., Segev, A., & Snodgrass, R. T. (Eds.). (1993). Temporal databases: Theory, design, and implementation. Database systems and applications series. Redwood City, CA: Benjamin/Cummings Pub. Co.

Tsotras, V. J., & Kumar, A. (1996). Temporal database bibliography update. ACM SIGMOD Record, 25(1), 41–51.

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. T., Subrahmanian, V. S., & Zicari, R. (1997). Advanced database systems. San Francisco: Morgan Kaufmann.

本文中で引用したもの

Berenson, H., et al. (1995). A critique of ANSI SQL isolation levels. Microsoft Research Technical Report: MSR-TR-95-51.

Bose, R. C., & Nelson, R. J. (1962). A sorting problem. Journal of the ACM, 9(2): 282–296

Gonzales, M. L. (2003). The IBM data warehouse. New York: Wiley.

Gonzales, M. L. (2004, September 18). The SQL language of OLAP. Intelligent Enterprise.

Hegeman, F. (1993, February). Sorting networks. The C/C++ User's Journal.

Larsen, S. (1996, Winter). Powerful SQL: Beyond the basics. DB2 Magazine.［以下のWebサイトでも閲覧可能 http://www.db2mag.com/db_area/archives/1996/q4/9601lar.shtml］

Melton, J. (1998). Understanding SQL's stored procedures. San Francisco:Morgan Kaufmann.

Moreau, T., & Ben-Gan, I. (2001). Advanced transact-SQL for SQL server 2000. Berkeley, CA:Apress.

Pascal, F. (1988/1998). SQL redundancy and DBMS performance. Database Programming and Design, 1(12).

The Unicode Consortium. (2003). The unicode standard, version 4.0. Reading, MA: Addison-Wesley.

Watson, E. J. (1962). Primitive polynomials (mod 2). Mathematics of Computation, 16, 368–369.

Zemke, F., Kulkami, K., Witkowski, A., & Lyle, B. Introduction to OLAP Functions. ANSI document.

翻訳・監修を終えて　――ミック

本書第2版の邦訳を最初に読んだときのことは、よく覚えている。当時、私は新米のDBエンジニアで、SQLという言語のつかみどころのなさに（他の多くのエンジニアやプログラマと同じく）、戸惑っていた。簡単なことを簡単に行えるという点では優れているが、少し難しいことをやろうとすると、途端に視界の利かない闇夜の森の中に迷い込んだような気分になる。内部的にどう動いているのか見えにくいし、意味のわからないところでパフォーマンスが悪化する。どうにも扱いにくい言語だ、というのが私のSQLに対する第一印象だった。

また当時は、SQLに関する良い解説書もあまりなかったと記憶している。初心者向けの参考書は簡単な操作を行うためのサンプル集という感じで、新米ながらに、SQLの難しさや扱いにくさについての肝心なポイントについてお茶を濁していると、不満を持ったものだ。そんなときに見つけたのが、本書の第2版だった。第2版の前書きにも「本書は初心者向けではない」とは書いてあったのだが、当時の私としては「SQLに初心者も上級者もあるか」というのが偽らざる気持ちだった。無知というのは恐ろしいものだ。

「どうせこの本も大したこと書いてないだろ」とたかをくくって読み始めた最初の感想は、次のようなものだった。

何が書いてあるのか全然わからないんですけど。

いや、本当に参った。よくチンプンカンプンな本について「理解できるところのほうが少ない」という言い方をするが、まさにその状態を地でいっていたのだから。しかし、理解できた数少ないパートに関して言えば、「これは極めて重要なことが書いてある」というのが感想だった。つまり私にとって本書は「非常に重要で目から鱗が落ちる」と「まったく意味不明」がない混ぜになった奇妙な本として現れたのだった。だが、自分が理解できた部分に重要なことが書いてあるということは、きっと理解できない部分にも同じぐらい重要なことが書いてあるに違いない。そう確信した私は、本書の「解読」に取り掛かった。まるで気分は古文書の解読だが、本当に、少しずつ岩を噛み砕くようにして本書を咀嚼することで、私はリレーショナルデータベースとSQLの奥深い世界に降りていくことができるようになったのだった。

その過程で、私はコッドに出会い、デイトに出会い、彼らの著作を読むことで多くを学び、そのあとで本書を読み返すとさらに理解できることが増えた。そのサイクルを何度か繰り返すなかで、徐々に、私が最初にSQLに感じた違和感が、SQLの集合指向的なコンセプトが手続き型と真っ向から対立することにあるのだということを理解できるようになった。そして、SQLが手続き型と物理層と決別した理由に感銘を受け、HAVINGやEXISTSといった道具が集合論や述語論理のロジックを表現しているという事実に驚き、SQLとリレーショナルデータベースの世界が持つ豊かさを見つけるたびに、目を見張ったのだった。

それ以来、私は何とかして、この素晴らしい、だが難解な本の伝えようとしていることをわかりやすく解説しようと努力を重ねてきた。ものぐさを自負する私が自分でWeb サイトを作って記事を書き、その内容に興味を持った翔泳社からWebマガジンで連載をさせてもらい、後にその連載は本になった（それが『達人に学ぶSQL徹底指南書』である）。2007年には、著者のもう一冊の著作である『SQL パズル』を訳すという機会を得て、セルコ本人とメールで意見を交わすという贅沢な経験を持った。その頃から、本書の最新版を自分で訳したい――今の自分ならば多くの人にわかりやすく伝えられる――という思いを持つようになった。この浩瀚（こうかん）な本の翻訳が骨の折れる作業になることは承知していたし、事実やってみて予想以上に大変だったのだが、ようやく10年来の願いを達成することができて、私としても肩の荷が下りた気分である。読者の皆さんに、10 年前の私が感じた感動を、難しさを半減しつつ感じてもらえれば、訳者として、それ以上言うことはない。

翻訳においては、坂井恵、大神驍人の両名に大変お世話になった。間違いの修正や訳文の拙さの指摘など有益なコメントを数多くいただいたおかげで、本書のクオリティは訳者1人では絶対に達成できなかったレベルを実現できた。また、翔泳社の片岡仁氏には、長期間に及んだ翻訳プロジェクト全般にわたってお世話になった。ここに謝意を表したい。

索引

※末尾にf が付くページは図を、t が付くページは表を示します。

■ 数字・記号

<drop behavior>句　92
<references specification>　73
10進数表現　60
1NF　171-176
1レベルのSELECT文　408
2000年問題　738
　～の影響　743
　閏年　739-741
　ゼロ問題　738-739
　千年に一度　741-742
　レガシーデータにおける奇妙な日付　742
2NF　176-178
2値論理　266, 267t
3NF　178-180
3値論理　266, 267, 267t
　～演算　266t
　～とEXISTS　393-395
　論理的述語　269
4NF　183-184
4ビットのジェネレータ　60f
5NF　184-186

■ A

A5 →データ項目の制約違反（A5）
A5A →読み込み異常（A5A）
A5B →書き込み異常（A5B）
ABSOLUTEフェッチ　137
ABS関数　213, 220, 588
ACID特性　12
ActualScheduleテーブル　188t, 189
ADD COLUMN句　93
ADD句　93
AFTERトリガー　125
ALLオプション　410, 758
　EXCEPT　658, 673
　INTERSECT　658
ALL述語
　～と極値関数　403-404
　～の使用　401-402
　EXISTS述語に変換　401-402
ALTER COLUMN句　93
ALTER TABLE文　93-94
AND集約関数　555-556
ANSI/ISOの一時テーブル　764
ANSIのシーケンス　41
ANY述語　401, 402

EXISTS述語に変換　402
AS IDENTITY句　40
ASSERTION　8, 88
AS演算子　478
Atomicity　12-14
AVG集約関数　532, 577
データがないグループを含む平均　534-536
複数列の平均　536-537

■ B

BCNF（Boyce-Codd Normal Form）
➡ボイス–コッド正規形（BCNF）
BEFOREトリガー　125
BETWEEN述語　322, 359-362
インデックスの構造　362
使わない場合　372
入力がNULLの場合の結果　361
入力が空集合の場合の結果　361
プログラミングの小技　361-362
BIGINT型　204
BIT_LENGTH関数　247
BIT型　206-208
BOOLEAN型　206-208
BYTE型　206-208

■ C

CASCADEオプション　74, 93, 474, 763
CASE（Computer Assisted Software
　Engineering）ツール　170, 186
CASE式　331-339, 509
　～の入れ子　107, 108, 110, 111, 112t
　～とLIKEの検索条件　351-352
　～によるクロス表　602
　～による集約の階層化　521-522
　～の強力さ　310
　CHECK制約　336-339
　COALESCE関数　334-335
　ELSE句　332, 333, 334
　GROUP BY句　335-336
　IF-THEN-ELSE文を置き換え　113
　NULLIF関数　334-335
　ORDER BY句と～　133-136
　THEN句　333
　UPDATE文　310-312
　WHEN句　332, 333, 334
　行のソート　341-345
　検索～　332
　条件法　336-339
　ローゼンシュタインの特性関数　339-340

CAST関数　211
CEILING関数　213, 220
CEIL関数　213
CHAR(32)列　285
CHAR(39)形式での格納（IPアドレス）　223
CHAR_LENGTH関数　247
CHARACTER_LENGTH関数　247
CHARINDEX関数　248
CHECK句　8, 27, 100
CHECK制約　69-71, 88, 641, 679
　～としてのWITH CHECK OPTION　472-474
　～の中でIN述語を使用　382
　EXISTS述語　392
　WITH CHECK OPTIONとの違い　469
CLOSE文（カーソル）　137
COALESCE関数　109, 216-217, 334-335, 536-537
COBOL　9
　～のOCCURSキーワード　6
　～メソッド（時間の表現）　229
COMMIT文　12
CONCAT関数　247
Consistency　14
CONTAINS述語　670
COUNT(DISTINCT x)　514
COUNT集約関数　528

DISTINCT付きの集約関数を最適化　530-531
カーディナリティのカウント　528
式のカウント　528, 530
CREATE　127
CREATE ASSERTION文　8, 28-35, 88-89
CREATE CHARACTER SET文　36
CREATE COLLATION文　36-37
CREATE DOMAIN文　26-27, 100-101
CREATE PROCEDURE文　101-102, 106, 123-124
CREATE SCHEMA文　25-26
CREATE SEQUENCE文　27-28
CREATE TABLE文　65-75
CHECK制約　69-71
DEFAULT句　67
NOT NULL制約　67-69
PRIMARY KEY制約　71-73
REFERENCES句　73-75
UNIQUE制約　71-73
列制約　65-67
CREATE TRANSLATION文　37
CREATE TRIGGER文　101, 106, 124-126
～による更新　458
～の利点　126
AFTERトリガー　125
BEFOREトリガー　125

OLD遷移表　125
トリガー　101, 124
トリガーイベント　124
CREATE VIEW文　107
ビュー定義　455-456
CROSS JOIN →クロス結合
CTE →共通表式
CUBE（GROUP BY句）　559, 561-562
CUME_DIST関数　567-568, 592
CURRENT OF　140, 305
Current Resident　95
CURSOR STABILITY分離レベル　19

■ D

DATA DIVISION　9, 64
DBA　23, 24
DBMS　99
DCL（Data Control Language）　2, 3
DDL（Data Declaration Language）　2
　～におけるモデリングクラスの階層　96-99
DEALLOCATE文　137
DECIMAL型　203, 204
DECLARE　127
DECLARE CURSOR文　128
　UPDATE文/DELETE文　140
　カーソルの使い方　137-140
DEFAULT句　67
DELETE FROM句　292
DELETE FROM文　291-302
　～の形式　291
　ROWIDを使った重複行削除　299-301
　WHERE句　292-297
　位置付け型削除　292
　探索型削除　292
　テーブルから重複行を削除　299
　同一テーブルのデータを削除　298-301
　複数のテーブルから削除　301-302

別のテーブルに基づいた削除　297
DELETE文　140
DENSE_RANK関数　566-567
DISTINCT　131, 410
～付きの集約関数を最適化　530-531
IS [NOT] DISTINCT FROM　265, 322-323
NULLと重複行に対するINTERSECT/EXCEPT　662
DKNF ➡ドメイン−キー正規形（DKNF）
DML（Data Manipulation Language）　2
DOUBLE PRECISION　205
DOW関数　730
DRI（Declarative referential integrity）　➡宣言整合性制約（DRI）
DROP ASSERTION文　29
DROP COLUMN句　93
DROP CONSTRAINT句　93
DROP TABLE文　92-93, 475
Durability　15
DWH（Data WareHouse）　557, 558

■ E

EKNF（Elementary Key Normal Form） ➡基本キー正規形（EKNF）
ELSE句（CASE式） 332, 333, 334
ER図 195
EST（Eastern Standard Time） 232
EXCEPT集合演算子 658-663
　ALLオプション 658, 673, 758
　NULLと重複行がない場合の～ 661-662
　NULLと重複行に対する～ 662-663
EXISTS述語 297, 385, 762
　～をORでつなげる 383
　3値論理 393-395
　ALL述語を書き換え 401
　ANY述語を書き換え 402
　CHECK制約 392
　IN述語に書き換え 374, 388-389
　NOT EXISTS 377, 767
　NULL 387-389
　SELECT DISTINCTを書き換え 758
　インデックス 385
　外部結合 390-391
　参照整合性制約 392-393
　内容結合 390

量子化　391-392
EXP関数　213, 220
EXTRACT関数　712

■ F

FAST_FORWARD　138
FETCH文　136-137
FIFO　508
　　～の部分集合　507-510
FIRST_VALUE関数　571-573
FLOOR関数　213, 220
FORWARD_ONLY　138
FROM句　479
　　～でのクロス結合　412
　　～とWHERE句　511
　　1レベルのSELECT文　408
　　DELETE FROM句　292
　　SELECT文　408
　　UPDATE文　312
FULL OUTER JOIN →完全外部結合functional dependency（FD）　➡関係従属性

■ G

GENERATED ALWAYS　40, 41

GENERATED BY DEFAULT　41

GLOBAL TEMPORARY TABLE　476

GMT（Greenwich Mean Time）　227

GREATEST関数　221, 536

GROUP BY GROUPING SETS　559-561

GROUP BY句　642

　～でCASE式を使う　335

　～とHAVING句

　　COUNT(DISTINCT x)　514

　　グループ特性　516-518

　～とソート　525-526

　～を避ける　759

　1レベルのSELECT文　408-415

　FROM句とWHERE句　511

　NULLとグループ　513

　SELECT句　512

　SELECT文と～　409

GROUPING SETS関数　559-561

GROUPING演算子　559

　CUBE　559, 561-562

　GROUPING SETS　559-561

　OLAP　562

ROLLUP　559, 561
GROUPING関数　559, 560
GUID（Global Unique Identifier）　45-46, 47

■ H

HAVING句　503, 515, 580
～とGROUP BY句
COUNT(DISTINCT x)　514
グループ特性　516-518
～を避ける　759
SELECT文　409-410, 415t

■ I

IANA（Internet Assigned Numbers Authority）　284
IDENTITY列　40-44
　～とシーケンス　41, 44
INDICATOR　127, 215, 271
INNER JOIN →内部結合
INSERT INTO句　302-303
INSERT INTO文　302-305
　DEFAULT VALUES句　303
　INSERTの性質　303-304
　クエリインサート　303
　構文　302
　テーブル定数　303
　バルクロードとデータ抽出ツール　304-305
INSTEAD OFトリガー　125, 458
INTEGER型　204
　整数列　146t, 282-283
INTERSECT集合演算子　658-663, 758
　ALLオプション　658
　NULLと重複行がない場合の～
　　661-662
　NULLと重複行に対する～　662-663
INTERVAL型　233, 234-236
IN述語　373-374, 446

～でORを置き換える　378
～と参照整合性制約　381-382
～とスカラサブクエリ　383-384
～を展開　373, 379
CHECK制約で使用　382
EXISTS述語に書き換え　374, 389
NOT～　377, 378, 379
NULLと～　378-381
結合演算で置き換え　376
結合へと同値変換　755
最適化　374-378, 753-755
サブクエリを伴う～　374, 377, 378
IPv4アドレスの格納方法　281-284
IPv6アドレスの格納方法　284-285
IPアドレス　222-224
IRR（Internal Rate of Return）　160, 161-164
IS [NOT] DISTINCT FROM演算子　322-323
IS [NOT] NORMALIZED述語　328-329
IS [NOT]{TRUE | FALSE | UNKNOWN}述語　326-328, 327t
IS NULL述語　325-326, 326t
NULLが発生する理由　326
ISO-8601フォーマット　711, 712
Isolation　14
ITU（International Telecommunication Union）　228

■ J

JOIN →結合

■ L

LAG関数　570-571

LAST_VALUE関数　571-573

LEAD関数　570-571

LEAST関数　221, 280, 536

LEFT OUTER JOIN →左外部結合

LEFT関数　247

LENGTH関数　247

LIFO　508

　　～の部分集合　507-510

LIKE述語　347

　　<escape character>　347

　　<match value>　349

　　<pattern>　347, 349

　　CASE式　351-352

　　NULLの結果　350

　　空文字列の結果　350

　　結合でLIKEを避ける　350-351

　　検索条件　351-352

　　同値性ではない　350

　　文字列パターンのトリック　348

LIST集約関数　546

　　クロステーブルとの併用　547-549

　　再帰共通表式と～　546-547

LN関数　213, 220
LOCATE関数　248
LOWER関数　245, 247

■ M

MATCH FULL　74
MATCH PARTIAL　74
MAX集約関数　537
MD5（Message-Digest algorithm 5）　46
MERGE文　314-316, 757
Metaphone　252-255
MIN集約関数　537
MOD関数　213, 218-220
　実装別の動作パターン　220t
　乱数　668
MQT　485
MUCK（Massively Unified Code-Key）テーブル　157

■ N

NaN（Not a Number）　206

NATO　48, 227

NEXTVAL式　44

　　シーケンスへアクセスする構文　141

NO ACTIONオプション　75

NOT EXISTS述語　390, 480, 767

　　NOT IN述語の置き換え　377

　　外部結合　390

NOT IN述語　377, 378, 379

NOT NULL制約　67-69, 72

NPV（Net Present Value）　160, 161, 163

NP完全　85, 748

NULL

　　～とIN述語　378-381

　　～と外部結合　428-430

　　～とホスト言語　270-271

　　～と論理　266-269

　　EXISTSと～　387-389

　　NULL⇔値の変換　215-217

　　ORDER BYと～　130-133

　　関数と～　270

　　行比較と～　320-322

　　サブクエリの述語における～　267-279

算術と～　214
数学と～　270
設計上のNULLの扱い　271-275
データ型での扱い　206-207
比較　264-265
複数の意味を使い分ける方法 275-277
ホスト言語で使用を避ける　274-275
NULLIF関数　215-216, 334-335
NUMERIC型　203, 204
NYSIISアルゴリズム　255-256

■ O

OCTET_LENGTH関数　247

OLAP　512, 557

　～関数　562

OLTP（Online Transaction Processing）　558, 641

　DWHでの違い　557

ON COMMIT ROWS　90

OPEN文　136

ORDER BY句　128

　～とCASE式　133-136

　～とNULL

　　カーソル結果順序　130, 130t, 131t

　　ソートキー　131

　ウィンドウ句の～　564

　標準構文　129

OR集約関数　554-555

OSF（Open Software Foundation）　46

OTLT（One True Look-up Table）

　157-159

OUTER JOIN →外部結合

OVERLAPS述語　363

　～のルール　363

　時間の期間と～　363-372, 367f, 369f

　使わない場合　372

OVER句　563, 569, 570

■ P

P0　15-16
P1　16
P2　16, 21
P3　16
P4　16
PARTITION BY句　563
Pascalのvariantレコード　6
PERCENT_RANK関数　567-568, 592
PlannedScheduleテーブル　188t, 189
POSITION関数　247
POWER関数　213, 220
PRD集約関数　549
　式による～　550-551
　対数による～　551-553
PREVVAL式　44
PRIMARY KEY制約　71-73, 73t

■ Q

QNaN（Quiet NaN）　206

■ R

RANK関数　566-567
READ COMMITTED分離レベル　18
READ UNCOMMITTED分離レベル　18
READ_ONLY　138
REAL型（数値型）　205, 213
REFERENCES句　73-75
RELATIVEフェッチ　137
REPEATABLE READ分離レベル　18
REPEAT関数　248
REPLACE関数　37, 249, 257
　経路列挙モデルでの使用　685
REPLICATE関数　248
RESTRICTオプション　93, 763
REVERSE関数　248
RIGHT OUTER JOIN →右外部結合
RIGHT関数　247
RIRs（Regional Internet Registries）　284
ROLLBACK文　12
ROLLUP（GROUP BY句）　559, 561
ROUND関数　221
ROW_NUMBER関数　84, 301, 566
　行列のソート　648
ROWID　40

～を使った重複行削除　299-301

■ S

SAVEPOINT　13

schedule_right制約　473

SELECT DISTINCT　51, 376, 404, 657

　　EXISTS述語で変換　758

　　ビュー　465, 467

SELECT句　410, 512

SELECT文　22, 106, 407, 642

　　高度な～　417

　　実行順序　407, 408-410

　　スカラ～　437-438

　　最も単純な～　408-415

　　　　FROM句　408

　　　　GROUP BY句　409

　　　　HAVING句　409, 410, 415t

　　　　WHERE句　408, 409

　　　　実行順序　408-410

SERIALIZABLE分離レベル　18

Seriesテーブル　55, 109, 146

SET DEFAULTオプション　75

SET NULLオプション　75

SET句（UPDATE文）　306-307

　　複数のUPDATE文　311, 312, 343

　　別のテーブルを使った更新　307-310

SIGN関数　220
SIMILAR TO述語　353-354, 353t, 354t
SMALLINT型　204, 223-224
SMALLINT列　284
SNaN（Signaling NaN）　206
Soundex　250-252
　～関数　249-250, 489
SPACE関数　248
SQL
　記述統計における多変量　604-607
　行比較　320-322
　最適化　745-746
　時間型　228-234
　実装別のMOD関数の動作パターン　220t
　文字列における問題　244-246
SQL:2006　607-608
SQL-89　243, 245
SQL-92　243, 246
SQLSTATE　138
　～コード　138, 139, 139t
SQL文の削減　757
SQRT関数　213, 220
SSN（Social Security Number）　286
SUBSTRING関数　247
SUM集約関数　531-532

■ T

t0　20
THEN句（CASE式）　333
TINYINT型　204
TRANSLATE関数　37, 247
Tricolorドメイン　263
TRIM関数　247
TRUNCATE関数　221

■ U

UCS（Universal Character Set）　35
UID（Universal Identifier）　46, 48
Unicodeコンソーシアム　35
UNION　541, 654-656, 758
　～を避ける　755-756
　UNION ALLとの混在　657
　同じテーブルの列に対する～　657
　実行順序　656
　ビュー　460-462
UNION ALL　654-656, 756, 758
　UNIONとの混在　657
　実行順序　656
　ビュー　460-462
UNION JOIN演算子　435-437
UNIQUE述語　404-405
UNIQUE制約　71-73
　入れ子の～　75-79
　　一列の一意性と複数列の一意性　82-88
　　オーバーラップするキー　79-82
UPDATE句　305-306
UPDATE文　22, 140, 305, 343, 469
　～でCASE式を使用　310-312
　～の形式　305

FROM句　312
SET句　306-307
WHERE句　306
加算の結果を反映　646
行列の転置　647
別のテーブルを使った更新　307-310
UPPER関数　245, 247
UTC（Universal Coordinated Time）　227, 228
UUID（Universal Unique Identifier）　45, 47-48

■ V

VALUES構築子　480
VARCHAR(15)列　282

■ W

WHEN句（CASE式）　332, 333, 334

WHERE句　292

　　～とFROM句　511

　　1レベルのSELECT文　408, 409

　　DELETE FROM文　292-297

　　UPDATE文　306

　　クロス結合　414

　　相関サブクエリ　409

　　複雑な条件　297

WIMP（Windows, Icons, Menus, Pointing device）インターフェイス　151

WITH CHECK OPTION句　468

　　CHECK制約との違い　469

　　CHECK制約としての～　472-474

　　SQL-89標準　468

　　UPDATE文　469

■ Y

Y2K危機　738

■ あ

アームストロングの公理系　186

アクセス制御のための言語　3

値のリスト

　　値の入れ替えと移動　629-630

　　シーケンスの始点と終点を見つける　630-631

　　リストの列挙（数列テーブル）　146-148

　　リストを折り返して表示　631-632

後入れ先出し　508

アルゴリズム

　　NYSIIS　255-256

　　音声　249

　　降順　449

　　昇順　448

　　ツェラーの公式　730

　　ハッシュ　232

安定な結婚問題　441-444

■ い

一意性　4
　　一列と複数列の～　82-88
一意制約　71, 72, 73t
一意な値　52-53
　　～の生成　52-53
　　事前に割り当てられる値　54-55
一意な識別子　45
一意量子化　397
一時テーブル　89
　　～の宣言　89-91
　　ANSI/ISOの～　764
　　グローバル～　89, 90
　　ローカル～　89, 90
一貫した状態　14
一貫性（Consistency）　14
入次数　704
入れ子
　　UNIQUE制約　75-88
　　木の表現　688t, 689t
　　ビュー　462-463
入れ子集合モデル　687-689
　　入次数と出次数　704
　　エッジ　704

階層集約　692-693
木の表現　688t, 689t
グラフのノード　702-703
経路の両端　703
孤立ノード　705
座標からわかる情報　689-690
シンクノード　705
ソースノード　705
到達可能ノード　703
同レベルのノード間の序列　691-692
内部ノード　705-706
ノードと部分木を削除　693-694
非循環グラフを入れ子集合に変換
706-708
包含関係　690-691
隣接リストモデルに変換　694-695
インジケータ　127, 136
インデックス　45
～とビュー　468
～を文字列に作る　357
種類　748
ツリー～　746-767
ハッシュ～　747-748
ビットベクトル～　748
複数列に対する～　753

インデックスアクセス　746
インデックス設計　748-749
インライン展開　465-467

■ う

ウィンドウ句　563
　　ORDER BY句　564
　　PARTITION BY句　563
　　ウィンドウフレーム句　564-565
ウィンドウ集約関数　566
　　～を使った中央値　585
失われた更新　16
閏年　739-741

■ え

エッジ　677, 697
　　グラフを入れ子集合モデルで表す　702
演算
　　四則～　212-213
　　数論演算子　218-220
エンティティ　4

■ お

黄金比　115
オプティマイザ　745, 761-762
音声アルゴリズム　249
　Metaphone　252-255
　NYSIISアルゴリズム　255-256
　Soundex　249-250, 250-252

■ か

カーソル　123, 127
- CLOSE文　137
- DEALLOCATE文　137
- DECLARE CURSOR文　128
- FETCH文　136-137
- OPEN文　136
- ORDER BY句とCASE式　133-136
- ORDER BY句とNULL　130-133
- ORDER BY句の標準構文　129
- SQLSTATEコード　138, 139, 139t
- 位置付けられたUPDATE/DELETE 文　140

カーディナリティ　528
会計的中央値　577
開始タイムスタンプ　20, 21
階乗関数　482
階層構造
- 入れ子集合モデル
  - 階層集約　692-693
  - 木の表現　688t, 689t
  - 座標からわかる情報　689-690
  - 同レベルのノード間の序列　691-692
  - ノードと部分木を削除　693-694
  - 包含関係　690-691

隣接リストモデルに変換　694-695
経路列挙モデル
木の深さと部分木を見つける　685-686
整合性制約　686-687
ノードと部分木の削除　686
部分木とノードを見つける 684-685
階層集約（入れ子集合モデル）　692-693
階層補助テーブル　156-157
下位中央値　578
外的結合　105
外部キー　159, 459
外部結合（OUTER JOIN）　423-424
～と集約関数　434
～の歴史　424-428
NOT EXISTSと～　390-391
NULLと～　428-430
完全～　434-435
自己～　431-432
自然～　430-431
条件付き～　430-431
スカラサブクエリで書き換え　437
左～　425, 430, 434
右～　425, 429, 434
書き込み異常（A5B）　21
加算　212, 483

数
素数　56-59
フィボナッチ数列　115-116
丸めと切り捨て　209-211
有理数　289
仮数　205
型変換　318-320, 319t
カッター分類法　256-257
空のテーブル　261
カレンダー
～テーブル　145
～標準　225-228
個人的な～　713-714
日付フォーマット　230
補助テーブル　736-738
関係従属性　170-171
アームストロングの公理系　186
データベースの～　186
関係除算
～と入れ子のEXISTS　497
結合を使った～　500
厳密な～　496-497
集合演算を使った～　501
剰余を持った～　495-496
トッドの除算　497-499

関係モデル　167
　　コッドの12のルール　168-170
関数
　　～とNULL　270
　　COALESCE～　216-217
　　NULLIF～　215-216
　　Soundex～　249-250
　　距離～　279-281
　　時間～　730
　　時間型のデータ抽出　712
　　指数～　220
　　四則演算　212-213
　　集約～　527
　　述語で使うための～　116-117
　　小数を扱う～　220-221
　　数学～　218-222
　　数値から文字への変換　221-222
　　標準SQL　245
　　文字列～　246-248
完全外部結合　434-435
関連テーブル　75

■ き

木
- 抽象的な～　677, 678f
- リーフ　677
- 隣接リストモデル
  - テーブルの更新　682-683
  - 手続き的な木の探索　681-682
  - 複雑な制約　679-680

キー　197
- ～の分類　197t
- サロゲート～　199
- 自然～　197
- 人工的な～　197t
- 露出した物理的ロケータ
  - 39-44, 198-200

幾何平均　604

期間のシーケンス　714
- ～おける欠落　715-717
- 日／時間の開始と終了　727
- 日付の引き算　725-727
- 連続的な期間　717-722
- 連続的な期間における失われた時間　722-725

期間の単位　233

疑似同値性　265, 606

記述統計　575
AVG関数　577
SQL:2006の統計関数　607-608
クロス表　599-603
最頻値　575-577
多変量　604-607
中央値　577-587
調和平均と幾何平均　603-604
分散と標準偏差　587-588
平均偏差　588
累積統計　589-598
疑似乱数　232, 666
～関数　667, 669
～生成関数　668
基数（Cardinal numbers）　146, 260
基数（radix）　204
機能的凝集　104
基本キー正規形（EKNF）　180-181
逆関数　648
補助テーブルを使った～　161-164
キャラクタセット　35-37, 244
CREATE CHARACTER SET文　36
CREATE COLLATION文　36-37
CREATE TRANSLATION文　37
行（row）　3, 64

～ VS レコード　6
～のソート　648
～の比較　320-322
～を指定したUPDATE/DELETE
140
式の比較　375
連番の割り当て　566
境界クエリ　617-618
凝集度　104
共通的結合　105
共通表式　3, 480-481
再帰的な～ ➡再帰共通表式
単純な～　481
共分散　604-605
行列式　648
行列操作
加法　645-646
逆関数　648
行列式　648
減法　645-646
固有値　648
乗法　646-647
相等性　643-645
ソート　648
転置　647-648

極小被覆　187, 192
極値関数　537
　　～の一般化　539-545
　　ALL述語と～　403-404
　　簡単な～　537-539
　　複数の基準の～　545-546
距離関数　279-281
切り捨て　209-211

■ く

偶発的凝集　104

クエリ

- ～によるデータ分割　487-510
- 境界～　617-618
- 古典的な方法　760
- シーケンスを求める～　618-623
- ランを求める～　618-623

区間　609

グラフ

- 入れ子集合モデル
  - 入次数と出次数　704
  - エッジ　704
  - 経路の両端　703
  - 孤立ノード　705
  - シンクノード　705
  - すべてのノード　702-703
  - ソースノード　705
  - 到達可能ノード　703
  - 内部ノード　705-706
  - 非循環グラフを入れ子集合に変換　706-708
- エッジ　677, 697
- 経路　697
- 多角形の中の点　708-710

隣接リストモデル
SQLと～　699-700
ノード　698, 698f
隣接行列モデル　700-702
グリニッジ標準時　227
グルーピング
～演算子　511
GROUP BY句　512, 642
GROUP BY句でペアを作る　523-525
GROUP BY句とソート　525-526
GROUPING演算子　559-562
NULLとグループ　513
グループ化したビュー　518-519
グループの特性とHAVING句　516-518
計算列による～　522-523
グループ化 ➡グルーピング
グレゴリオ暦　729
グローバル一時テーブル　89, 90
グローバル定数テーブル　165-166
クロス結合（CROSS JOIN）　408, 412t
～を避ける　760-761
FROM句　412
WHERE句　414
クロス集計表　561
クロス表　599

CASE式による～　602-603

■ け

計算

〜関数　212-213

算術とNULL　214

時間の〜　236-237, 711-713

計算列　107, 142

〜によるグループ化　522-523

ビュー　458-459

経路　697

グラフ　697-698

経路列挙モデル

木の深さと部分木を見つける　685-686

整合性制約　686-687

ノードと部分木の削除　686

部分木とノードを見つける　684-685

結合（JOIN）

UNION JOIN　435-437

外部結合 ➡外部結合

完全外部結合　434-435

結合構文の新旧対決　438-439

コッドのT-結合　447-453

自然結合　430

条件付きの結合　439-447

内部結合　390

標準的な構文の内部結合　422-423
結合演算　500
結合演算子　422, 438
結合射影異常　185
結合射影正規形　184, 185
結合テーブル　75
結合度　105
欠番　54
～のある連番　53-54
検索CASE式　332
検索条件　749-750
CASE式とLIKEの～　351-352
文字列式の最適化　750-751
原子性（Atomicity）　12-14

■ こ

合計　626-629
交差（INTERSECT） ➡INTERSECT
降順アルゴリズム　449
更新異常　172
更新可能なビュー　456-458
互換可能性　51
コストベースのオプティマイザ　745
コッドの12のルール　168-170
コッドのT-結合　447-453
　降順アルゴリズム　449
　昇順アルゴリズム　448
　手続き型のアプローチ　450-453
固有値　648
孤立ノード　705
コントロールブレイクレポート　6

■ さ

差（EXCEPT） ➡EXCEPT
再帰　114
　　～関数　482
　　線形～　482
再帰共通表式　481-485
　　～とLIST関数　546-547
　　単純な加算　483
　　単純なツリー検索　483-485
　　フィボナッチ数列　115
　　隣接リストモデル　682, 699
最適化
　　DDLの制約　751-752
　　DISTINCT付きの集約関数　530-531
　　IN述語　374-378, 753-755
　　SQLの～　745-746
　　SQL文の削減　757
　　UNION　755-756
　　アクセス方法　746-748
　　一時テーブル　764-766
　　インデックス設計　748-749
　　オプティマイザの癖　761-762
　　クロス結合　760-761
　　検索条件　749-750

サブクエリ　756-757
時間式　751
スキーマ　763-764
ソート　758-760
複数列に対するインデックス　753
文字列式　750-751
最頻値　575-577
先入れ先出し　508
削除異常　172
サブクエリ　756-757
～による集約の階層化　519-521
～の述語におけるNULL　267-279
～を避けて結合を使う　756-757
IN述語とスカラ～　383-382
相関～　417-421
量子化と～　397
サマータイム　228, 233-234
サロゲートキー　199
算術関数　213
参照整合性制約　73, 126
EXISTSと～　392-393
IN述語と～　381-382
トリガーと～　126
参照補助テーブル　151-159
OTLT（単一参照ーブル）　157-159

階層補助テーブル　156-157
パラメータを持つ補助テーブル
　　154-155
範囲補助テーブル　155-156
読み替えを行う補助テーブル　153-154

■ し

シーケンシャルアクセス　746
シーケンシャルインデックス　746, 753
シーケンス　27, 141-142
　～を求めるクエリ　618-623
　ANSIの～　41
　IDENTITY列と～　41, 44
　最小値と最大値　630
　始点と終点を見つける　630-631
　歯抜けを許した～　618
シーケンス生成関数　52
　一意な値の生成　52-53
　欠番のある連番　53-54
シータ演算子　318
シード　232, 668
時間
　～関数　730
　～データのモデリング　733-736
　～の扱い　232-233
　～の計算　236-237, 711-713
　期間同士の関係　240-241
　期間のモデリング　237-240
　週　730-733
　シンプルな時間式　751

タイムゾーン　228, 232, 233-234
連続的な期間における失われた～　722-725
時間型　228-234
INTERVAL型　234-236
時間の扱い　232-233
主要な～　712
タイムスタンプの扱い　230-232
データベース内部での時間の表現　229
表示フォーマットの標準　230
時間的凝集　104
識別子
～の例　48-50
GUID　45-46
UID　48
UUID　46-47
一意な識別子の生成　45
業界標準の一意～　47
検証するための情報源　48-50
自己外部結合　431-432
時刻
開始時間／終了時間　727
自己結合　70, 432, 610
指数関数　220
自然外部結合　430-431
自然キー　197

自然結合　430
自然左外部結合　430
四則演算　212-213
ジップの法則　110
自動インクリメント列　42
　IDENTITY列　40-44
　IDENTITY列とシーケンス　41, 44
自動サマリテーブル　485
射影結合正規形　184
社会保障番号　286-288
週　730-733
集合演算　653
　EXCEPT　501, 658-663
　　NULLと重複行がない場合の～　661-662
　　NULLと重複行に対する～　662-663
　INTERSECT　658-663
　　NULLと重複行がない場合の～　661-662
　　NULLと重複行に対する～　662-663
　SELECT DISTINCT　656, 657, 661
　UNION　654-657
　UNION ALL　654-657
　集合の相等性　672, 675
集約関数　335, 370, 409
　～とともにCASE式を使用　335-336
　AVG関数　532

データがないグループを含む平均　534-536
複数列の平均　536-537
COUNT関数　528
カーディナリティのカウント　528
式のカウント　528, 530
DISTINCT付きの集約関数を最適化
530-531
LIST関数　546
クロステーブルとの併用　547-549
再帰共通表式と～　546-547
MAX関数　537
MIN関数　537
PRD関数　549
式による～　550-551
対数による～　551-553
SELECT句　414
SUM関数　531-532
外部結合と～　434
極値関数
～の一般化　539-545
簡単な～　537-539
複数の基準の～　545-546
単純な～　527
ビット単位
AND関数　555-556

OR関数　554-555
集約対同値性　131
集約の階層化
CASE式による～　521-522
グループ化したビューによる～　518-519
サブクエリによる～　519-521
主キー（PRIMARY KEY）　45, 71
～制約　71-73, 73t
～と表明　33-35
～のインデックス　748
述語
～で使うための関数　116-117
ALL ➡ALL述語
ANY　401, 402
BETWEEN ➡BETWEEN述語
CHECK制約で使用　382
DISTINCT　51, 131, 410
EXISTS ➡EXISTS述語
IN ➡IN述語
IS [NOT] NORMALIZED　328-329
IS [NOT]{TRUE | FALSE | UNKNOWN}　326-328, 327t
IS NULL ➡IS NULL述語
LIKE ➡LIKE述語
NOT EXISTS　390, 480, 767
NOT IN　377, 378, 379

OVERLAPS　363-372
SIMILAR TO　353-354, 353t, 354t
UNIQUE　404-405
量子化とサブクエリ　397
循環参照　74
順序関数　566
CUME_DIST　567-568, 592
DENSE_RANK　566-567
PERCENT_RANK　567-568, 592
RANK　566-567
ROW_NUMBER　566
物理的グループ　569
論理的グループ　569
上位中央値　578
条件付きの結合　439-447
安定的な結婚問題　441-444
在庫と注文　440-441
条件付き外部結合　430-431
ボールの箱詰め問題　444-447
乗算　212, 221
昇順アルゴリズム　448
小数を扱う関数　220-221
正味現在価値（NPV）　160, 161, 163
除算　212, 221
関係～　494-501

レムレーの～　501-505
序数（Ordinal numbers）　146, 260
シンクノード　705
人工的なキー　198
真部分集合　671-672

■ す

推移的関数従属　178, 179
数学関数　218-222
　　数論演算子　218-220
数学とNULL　270
数字の割り当て方　54-55
数値型　203-208
　　～の型変換　209-211
スーパーキー　78, 181
数列　55, 609-610
　　～テーブル　55-56, 145-151
　　～による分割　490-492
　　～の合計　626-629
　　～の歯抜け　623-626
　　オーバーラップする被覆　632-636
　　境界クエリ　617-618
　　シーケンス　141-142, 630-631
　　順序のランダムな～　59-61
　　素数　56-59
　　歯抜けの穴埋め　621-623
　　フィボナッチ～　115-116
　　ランを求めるクエリ　618-623
　　リージョン　610-611, 611-613, 614-617
　　リスト　146-148, 629-630, 631-632

数列テーブル　55-56, 145-151
　　循環数列にマッピング　148-149
　　リストの列挙　146-148
　　ループを置き換える　150-151
数論演算子　218-220
スカラSELECT式　437-438
スカラサブクエリ　308, 519
　　～の比較　398-399
　　IN述語と～　383-384
　　外部結合を書き換え　437
　　相関～　760
スカラ式　55, 383
スカラ値　4, 7
スキーマ　3, 99, 763-764
スキーマオブジェクト　25, 99
　　CASCADEオプション　763
　　RESTRICTオプション　763
　　X/Openトランザクションモデル　763
スキーマテーブル　99-100
スタースキーマ　558-559
スタンプ結合　105
スナップショットテーブル　485
スナップショット分離　20-22
スミステルルール　337

■ せ

正規化　170
　　実践的なヒント　195-196
正規形　167
　　関係モデルの～　167
　　基本キー～　180-181
　　第1～　171-176
　　第2～　176-178
　　第3～　178-180
　　第4～　183-184
　　第5～　184-186
　　ドメイン−キー～　186-195
制御的結合　105
整合性制約（経路列挙モデル）　686-687
整数 ➡INTEGER型生成された識別子　45-50
生成列　142
制約
　　CHECK～ ➡CHECK制約
　　NOT NULL～　67-69, 72
　　UNIQUE～　71-73, 75-88
　　一意～　71, 72, 73t
　　主キー～　71-73, 73t
　　隣接リストモデル（複雑な制約）
　　　　CHECK制約　679

循環（Cycle）　680, 680t
セッション　11-12
セルコの中央値　582-584
ゼロ除算　165, 205, 206, 214, 218
遷移表　125
線形再帰　482
線形内挿　164
宣言型言語　103, 123
宣言整合性制約（DRI）　7, 34, 127
全称量子化　397
千年に一度　741-742
　バグ ➡Y2K危機

■ そ

相関　608

相関サブクエリ　417-421

　　WHERE句　419

相乗平均　604

相等性　51, 672, 675

　　テーブル　672-675

　　部分集合　675-676

挿入異常　172

ソースノード　705

ソート

　　～を避ける　758-760

　　EXCEPT句　758

　　GROUP BY句　525-526

　　INTERSECT　758

　　SELECT DISTINCT　758

　　UNION　758

　　ウィンドウ句（ORDER BY句）　564

　　カーソル（ORDER BY句）　128-136

　　古典的な方法　760

　　相関スカラサブクエリ　760

　　曜日による～　732-733

ソートキー　129, 131, 564

属性分割　94-96

行レベル　96
テーブルレベル　94-96
素数　56-59
ソフトウェア開発の基礎　103

■ た

ダーティライト　15
ダーティリード　16
第1正規形（1NF）　171-176, 637
　繰り返し列　173-176
　文字列のリストをパース　175-176
第2正規形（2NF）　176-178
第3正規形（3NF）　178-180
第4正規形（4NF）　183-184
第5正規形（5NF）　184-186
耐久性（Durability）　15
ダイクストラのアルゴリズム　700
タイムスタンプ　230-232
タイムゾーン　228, 232, 233-234
多値従属性（multivalued dependency）　170-171
単一参照テーブル　157-159
単一列の範囲テーブル　488-489
単精度の数値　205
単調区間　609

■ ち

地域インターネットレジストリ　284
地域コード（米国）　288t
チェビシェフ多項式　114
逐次的凝集　104
中央値
　ウィンドウ関数を使った～　585-587
　セルコの～　582-584
　特性関数を使った～　580-582
　プログラミング問題としての～　578-580
重複行　50-52, 299
　ROWIDを使った削除　299-301
　テーブルから削除　299
重複度　51
調和平均　603-604
直列性　14

■ つ

通貨変換　285-286
通信的凝集　104
ツェラーの公式　730
月
　　開始日／終了日　727
ツリーインデックス　746
ツリー検索　483-485

■ て

データ
- 失われた更新　16
- ～が失われるのを防ぐ　274-275
- アクセス方法　746
- 失われたデータを追跡　274

データ型　66
- 型変換　318-320, 319t
- 時間型　228-234
- 数値型　203-208
- 文字列型　243-244

データ結合　105

データ構造結合　105

データ項目の制約違反（A5）　21

データテーブル対ジェネレータコード　113-114

データ表示フォーマット　319-320

データ分割
- ウィンドウによる分割　492-493
- クエリによる分割
  - FIFOとLIFOの部分集合　507-510
  - 関係除算　494-501
  - 被覆とパーティション　487-493
  - ブール式　505-507
  - レムレーの除算　501-505

数列による分割　490-492
単一列の範囲テーブル　488-489
範囲による分割　488
データベース
エンジン
インデックスとビュー　468
インライン展開　465-467
ビューの実体化　464-465
ビューの列リスト　464
ポインタ構造　467
スキーマオブジェクト
CASCADEオプション　763
RESTRICTオプション　763
X/Openトランザクションモデル　763
データベース管理者 ➡DBA
データ要素　279
テーブル　63
～からランダムな行を選択　666-670
～関数　143-144
～におけるN番目ごとの行　665-666
～の相等性　672-676
アクセス方法　746
失われた～　261
エンティティとしての～　4-5
カッター分類法　256-257

空のテーブルと失われたテーブル　261
関連としての～　5
グローバル定数～　165-166
参照補助～　151-159
時間データのモデリング　733-736
数列～　145-151
ファイルとの違い　3, 4
補助関数～　160-165
テーブルの操作　91, 291
ALTER TABLE文　93
DELETE FROM文　291
DROP TABLE文　92
INSERT INTO文　302
MERGE文　314
UPDATE文　305, 312
出次数　704
手続き型　103, 123
半手続き型への書き換え　112-116
手続き型のコード　112
隣接リストモデル　699
手続き的凝集　104
手続き的分解による解法　119-120
デッドロック　23
デューイ十進分類　156, 157
転置（行列）　647

■ と

ド・モルガンの法則　379
同一性　50
統計関数（SQL:2006）　607
　相関　608
　分散、標準偏差、記述統計　607-608
統計情報の更新　766
統計的中央値　577
同時実行制御　15-19
　5つの現象　15-17
　悲観的な～　19-20
　分離レベル　17-19, 18t
　楽観的～　20-22
　論理的な～　22-23
導出テーブル　478
　AS演算子　478
　FROM句　479
　VALUES構築子　480
導出列　142
到達可能ノード　703
同値性　5
　文字列の～　244-245
同値性テスト　245, 322, 350, 753
　IN述語　753

特性関数　580
　　～を使った中央値　580-582
　　レムレーの除算　503
　　ローゼンシュタインの～　339-340
独立性（Isolation）　14
トッドの除算　497-499
ドメイン　26
ドメイン−キー正規形（DKNF）　186-195
トランザクション　12
　　～分離レベル　17-19
　　ACID特性　12
　　SAVEPOINT　13
トリガー　101, 123, 124
　　～と参照整合性制約　126
　　遷移表　125
トリガーイベント　124

■ な

内挿

　　補助関数テーブルを使った～　163t, 164-165

内部結合（INNER JOIN）

　　EXISTSと～　390-391

　　標準的な構文　422-423

内部収益率（IRR）　160, 161-164

内部的結合　105

内部ノード　705-706

■ に

二次インデックス　748
日時の期間　233, 235t

■ ね

年月の期間　233, 235t

■ の

ノード　677, 698, 698f
　　入次数　704
　　孤立～　705
　　削除　686
　　シンク～　705
　　ソース～　705
　　出次数　704
　　到達可能～　703
　　内部～　705-706
　　隣接リストモデル　698, 698f

■ は

倍精度の数値　205
バイナリ形式での格納（IPアドレス）　223
配列
　　添え字列を使った～　641-642
　　第1正規形　637
　　テーブル形式の配列を計算　650-652
　　テーブルを配列に展開　649-650
　　名前を持った列による疑似～　638-641
ハッシュアルゴリズム　232
ハッシュインデックス　747-748
歯抜け
　　～の穴埋め　621-623
　　数列の～　623-626
範囲補助テーブル　155-156
反復不能読み取り　16, 21

■ ひ

ピアソンのr　605-606

　　相関　608

比較

　　NULLの～　264-265

　　行～　320-322

比較演算子　317

　　～の記号と意味　318t

　　型変換　318-320, 319t

悲観的な同時実行制御　19-20

非決定的な関数　108

非正規化　200-202

　　～数　205

左外部結合　425, 434

　　自然～　430

左中央値　578

日付　725

　　～関数

　　～の引き算　725-727

　　イベントと～　737-738

　　月の開始日／終了日　727

　　～フォーマット　230

　　ユリウス通日　728-729

　　レガシーデータにおける奇妙な日付　742

ビット単位の集約関数　553-556
ビットベクトルインデックス　748
被覆　487
　オーバーラップする～　632-636
　極小～　187, 192
ビュー
　～と一時テーブルの使い分け　476-478
　クエリにおける～　455-456
　更新可能および読み取り専用～　456-458
　削除
　　CASCADEオプション　474
　　DROP TABLE文　475
　定義　455
　データベースエンジン
　　インデックスと～　468
　　インライン展開　465-467
　　ビューの実体化　464-465
　　ビューの列リスト　464
　　ポインタ構造　467
　分類
　　UNIONを使ったビュー　460-462
　　入れ子のビュー　462-463
　　グループ化されたビュー　460
　　計算列　458-459
　　結合を使ったビュー　462

コード化された列　459
単一のテーブルからの射影および制限　458
評価述語　325
IS [NOT] NORMALIZED述語　328-329
IS NULL述語　325-326, 326t
表示フォーマット　319-320
標準偏差　587-588, 607-608
表明　8, 28, 88
制約　33-35

■ ふ

ファイ　115, 116

ファイル　3, 4

ファントム　16

フィールド　4, 7

　　列 VS ～　7-8

フィボナッチ数列　115-116

ブール式　505-507

　　NOT INを変換　380

フェッチ方向（Fetch orientation）　136-137

符号関数　220

物理ディスクのアドレス　40

物理的グループ　569

物理的ロケータ　3, 39

　　IDENTITY列　40-44

　　ROWID　40

　　物理ディスクのアドレス　40

　　露出した～　39-44, 198-200

部分木の削除　686

部分集合

　　～演算　665

　　FIFOとLIFOの～　507-510

　　相等性　672, 675-676

　　テーブルからランダムな行を選択　666-670

テーブルにおけるN番目ごとの行　665-666
包含演算子　670-676
プロシージャ　101
分割 ➡データ分割
分散　587-588
SQL:2006の統計関数　607-608
共～　604-605
分散コンピューティング環境　46
分離レベル　17-19, 18t
CURSOR STABILITY～　19

■ へ

平均　532, 534, 536
平均偏差　588
米国会計基準（GAAP）　285
ベンダー拡張
　～が持つ欠陥（UPDATE文）　312-313
　ウィンドウ関数　570-573
　文字列型　248-256

■ ほ

ボイス–コッド正規形（BCNF）　181-183
ポインタ構造　467
包含演算子　670
　真部分集合演算子　671-672
　テーブルの相等性　672-676
ボース–ネルソンソート　341
補助テーブル　145, 160-165
　～を使った逆関数　161-164
　Seriesテーブル　55, 109, 146
　階層～　156-157
　カレンダー～　736-738
　関数　160-165
　参照～　151-159
　数列テーブル　145-151
　単一参照テーブル（OTLT）　157-159
　単純な読み替えを行う～　153
　使うメリット　163
　内挿　163t, 164-165
　範囲～　155-156
　複数回の読み替えを行う～　153-154
　複数のパラメータを持つ～　154-155
ホスト言語　270-271

■ ま

マッピング
　　数列を循環数列に～　148-149
マテリアライズドクエリテーブル　485
マテリアライズドビュー　485
丸め　209-211

■ み

右外部結合　425, 429, 434
右中央値　578

■ む

無限大　205

■ め

メッセージ結合　105

■ も

文字集合 ➡キャラクタセット

文字列

- ～の同値性　244-245
- ～の中身を調べる　354-356
- ～のリストをパース　175-176
- インデックスを作る　357
- エンコーディング技術　357
- 検索と宣言　356-357
- 同値性テスト　245, 322, 350, 753
- 問題
  - グループ化　246
  - 順序　245
  - 同値性　244-245

文字列型　243-244

文字列関数　246-248

文字列置換　37, 57, 249

文字列連結　247

■ ゆ

ユーザーセッション　11
ユーザー定義関数　107, 108
有理数　289
ユリウス通日　728-729
ユリウス暦　226, 728

■ よ

様式番号　54
読み込み異常（A5A）　21
読み取り専用ビュー　456-458

■ ら

ライブロック　23
楽観的同時実行制御　20-22
ラン　618
　　～を求めるクエリ　618-623
乱数
　　～関数　667, 668
　　～発生モジュール　232
　　疑似～ ➡疑似乱数
乱数生成　666
　　～関数　668
ランダムな順序の数列　59-61

■ り

リージョン　610
　～に番号を割り振る　611-613
　大きさnのリージョンを見つける　610-611
　最も大きいリージョンを見つける　614-617
リーフ（ノード）　677
量子化　397
　～と失われたデータ　399-402
　～とサブクエリ述語　397
　EXISTSと～　391-392
　一意～　397
　全称～　397
隣接行列モデル　700-702
隣接リストモデル　483, 678
　SQLと～　699-700
　テーブルの更新　682-683
　手続き的な木の探索　681-682
　ノード　698, 698f
　複雑な制約
　　CHECK制約　679
　　循環（Cycle）　680, 680t
　隣接行列モデル　700-702

■ る

累積統計　589
　　五分位と関連の指標　598
　　順序関数　593-598
　　累積差分　590-591
　　累積率　591-593
ルート（ノート）　677
ループ　150-151
ルールベースのオプティマイザ　745

■ れ

レコード　3

　　行 VS ～　6

列（column）　4

　　～ VS フィールド　7-8

　　～制約　65-67

　　～における失われた値　261-263

　　CHAR(32)　285

　　SMALLINT～　284

　　VARCHAR(15)～　282

　　一列と複数列の一意性　82-88

　　繰り返し～　173-176

　　整数～　282-283

　　ソート　648

レムレーの除算　501

　　HAVING句　503

　　特性関数　503

連結テーブル　75

連番

　　欠番のある～　53-54

■ ろ

ローカル一時テーブル　89, 90
ローゼンシュタインの特性関数　339-340
ロケータ　39
　　露出した物理～　39-44, 198-200
論理値述語　269
論理的凝集　104
論理的グループ　569
論理的な同時実行制御　22-23
論理的分解による解法　120-121
論理とNULL　266-269

■ わ

和（UNION） ➡UNION

■ 著者プロフィール

ジョー・セルコ

著名なコンサルタント兼講師であり、SQL 関係の書籍では世界で最も読まれている著者の一人。ANSI SQL 標準化委員会での10 年に及ぶ貢献や、『Intelligent Enterprise』誌上のコラム（読者投票による賞をいくたびも受賞）、現実世界に対する洞察をSQLプログラミングに反映する取り組みでも有名である。

著作に『Joe Celko's SQL Trees and Hierarchies in SQL for Smarties』（Morgan Kaufmann、2004・2012）、『SQL Puzzles and Answers』（Morgan Kaufmann、1997・2006）［邦訳『SQLパズル第2版』（翔泳社、2007）］ほか。

■ 監訳者プロフィール

ミック

SI 企業に勤務するDB エンジニア。主にDWH/BI 分野のデータベース構築とパフォーマンスチューニングを専門としている。自身のサイト「リレーショナル・データベースの世界」でデータベースとSQLについての技術情報を公開している。

著書 『達人に学ぶSQL徹底指南書』（翔泳社、2008）
『SQL ゼロからはじめるデータベース操作』（翔泳社、2010）
『達人に学ぶDB設計徹底指南書』（翔泳社、2012）

訳書 ジョー・セルコ『SQLパズル 第2版』（翔泳社、2007）

DTP・編集協力：川月 現大（風工舎）

プログラマのためのSQL　第4版
すべてを知り尽くしたいあなたに

2013年 5月23日　初版 第1刷発行
2021年10月 5日　初版 第6刷発行

著　　者　ジョー・セルコ（Joe Celko）

監　　訳　ミック

発 行 人　佐々木 幹夫

発 行 所　株式会社 翔泳社 （https://www.shoeisha.co.jp）

＊ 本書は著作権法上の保護を受けています。本書の一部または全部について（ソフトウェアおよびプログラムを含む）、株式会社翔泳社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。

＊ 本電子書籍は同名出版物を底本として作成しました。記載内容は印刷出版当時のものです。印刷出版再現のため電子書籍としては不要な情報を含んでいる場合があります。印刷出版とは異なる表記・表現の場合があります。予めご了承ください。

ISBN 978-4-7981-4119-0
2022-10-17版