SQLパズル

第2版

プログラミングが変わる書き方／考え方

SQL Puzzles and Answers 2nd Edition

[著]＝ジョー・セルコ
[訳]＝ミック

SE
SHOEISHA

SQLパズル 第2版

プログラミングが変わる書き方／考え方

SQL Puzzles and Answers 2nd Edition

著] = ジョー・セルコ
訳] = ミック

SE
SHOEISHA

SQLパズル 第2版

[著]＝ジョー・セルコ
[訳]＝ミック

プログラミングが変わる書き方／考え方

SQL Puzzles and Answers 2nd Edition

SE
SHOEISHA

翔泳社 ecoProject のご案内

株式会社 翔泳社では地球にやさしい本づくりを目指します。
制作工程において以下の基準を定め、このうち4項目以上を満たしたものをエコロジー製品と位置づけ、シンボルマークをつけています。

資材	基準	期待される効果	本書採用
装丁用紙	無塩素漂白パルプ使用紙 あるいは 再生循環資源を利用した紙	有毒な有機塩素化合物発生の軽減（無塩素漂白パルプ） 資源の再生循環促進（再生循環資源紙）	○
本文用紙	材料の一部に無塩素漂白パルプ あるいは 古紙を利用	有毒な有機塩素化合物発生の軽減（無塩素漂白パルプ） ごみ減量・資源の有効活用（再生紙）	○
製版	CTP（フィルムを介さずデータから直接プレートを作製する方法）	枯渇資源（原油）の保護、産業廃棄物排出量の減少	
印刷インキ*	大豆インキ（大豆油を20%以上含んだインキ）	枯渇資源（原油）の保護、生産可能な農業資源の有効利用	○
製本メルト	難細裂化ホットメルト	細裂化しないために再生紙生産時に不純物としての回収が容易	○
装丁加工	植物性樹脂フィルムを使用した加工 あるいは フィルム無使用加工	枯渇資源（原油）の保護、生産可能な農業資源の有効利用	

*：パール、メタリック、蛍光インキを除く

本書内容に関するお問い合わせについて

本書に関するご質問、正誤表については、下記のWebサイトをご参照ください。

正誤表　http://www.seshop.com/book/errata/
出版物Q&A　http://www.seshop.com/book/qa/

インターネットをご利用でない場合は、FAXまたは郵便で、下記にお問い合わせください。

〒160-0006　東京都新宿区舟町5
（株）翔泳社 編集部読者サポート係
FAX番号：03-5362-3818

電話でのご質問は、お受けしておりません。

JOE CELKO'S SQL PUZZLES AND ANSWERS
by Joe Celko
ISBN-10:0-12-373596-3 ISBN-13:978-0-12-373596-6

まえがき

1990年代中ごろのことだが、私は『Database Programming & Design』誌と、その後に『DBMS』誌でコラムを連載していた。そのとき、読者を惹きつけるテクニックとして使っていたのが、コラムの最後にSQLのパズルを載せることだった。10年後、2つの雑誌は統合されて、『Intelligent Enterprise』誌になった。そして、私のSQLパズルは少し小さな媒体に移され、やがて姿を消した。最近では紙媒体よりも、むしろインターネット上（www.dbazine.comなど）に少しずつパズルを載せている。

もう何年もの間、大学生の間ではあらゆる種類の手続き型言語 —— 昔はCやPascal、今ならJavaやC++ —— を使うプログラミングのコンテストが行われている。それにひきかえ、DBプログラマには、私の小さなパズル本を除いて腕試しの機会はあまりない。

私のパズルが、学生たちの宿題に使われているのもよく見かけた。教師たちにしても、私の本はSQLの練習問題が得られる唯一の情報源だったのである。宿題のネタ元が私だと全く知らない怠惰な学生から、「宿題を解いてくれ」という電子メールを受け取ったこともあった。

その懐かしき時代には、SQLのデファクトスタンダードはSQL-86で、SQL-92はDBベンダにとってまだ設計目標だった。今日ではほとんどのベンダが、SQL-92の大半の機能を自社のRDBMS製品に取り入れている。設計目標は今やSQL-99のOLAP機能になった。

10年前、大学生がRDBMSのコースを専攻しSQLプログラマになるには、いくらか専門的な知識を要求された。RDBMS製品は高価で、最高のものはメインフレーム上で動いていたからだ。

今日、学士課程でRDBMSの理論を教えている大学はない。SQLはかつてほど新奇な言語ではなくなり、安上がりなオープンソースのRDBMSも普及した。特定の製品についての助言を得られるニュースグループもインターネット上多く存在している。

一方で、悪いニュースもある。SQLプログラマの質が以前より落ちてしまったのだ。RDBMSの基礎知識を持たずSQLの訓練を受けたこともない彼らに、普段使用しているプログラミング言語の中にSQL文を書くよう求めていることが原因である。

本書で集めたパズルには、初版に入っていたオリジナルの問題も含まれている。だから、初版を読んでくれた読者も、お気に入りの問題を見つけられるだろう。だが、そうした既存の問題にも、いくつか新しい解法を追加している。新しい解法には古い構文の

解もあれば、新機能を使った解もある。初版刊行以後の長い歳月の間に、さまざまな人がオリジナルの解法の多くを“料理”してくれた。ここで言う“料理”とは、問題を出した人間の解よりも優れた解を与えることを指す、パズラー用語である。初版は50問だったが、この版では75問収録した。

初版を書いたとき、私は作成順や難易度ではなくカテゴリ別でパズルを分類しようとした。だが、それは無益なことだと悟って、この版ではカテゴリ別の分類をあきらめた。あるパズルがDDL（データ定義言語）を変更することでも、クエリを変更することでも解けるとして、それをDDLのパズルとDML（データ操作言語）のパズルのどちらに分類すればよいというのだろう？

各パズルに関係する人々はクレジットするように努めたが、もし見落とした人がいたらここでお詫びする。

謝辞、間違いの修正および将来の版について

間違いの指摘や、新しいトリックやテクニック、および本書の将来の版に活かせる情報は大歓迎である。出版社のモーガン・カウフマン（Morgan Kaufmann）を通じて、皆さんの考えを送ってほしい。

以下の人々に感謝したい。モーガン・カウフマン社のダイアン・セーラ、『DBMS』誌のデヴィッド・カルマン、『Database Programming & Design』誌のデヴィッド・ストッダー、ミラー・フリーマン社のフィル・チャプニック、『Boxes & Arrows』のフランク・スウィート、www.dbazine.comのダナ・ファーヴァー。

私のパズルをたくさん“料理”してくれたスミス・バーニー社のリチャード・レムレー、何年にもわたって電子メールを送ってくれたCompuServeとSQLニュースグループのすべての人々、および今日ニュースグループに投稿している人々には、特に厚く感謝したい（ニュースグループで皆さんが使っているハンドル名を載せたので、ご自分の投稿が見つかるかもしれない）。全員ではないが、何人かを以下に挙げる。レイモンド・アンジュー、ディーター・ネース、アレクサンダー・クズネツォーフ、アンドレイ・オデゴフ、スティーブ・カシュ、ティボー・カラスチ、デヴィッド・ポータス、フーゴ・コーネリアス、アーロン・バートランド、イツィク・ベン-ガン、トム・モロー、サージ・リーラウ、エーランド・ソマルスコク、ミキト・ハラキリ、アダム・マカニック、ダニエル・A・モルガン。

訳者まえがき

本書は、米国データベース界の重鎮の1人であるジョー・セルコが著した有名なパズルブックである。恐らく世界で最も広く読まれているSQL問題集の1冊と言ってよいだろうが、初版の邦訳が出版されたのが1997年だから、日本ではほぼ10年ぶり、久々の復活ということになる。初版の刊行以来、SQLとデータベースの機能も目覚しい拡張を遂げ、SQL:1999とSQL:2003という2度の標準SQLの改訂も行われた。本書にも、OLAP関数や共通表式に代表されるこれら新機能を積極的に取り入れた解法が多く追加されている。

本書の第1の魅力は、何と言っても具体的なケーススタディを通じて、実践的なSQLのテクニックを知り、その考え方を身に付けられることだろう。行列変換や結果のフォーマッティング、複雑な条件を用いる集約や結合、集合演算の応用など、本書に収められた事例は実に多彩で、そのほとんどが、世界中のDBエンジニアたちが日々の業務で直面した問題をWebに投稿したり、セルコのところへ持ち込んで助言を求めたものばかりである。そして、それに対するセルコら熟練したエンジニアたちから寄せられた解決策も――よくもまあ、これだけいろいろ考えつくもの、と少し呆れてしまうぐらい――多彩で創造的だ。そう、本書の第2の魅力は、SQLという言語の非常に豊かな可能性を発見できることである。きっと、皆さんも本書を読み進めるうちに「そんな考え方があるのか……」とか「そんなことがSQLでできるのか……」というつぶやきを、何度となく口にすることになるはずだ。それはまた、訳者自身が持った感想でもある。「煌くようなアイデアの宝庫」とはこのような本を指して呼ぶに違いない、と読み返すたびに思ったものだった。

ところで、冒頭でこの本は問題集であると述べたが、これには2つの含意がある。1つは「対になる教科書の存在」だ。これに相当するのが、セルコのもう1冊のベストセラー『プログラマのためのSQL 第2版』(ピアソン・エデュケーション刊, 2001) である。本書とも扱っている問題が一部重複しているうえ、本書がとにかくたくさんの問題と解答を詰め込もうとして、ややもすると解説不足に陥る傾向があるのに対し、非常に詳細かつ網羅的な解説を得ることができる。併読すると読者の理解を助けてくれるに違いない。

もう1つの意味は、「この本は決して『クックブック』ではない」ということだ。パズルという親しみやすい形式をとっているので勘違いしやすいのだが、この本をクックブックだと思って読むと頭がクラクラするだけで終わってしまうだろう。確かに、非常に豊

富な実例を含む本だから、自分が今まさに直面していることと類似の問題を見つけられることもあるだろうし、コードをコピーすることで急場をしのげることもあるだろう。それはそれでこの本の効用の1つと言える。しかし、そうした目的のためにはもっと適した書籍が多くあるし、本書をそれだけのために使うのは、いかにももったいない。

本書は、その内容を理解するために一定の努力を読者に強いてくるため、多くの読者にとって、寝転んでスラスラ読み進められるタイプの本ではないと思う。何度も立ち止まり、実際に手を動かしてコードを書き、結果を確認し、自分でSQL文をカスタマイズして、また実行してようやく1つ納得する——そういう地道なプロセスの繰り返しを抜きにして、本書を理解することは難しい（これは一部には、極めて多くのことを凝縮したロジックで実現するSQLの性質にもよるのだが）。少し格好をつけて言えば、本書は「読者の代わりに考えるのではなく、読者を自ら考えるように仕向ける」本だということだ。名著とは、きっと多かれ少なかれそういう性質を備えているものだろう。

これとよく似た意味において、本書は「パズルのためのパズル」でもない。中には「パズル66：数独パズル」や「パズル75：もう1軒行こう」のように、純然たるパズルから問題を借りてきているものもあるが、それらはいわば一種の“つかみ”のようなもので、読者をそうしたパズルの達人にすることが目的というわけではない。セルコの狙いは、あくまで読者にSQLという言語の本質を理解してもらうことにある。

少し話が長くなったが、ともあれ、本書が大変に面白く、かつ有用であることは疑いを入れない。本書がより多くのDBエンジニアの方に読まれ、セルコの知見が日本のデータベース界にも浸透していくことを訳者も切に願っている。そのことに幾ばくかの貢献ができたのなら、訳者としてこれに勝る喜びはない。

2007年9月9日
訳者

Contents

本書の読み方と注意事項

- 本書を読むにあたっては、SQLの基本的な文法や機能について一通り理解していることが望ましい。また、ケーススタディの体裁をとっており、どこから読み始めてかまわない。ただし、クックブックではないので、解答のロジックを理解できるまでじっくり考えながら読むことが重要。別解を考えてみるのも効果的である。
- 翻訳に際し、各パズルのサブタイトル（解答で使用されているテクニックや技術的テーマなどを表示）と、テーブル名や列名の日本語訳を付け加えた。
- 本書のSQL文は、基本的に標準SQL（SQL-89/92/99）にのっとって記述されている。ただし、RDBMS（リレーショナルデータベース管理システム）によっては動作しないSQL文や、書き換えが必要なSQL文がある点は、あらかじめご了承いただきたい。
- RDBMSにより動作しない可能性のある構文の例を、以下に挙げておく。例えば、Oracleではテーブルの相関名を定義するのに「AS」キーワードを使用できない（削除する）。いくつかのRDBMSでは、1文で複数行を挿入するINSERT文がエラーになる（1文につき1行挿入に直す）。また、テーブル列のデータ型も変更が必要な場合がある。

【動作しない可能性のある構文の例】

□テーブル列のデータ型　□共通表式　□OLAP関数
□EXCEPT/INTERSECT演算子　□1文で複数行を挿入するINSERT文
□INTERVAL定数の書式　□テーブルの相関名を定義する「AS」キーワード

なお、SQL文中にあるコロンを付けた識別子（:user_id）はパラメータを意味する。

- 本書に掲載されているSQL文のテキストデータは、下記の翔泳社Webサイトからダウンロードできる。また、訳者が運営する下記のWebサイトで、各パズルのサンプルデータを作成するCREATE TABLE文およびINSERT文を提供するとともに、より詳細なSQL文の解説や別解の紹介を行っている。本書の学習にお役立ていただきたい。

翔泳社「サンプルデータ ダウンロード」
http://www.seshop.com/book/download/

訳者「SQLパズル 第2版のサポートページ」
http://www.geocities.jp/mickindex/database/db_support_sqlpuzzle.html

パ|ズ|ル|
1

会計年度テーブル

範囲外の日付を入力しないための制約

さっそくだが、「できる限り完ぺきなCREATE TABLE文」の作成にチャレンジしてみよう。ここで行うちょっとした訓練は、SQLの記述力を高める上で大切なものだ。というのも、SQLは宣言的言語なので、物事をコードではなくデータベース定義によって規定する技術を学ぶ必要があるからだ。

訓練に使うテーブルは、次のような形をしている。

```
CREATE TABLE FiscalYearTable1
(fiscal_year INTEGER,
 start_date  DATE,
 end_date    DATE);
```

FiscalYearTable1：会計年度テーブル	fiscal_year：会計年度
start_date：年度開始日	end_date：年度終了日

このFiscalYearTable1テーブルは、各会計年度がいつから始まりいつで終わるのか、その日付の範囲を格納する。任意の日付を条件にこのテーブルへ問い合わせれば、その日がどの会計年度に属するのかが分かるわけだ。ここでは、米国政府の会計年度（10月1日～9月30日）を例に使うことにしよう[訳注1]。ある日付の会計年度を求めるサブクエリは、次のように書ける。

```
(SELECT f1.fiscal_year
   FROM FiscalYearTable1 AS F1
  WHERE <任意の日付データ> BETWEEN F1.start_date AND F1.end_date)
```

さて、皆さんにはこのテーブルに対し、正しい情報だけを持つように制約をもれなく設定してもらいたい。

RDBMSによって使用できる日付／時間関数は異なるが、ここではSQL-92規格で定義されている時間計算と、EXTRACT([YEAR | MONTH | DAY] FROM <データ式>)関数だけを使えるものとしよう。EXTRACT関数は、第2引数に指定した日付データをもとに、第1引数で指定したフィールド（年・月・日のいずれか）の値を整数で返してくれる。

訳注1：米国の会計年度は、終了月の年が適用される。ここでは9月30日時点の年が使われる。

あなたは完ぺきなテーブル定義にどこまで近づけるだろうか。

その1

答え

1. 何はなくとも、まず全列にNOT NULL制約を付けよう。NULLを許可するしかるべき理由はないのだから。

2. たいていのSQLプログラマは、すぐに主キーを設定することを考える。ここでは、主キーを(fiscal_year, start_date, end_date)と定義しようと思ったかもしれない。けれども会計年度というのは、実質的に会計年度期間(start_date, end_date)に付けられた"別名"なので、主キーはfiscal_yearだけで足りる。また、どの列についても重複する年月日があっては困るので、UNIQUE(start_date)、UNIQUE(end_date)の各制約を設定するのもよい。

3. あまりに当たり前すぎて、ほとんどの人が設定を忘れてしまうのが、CHECK(start_date < end_date)、またはCHECK(start_date <= end_date)という制約である。これも正解だ。

4. しかし、制約はこれでもまだ不十分だ。例えば、次のようなエラーは防げない。

```
(1995, '1994-10-01', '1995-09-30')
(1996, '1995-10-01', '1996-08-30')     ←エラー！ (年度終了日が8月)
(1997, '1996-10-01', '1997-09-30')
(1998, '1997-10-01', '1998-09-30')
```

そこで、年度開始日と年度終了日がどの年度も10月1日(年次は1年前)と9月30日になるよう、start_date列とend_date列には次のように制約を設定しよう。

```
CREATE TABLE FiscalYearTable1
(fiscal_year INTEGER NOT NULL PRIMARY KEY,
 start_date  DATE NOT NULL,
 CONSTRAINT valid_start_date
   CHECK ((EXTRACT(YEAR FROM start_date) = fiscal_year - 1)
          AND (EXTRACT(MONTH FROM start_date) = 10)
          AND (EXTRACT(DAY FROM start_date) = 01)),
 end_date DATE NOT NULL,
 CONSTRAINT valid_end_date
   CHECK ((EXTRACT(YEAR FROM end_date) = fiscal_year)
```

```
        AND (EXTRACT(MONTH FROM end_date) = 09)
        AND (EXTRACT(DAY FROM end_date) = 30)));
```

あるいは、ANDで結ばれている各述語[訳注2]を個別の制約に切り分けて、エラーメッセージを細かく出力させることもできる。年度開始日と年度終了日の「年」の部分も一意な会計年度から導かれるものだから、やはり一意性が保証される。

5. 残念なことに、ここまでに述べてきた方法はあらゆる企業で通用するとは限らない。多くの企業は週、週末、平日の数え方に関して、複雑な規則を持っているからだ。具体的には、1つの会計年度がぴったり360日または52週間となるように定めている。事実、第4四半期を「4週・4週・5週」とし、年度の終わりをちょっぴりごまかすことが、会計の一般的な慣習として行われている。その結果、年度末では3日から11日ほど余ってしまうのである。

この問題は、FiscalYearTable1テーブルと同じ要領で「会計月テーブル」を作ることで解決できる。また、こういうケースでは、

```
CHECK ((end_date - start_date) = INTERVAL '359' DAY)
```

という制約が驚くほどうまくはまる[訳注3]。359という数は、各企業の規則に合わせて適当に変えてほしい（例えば、52週×7日＝364日というように）。もし1年度の総日数を多少動かせるようにしたいなら、等号の代わりにBETWEEN述語を使おう。

さて、最後に1つだけ白状しておこう。こういうテーブルをデータベースに作らなければならなくなったとき、私は表計算ソフトとその時間関数を利用することにしている。最近の表計算ソフトは、データベースよりずっと便利な時間関数を持っているからね。それに、会計部門がすでに表計算ソフトで会計カレンダーを作っている可能性だって、十分あり得ると思う。

訳注2：真理値（True、False、Unknown）を返す関数のこと。=、>、< やBETWEEN、LIKE、IN、IS NULL などはみな述語の仲間。

訳注3：INTERVAL定数の記述は、標準SQLにおいては、「INTERVAL + シングルクォーテーションで囲んだ数値 + 日時フィールド（DAY、MONTH、YEARなど）」と定められている。しかし、実際にはRDBMS製品ごとに若干の表記揺れがあるため、マニュアルを確認したほうがよいだろう。

パ|ズ|ル|
2

欠勤

条件付きのUPDATE/DELETE

この問題は、ジム・チャペラがCompuServe[注1]のMicrosoft Accessフォーラムに投稿したものだ。彼は、社員の欠勤状況を記録するためのデータベースを作ろうとしていた。彼が考えたテーブルはこんな構造をしている。

```
CREATE TABLE Absenteeism
(emp_id INTEGER NOT NULL REFERENCES Personnel (emp_id),
 absent_date DATE NOT NULL,
 reason_code CHAR(40) NOT NULL
             REFERENCES ExcuseList (reason_code),
 severity_points INTEGER NOT NULL
             CHECK (severity_points BETWEEN 1 AND 4),
 PRIMARY KEY (emp_id, absent_date));
```

Absenteeism：欠勤テーブル	emp_id：社員ID	absent_date：欠勤日
reason_code：理由コード	severity_points：罰点	Personnel：社員テーブル
ExcuseList：言い訳テーブル		

社員はemp_idで一意に識別できる。reason_codeは欠勤理由の短い説明である（「ビールを積んだトラックにはねられた」とか「気が乗らない」とか）。こうした理由、つまり言い訳はどんどん増えるし、実に想像力豊かでもある。severity_pointsは、欠勤に対して与えるペナルティを罰点として表したものだ。

社員は、罰点を年間40ポイントためるともれなくクビになる（Personnelテーブルから該当する社員のレコードを削除する）。これを第1のルールとする。ただし、2日以上連続して休んだ場合は「長期病欠」扱いとなり、2日目以降の欠勤には罰点が付かないし、欠勤の総日数にもカウントされない。これを第2のルールとする。

あなたの任務は、これら2つのビジネスルールを実現するSQL文を書くことだ。もし必要なら、スキーマを変更してもよい。

訳注1：米国の大手パソコン通信サービス。かつては全米最大だったが、1995年に加入者数でAOLに抜かれ、1997年同社に買収された。

その1

答え　第2のルールに関する最も犯しがちな間違いは、テーブルから2日目以降の欠勤日を削除することだ。このアプローチをとってしまうと、病欠日を数えるクエリが難しくなり、連続する病欠日を見つけ出すのもかなり大変になる。

ここでの“トリック”は、severity_points列に0点が入るのを許すことである。そうすれば、Absenteeismテーブルから社員が長期病欠している期間を見つけ出せる。severity_points列の制約を、CHECK(severity_points BETWEEN 0 AND 4)と変えればOKだ。

一見すると、ゼロを格納するのは領域の無駄遣いに思えるので、特に新米エンジニアはこのトリックを見落としがちだ。しかし、ゼロというのも立派な数であり、病欠という出来事は記録する必要のある事実だから、軽々しく削除してはいけない。

Absenteeismテーブルに新しい行が挿入されたときは、次のUPDATE文を実行する。

```
UPDATE Absenteeism
   SET severity_points= 0,
       reason_code = 'long term illness'
 WHERE EXISTS
   (SELECT *
      FROM Absenteeism AS A2
     WHERE Absenteeism.emp_id = A2.emp_id
       AND Absenteeism.absent_date =
           (A2.absent_date + INTERVAL '1' DAY));
```

このUPDATE文は前日に欠勤していないかを調べて、もし欠勤していれば第2のルールに従い、挿入された行の罰点を0点にし、理由を“長期病欠”に変更する。

一方、社員を解雇するという第1のルールを実現するには、まず社員の現在の点数を知る必要がある。これは次のようなクエリで十分だろう。

```
SELECT emp_id, SUM(severity_points)
  FROM Absenteeism
 GROUP BY emp_id;
```

このクエリが、最終的に求めるべき（つまり、Personnelテーブルから該当社員のレコードを削除する）DELETE文のサブクエリのもととなる。DELETE文の最終形は次のとおりだ。

```
DELETE FROM Personnel
  WHERE emp_id =
    (SELECT A1.emp_id
       FROM Absenteeism AS A1
      WHERE A1.emp_id = Personnel.emp_id
      GROUP BY A1.emp_id
     HAVING SUM(severity_points) >= 40);
```

このDELETE文のサブクエリは、severity_pointsの合計が40点未満の社員についてはNULLを返すので、Personnel.emp_id列との比較に失敗する。したがって、そのレコードは削除されず、社員当人もクビにならない。

その2

答え

オラクル社で上級インストラクターを務めるバート・スカルツォは、答えその1の解には欠点が2つあり、パフォーマンスの面でも改善の余地があることを指摘した。

欠点はごく単純なものだ。まず、先ほどのDELETE文のサブクエリはルールに反して、社員が1年間という期間内で罰点を40点以上ためたかどうかのチェックを行っていない。罰点の集計期間を過去1年間に限定するための条件を、WHERE句に加える必要がある。

```
DELETE FROM Personnel
 WHERE emp_id =
   (SELECT A1.emp_id
      FROM Absenteeism AS A1
     WHERE A1.emp_id = Personnel.emp_id
       AND absent_date
           BETWEEN CURRENT_TIMESTAMP - INTERVAL '365' DAY
               AND CURRENT_TIMESTAMP
     GROUP BY A1.emp_id
    HAVING SUM(severity_points) >= 40);
```

また、このDELETE文で削除されるのは社員だけで、Absenteeismテーブルにある欠勤の履歴はそのまま残ってしまう。明示的にせよ暗黙的にせよ、これも削除する必要がある。先のDELETE文を再利用してもよいが、Absenteeismテーブルの制約にカスケード削除のオプション（ON DELETE CASCADE）を追加するのがベストだ。

```
CREATE TABLE Absenteeism
(emp_id INTEGER NOT NULL
        REFERENCES Personnel (emp_id)
        ON DELETE CASCADE,
 ...);
```

さらに、もし第2のルールを実現するUPDATE文が定期的に実行され、かつ欠勤中の社員に部署の異動がないことが確実だとしたら、そのサブクエリを次のように書き直すことでパフォーマンスを改善できるという。

```
UPDATE Absenteeism AS A1
   SET severity_points = 0,
       reason_code = 'long term illness'
 WHERE EXISTS
   (SELECT *
      FROM Absenteeism AS A2
     WHERE A1.emp_id = A2.emp_id
       AND (A2.absent_date + INTERVAL '1' DAY) = A1.absent_date);
```

さて、パフォーマンス面での改良も施したSQL文だが、まだ週をまたぐ長期病欠の処理に問題が残っている。

「週末の休みを病気療養に充てたい」という社員がいたら、それは会社として大変結構なことだろう。しかし、先のUPDATE文のままでは、その気持ちが無駄になってしまう。というのも、ある社員が第1週の金曜日、第2週の全部、および第3週の月曜日を欠勤日として報告したときに、このUPDATE文では第2週の5日間しか長期病欠期間として認めないからだ。第1週の金曜日と第3週の月曜日は、罰点付きの欠勤日として扱われる。この見逃された連続期間まで正確に把握するには、UPDATE文のサブクエリを変えなければならない。

私なら、週末に関するこの問題を休日のための理由コード（週末、祝日、バカンス休暇など）を用意することで回避する。そうした日の罰点は、もちろんゼロである。週末も出勤するような職種では、こういうコードが必要だろう。あとは、上司に土曜日と日曜日の理由コードを“週末”から“長期病欠”に手で変えてもらうことさえできれば、先のUPDATE文はそのまま使用できる。

同じトリックを使えば、クルージングに出かける直前に運悪く伝染病にかかった場合にも、計画していたバカンスをあきらめなくて済むかもしれない。上司が真に温情ある人物なら、失った週末の埋め合わせに罰点0の欠勤日をAbsenteeismテーブルに追加し

てくれたり、欠勤日を未来の日付に書き換えたりして、改めてバカンスへ出かけられるよう再調整してくれるだろう。

1年以上経ち“時効”になった欠勤データについては、放っておいてかまわないと思う。ただ、テーブルのサイズをできるだけ小さく保つために、1年以上前の記録（行）をAbsenteeismテーブルから削除するDELETE文を組み込むのもよいかもしれない。

また、今日の日付から1年前までの期間を取得する、

```
(BETWEEN CURRENT_TIMESTAMP - INTERVAL '365' DAY
     AND CURRENT_TIMESTAMP)
```

という式は、

```
(BETWEEN CURRENT_TIMESTAMP - INTERVAL '1' YEAR
     AND CURRENT_TIMESTAMP)
```

という式で置き換えることができる。こうすると、うるう年も扱える。さらに優れた方法は、PostgreSQLなどのRDBMSが持っているAGE関数を使うことだ。AGE関数は、引数の日付に起きた出来事からの経過年数を返してくれる。これを使えば、先のBETWEEN述語を(AGE(absent_date) >= 1)と書くことができる。

その3

答え

この種の問題に対する便利なもう1つの道具は、カレンダー（Calendar）テーブルである。これは社員にはありがたくない、出勤日を記録するテーブルだ。本書の初版が世に出てからの10年間で、この方法はSQLプログラミングにおける定石の1つになった。

```
SELECT A.emp_id,
       SUM(A.severity_points) AS absentism_score
  FROM Absenteeism AS A, Calendar AS C
 WHERE C.cal_date = A.absent_date
   AND A.absent_date
       BETWEEN CURRENT_TIMESTAMP - INTERVAL '365' DAY
           AND CURRENT_TIMESTAMP
   AND C.date_type = 'work'
 GROUP BY emp_id
HAVING SUM(severity_points) >= 40;
```

cal_date：カレンダーテーブルの日付列　　cal_type：「平日」「週末」「祝日」といった日の種類

中には、平日をユリウス通日[訳注2]で記録するための列（Julian_workday）をカレンダーテーブルに追加する人もいる。その場合、祝日と週末は直前の平日と同じ番号を持つことになる。(cal_date, Julian_workday) の組み合わせは、例えばこんな感じだ。

```
('2006-04-21', 42) …… 金曜日
('2006-04-22', 42) …… 土曜日
('2006-04-23', 42) …… 日曜日
('2006-04-24', 43) …… 月曜日
```

今日からさかのぼって1年前の日を知るには、今日のユリウス通日から計算すれば簡単である。

訳注2：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暦法。数年をまたぐ2つの日付間の日数を計算するのに便利。

パ|ズ|ル|
3

忙しい麻酔医

重複する期間の抽出（その1）

ずいぶん前のことだが、レオナルド・C・メダルは、麻酔医を題材にした面白い小問を思いついた。病院の麻酔医たちは、院内で行われる外科手術中に、患者に麻酔を施して回っている。彼らが行った処置の記録は、次のようなテーブルに保管されている。

```
Procs

proc_id   anest_name   start_time   end_time
=========================================
     10   'Baker'      '08:00'      '11:00'
     20   'Baker'      '09:00'      '13:00'
     30   'Dow'        '09:00'      '15:30'
     40   'Dow'        '08:00'      '13:30'
     50   'Dow'        '10:00'      '11:30'
     60   'Dow'        '12:30'      '13:30'
     70   'Dow'        '13:30'      '14:30'
     80   'Dow'        '18:00'      '19:00'
```

Procs：処置テーブル	proc_id：処置ID	anest_name：麻酔医の名前
start_time：開始時刻	end_time：終了時刻	

よく見ると、麻酔医の処置時間のうち、いくつかが重複しているが、これは間違いではない。外科医と違って、麻酔医は1つの手術に付きっきりになる必要はない。手術中にほかの手術室へ出向き、「掛け持ち」で複数の患者を順番に診て回ることができる。そうして患者の投薬量を調節したら、あとは若手の医者や看護士に、分単位で患者の容態を監視するよう命じておくわけだ。

麻酔医への報酬は処置ごとに決まるのだが、実はそこにやっかいなルールがある。報酬の金額は、麻酔医が同時に掛け持ちした処置の最大数（「最大瞬間掛け持ち数」と呼ぼう）に応じて決まる。そして、その数が多くなればなるほど、1つの処置に対して支払われる報酬は低くなっていくのである。

例えば、Baker医師が担当した10番（proc_id = 10）の処置では、最大瞬間掛け持ち数は2である。その場合、10番の処置に対して同医師が得られる報酬は、本来支払われる金額の75％になるのだという。

さて、ここで問題とするのは金額ではなく、「各処置における最大瞬間掛け持ち数をどう求めるか」である。

手始めにグラフを使って答えを探ってみると、問題をより明確に理解できる。

図1は、Baker医師が並行して進めた2つの麻酔処置（10番と20番）を表したグラフである。図上部のガントチャートに似たグラフは、2つの処置が行われた時間帯を示している。

一方、図下部のステップチャートは、瞬間掛け持ち数を表している。図上部のガントチャートから30分区切りで重複数、つまり掛け持ち数を求め、それを棒グラフにしたものだと考えると分かりやすいだろう。

このグラフからは、Baker医師が行った10番の処置についての最大瞬間掛け持ち数は2であることが分かる。

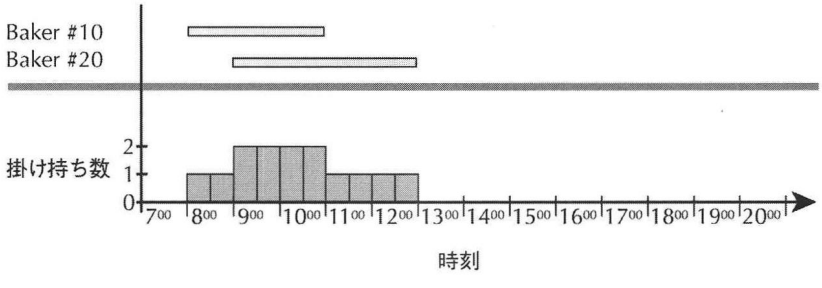


図1：Baker医師の処置記録

図2は、処置どうしの重複がより複雑になっているが、見方は図1と同じだ。Dow医師が行った30番の処置についての最大掛け持ち数は3で、それが2回生じている。

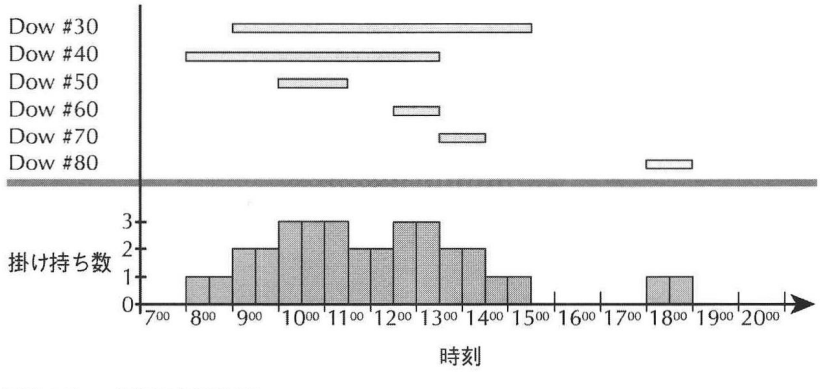


図2：Dow医師の処置記録

再度確認しておくと、求めたい答えは処置の重複回数ではなく、その最大瞬間掛け持ち数である。では、Procテーブルに記録されたデータから、どうしたらSQLで各処置の最大瞬間掛け持ち数を求められるだろう。以下に、求めたい結果を示す。

```
proc_id   max_inst_count
========================
     10                2
     20                2
     30                3
     40                3
     50                3
     60                3
     70                2
     80                1
```

proc_id：処置ID　　max_inst_count：最大瞬間掛け持ち数

その1

答え　まず最初にすることは、各処置を2つのイベント――すなわち処置の開始と終了――に変換して、それをビューに入れることだ。

```
CREATE VIEW Events
(proc_id, comparison_proc, anest_name, event_time, event_type) AS
SELECT P1.proc_id, P2.proc_id, P1.anest_name, P2.start_time, +1
  FROM Procs AS P1, Procs AS P2
 WHERE P1.anest_name = P2.anest_name
   AND NOT (P2.end_time <= P1.start_time
            OR P2.start_time >= P1.end_time)
UNION
SELECT P1.proc_id, P2.proc_id, P1.anest_name, P2.end_time,
       -1 AS event_type
  FROM Procs AS P1, Procs AS P2
 WHERE P1.anest_name = P2.anest_name
   AND NOT (P2.end_time <= P1.start_time
            OR P2.start_time >= P1.end_time);
```

Events：イベントビュー	proc_id：評価対象の処置番号
comparison_proc：重複している処置の番号	anest_name：麻酔医の名前
event_time：処置の開始／終了時刻	event_type：イベントの種類

UNION演算子により、開始イベントの集合に終了イベントの集合をマージしている。event_typeの＋1は開始イベント、－1は終了イベントを表す。また、WHERE句は、比較する処置がともに同じ麻酔医の担当であることを保証している。NOT条件は、主となる処置と重複していない処置を除外している。

10番の処置についてのみ結果を表示すると、次のようになる（見やすいようにevent_timeでソートしてある）。1人の麻酔医が複数の処置を同時に開始できる点に注目してほしい。

```
Events

proc_id   comparison_proc   anest_name   event_time   event_type
=================================================================
     10                10   'Baker'      '08:00'              +1
     10                20   'Baker'      '09:00'              +1
     10                10   'Baker'      '11:00'              -1
     10                20   'Baker'      '13:00'              -1
```

この同じproc_idを持つイベントの集合から、各イベントについて、それ以前に始まったイベントのevent_typesの合計を求められる。時間をさかのぼって計算したこの合計値は、先の図1、図2のステップチャートにおける1つ1つのステップの値を表している。

```
SELECT E1.proc_id, E1.event_time,
       (SELECT SUM(E2.event_type)
          FROM Events AS E2
         WHERE E2.proc_id = E1.proc_id
           AND E2.event_time < E1.event_time)
        AS instantaneous_count
  FROM Events AS E1
 ORDER BY E1.proc_id, E1.event_time;
```

このクエリの結果を10番の処置についてのみ表示すると、次のようになる。

```
proc_id   instantaneous_count
============================
     10                 NULL
     10                    1
     10                    2
     10                    1
```

proc_id：処置ID　　instantaneous_count：瞬間掛け持ち数

このSELECT文はビューに入れてもよいかもしれない（ビュー名はConcurrentProcsだとしよう）。そうすれば、次のSQL文で各処置の最大瞬間掛け持ち数を求められる。

```
SELECT proc_id, MAX(instantaneous_count) AS max_inst
  FROM ConcurrentProcs
 GROUP BY proc_id;
```

あるいは、直接Eventsビューから望む結果を引き出すこともできる。先の2つのSELECT文を次のようにまとめるのだ。

```
SELECT E1.proc_id,
       MAX((SELECT SUM(E2.event_type)
              FROM Events AS E2
             WHERE E2.proc_id = E1.proc_id
               AND E2.event_time < E1.event_time))
       AS max_inst_count
  FROM Events AS E1
 GROUP BY E1.proc_id;
```

ただし、このクエリのように集約関数の中にサブクエリ式を入れるのは、SQL-92では違法である。そのため、このクエリを使えるかどうかはRDBMSに依存する。

その2

リチャード・レムレーが考えた答えは、FROM句でサブクエリを使うというSQL-92の機能を利用している。そのため、答えその1でビューにしていた部分まで、1つのクエリの中に組み込むことができている。

```
SELECT P3.proc_id, MAX(ConcurrentProcs.tally)
  FROM (SELECT P1.anest_name, P1.start_time, COUNT(*)
          FROM Procs AS P1 INNER JOIN Procs AS P2
            ON P1.anest_name = P2.anest_name
           AND P2.start_time <= P1.start_time
           AND P2.end_time > P1.start_time
         GROUP BY P1.anest_name, P1.start_time)
       AS ConcurrentProcs(anest_name, start_time, tally)
       INNER JOIN Procs AS P3
               ON ConcurrentProcs.anest_name = P3.anest_name
              AND P3.start_time <= ConcurrentProcs.start_time
              AND P3.end_time > ConcurrentProcs.start_time
 GROUP BY P3.proc_id;
```

その3

答え　ここで紹介する解法は、2000年6月9日にレックス・ファン・デ・ポール（aavdpol@hotmail.com）から寄せられた。そのアイデアは、「すべての処置（P1）についてループする」というものである。

まず、開始時刻が処置P1の処置時間内に含まれている処置P2を探す。そういうP2を見つけたら、今度はその処置P2の開始時刻を含む処置（P3）の数をカウントする。これにより、特定の処置P1についての最大瞬間掛け持ち数が得られるのである。

これを実現するために、レックスは最初にこういうビューを作った。

```
CREATE VIEW Vprocs (id1, id2, total) AS
SELECT P1.proc_id, P2.proc_id, COUNT(*)
  FROM Procs AS P1, Procs AS P2, Procs AS P3
 WHERE P2.anest_name = P1.anest_name
   AND P3.anest_name = P1.anest_name
   AND P1.start_time <= P2.start_time
   AND P2.start_time < P1.end_time
   AND P3.start_time <= P2.start_time
   AND P2.start_time < P3.end_time
 GROUP BY P1.proc_id, P2.proc_id;
```

次に、このビューを使って処置P1ごとに最大数を求める。

```
SELECT id1 AS proc_id, MAX(total) AS max_inst_count
  FROM Vprocs
 GROUP BY id1;
```

面白いことに、この方法だとP2の終了時刻を見なくて済む。

その4

答え バート・C・ヒューズ（bhughes@twincities.net）は、Microsoft AccessのSQLに近い独自言語を使って解法を思いついた。ここに示すのは、彼のコードを単一のSQL文に書き換えたものである。

```
SELECT P1.proc_id, P1.anest_name, MAX(E1.ecount) AS maxops
  FROM Procs AS P1, -- E1は各麻酔医の処置の瞬間掛け持ち数
       (SELECT P2.anest_name, P2.start_time, COUNT(*)
          FROM Procs AS P1, Procs AS P2
         WHERE P1.anest_name = P2.anest_name
           AND P1.start_time <= P2.start_time
           AND P1.end_time > P2.start_time
         GROUP BY P2.anest_name, P2.start_time)
         AS E1(anest_name, etime, ecount)
 WHERE E1.anest_name = P1.anest_name
   AND E1.etime >= P1.start_time
   AND E1.etime <  P1.end_time
 GROUP BY P1.proc_id, P1.anest_name;
```

その5

答え もう1つのアプローチは、「時計」テーブルを用意することだ。分より小さい単位については考えなくてもよいだろうから、1日分なら24時間×60分＝1440行、1年分を用意するとしても52万5600行で済む。または、今日1日分をビューで作ることもできる。

```
SELECT X.anest_name, MAX(X.proc_tally)
  FROM (SELECT P1.anest_name, COUNT(DISTINCT proc_id)
          FROM Procs AS P1, Clock AS C
         WHERE C.clock_time BETWEEN P1.start_time AND P1.end_time
         GROUP BY P1.anest_name) AS X(anest_name, proc_tally)
 GROUP BY X.anest_name;
```

要するに、これはカレンダー補助テーブルの別バージョンである。この種のテーブルの行数は、よく知られているように粒度[訳注1]次第で決まる —— 普通はカレンダーなら1日を、時計なら1分を基本単位とする。あるいは次のように、現在時を分単位で格納するテーブルと、システム定数CURRENT_DATEを利用したビューを作ることもできる。

```
CREATE VIEW TodayClock (clock_time)
AS SELECT CURRENT_DATE + ticks FROM DayTicks;
```

訳注1：ここで言う粒度（りゅうど：granularity）とは、データをある目的に適切と思われるレベルに分解し格納するときの、そのレベルや度合い。

パズル 4

入館証

UPDATEで相関サブクエリを使用する

あなたの務める会社が「合理化（ライトサイジング）」（こういう場合、人員削減（ダウンサイジング）や外部委託（アウトソーシング）とは言わないんだな）を進めた結果、このたびあなたは、セキュリティ責任者とデータベース管理者を兼任することになった。そこであなたは手始めに、社員と社員が持つ有効な入館証番号を記載したリストを作ることにした。

仕様はこうである。社員は現在働いている職場に応じて、複数の入館証を持つことができる。ただし、同時に有効な入館証はそれらのうち1つだけだ。デフォルトで有効なのは、直近に発行された入館証、つまり新しい職場で発行された入館証だとする。入館証の番号は、偽造防止のためにランダムに割り当てられる。

この条件の下、あなたには社員と社員の有効な入館証番号をまとめたリストを作ってほしい。なお、入館証番号が有効であることは'A'（Active）で、無効であることは'I'（Inactive）で表すとしよう。

その1

答え 仕様から、各社員は1つしか有効な入館証を持てないことが分かる。したがって、このルールはデータベースレベルで実装しておくのがよいだろう。

```
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 Badges
(badge_nbr INTEGER NOT NULL PRIMARY KEY,
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
 issued_date DATE NOT NULL,
 badge_status CHAR(1) NOT NULL
   CHECK (badge_status IN ('A', 'I')),
 CHECK (1 <= ALL (SELECT COUNT(badge_status)
                    FROM Badges
                   WHERE badge_status = 'A'
                   GROUP BY emp_id)));
```

Badges：入館証テーブル	badge_nbr：入館証番号	emp_id：社員ID
issued_date：発行日	badge_status：入館証の状態	

公正を期して言っておくと、多くのRDBMSでは述語内での自己参照ができないため、最後のCHECK句のところでエラーが返されるだろう。しかし、これはSQL-92の適法な構文なのだ。

もちろん、CHECK句を削除して、社員が1つも有効な入館証を持たない状態を許すこともできる。でもそうすると、直近に発行された入館証の状態を'A'に更新する方法を考えなければならなくなってしまう。その方法とは、例えば次のようなUPDATE文だ。

```
UPDATE Badges
   SET badge_status = 'A'
 WHERE ('I' = ALL (SELECT badge_status
                     FROM Badges AS B1
                    WHERE Badges.emp_id = B1.emp_id))
   AND (issued_date = (SELECT MAX(issued_date)
                         FROM Badges AS B2
                        WHERE Badges.emp_id = B2.emp_id));
```

残念ながら、やはりこのUPDATE文にも多くのRDBMSはエラーを返す。今度の問題は相関名である。SQL-92では、UPDATE文におけるテーブル名の有効範囲を文全体としており、カレント行は更新対象となる列の値を参照するために使われることになっている。したがって、同じテーブルのほかの行を参照するには相関名を使わなければならないのだ。

まあ、そうした制限はさておき、これでパズルを解くクエリはとても簡単になった。

```
SELECT P.emp_id, emp_name, badge_nbr
  FROM Personnel AS P, Badges AS B
 WHERE B.emp_id = P.emp_id
   AND B.badge_status = 'A';
```

その2

答え　このパズルにはもう1つ、「入館証ごとに連番（badge_seq）を割り当てる」というアプローチがある。有効な入館証は、連番列にMIN関数やMAX関数を使って示す。

```
CREATE TABLE Badges
(badge_nbr INTEGER NOT NULL PRIMARY KEY,
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
 issued_date DATE NOT NULL,
 badge_seq INTEGER NOT NULL CHECK (badge_seq > 0),
 UNIQUE (emp_id, badge_seq),
 ... );
```

次に、有効な入館証の番号を得るビューを作ろう。

```
CREATE VIEW ActiveBadges (emp_id, badge_nbr)
AS SELECT emp_id, badge_nbr
     FROM Badges AS B1
    WHERE badge_seq = (SELECT MAX(badge_seq)
                         FROM Badges AS B2
                        WHERE B1.emp_id = B2.emp_id);
```

このアプローチの場合にも、入館証の紛失時や失効時に連番を振り直すための更新処理が必要になる。

```
UPDATE Badges
   SET badge_seq = (SELECT COUNT(*)
                      FROM Badges AS B1
                     WHERE Badges.emp_id = B1.emp_id
                       AND Badges.badge_seq >= B1.badge_seq);
```

また、このテーブル定義には、入館証の連番を目で見ることができ、各社員が持っている入館証の個数がより手軽に分かるという、思わぬメリットもある。

パ|ズ|ル|
5 アルファベット
あいまい検索と正規表現検索

「アルファベットだけを含む列」「少なくとも1文字はアルファベットを含む列」「アルファベットを1文字も含まない列」を制約で保証するにはどうすればよいだろう？

古い手続き型言語では、ファイル宣言においてデータフィールドをフォーマット制限付きで宣言する必要がある。分かりやすい例だと、COBOLやPL/Iがそうだ。また、データを読み込むときにフィルタリングするためのテンプレートを使うという方法もある。こちらの最もよく知られている例は、FORTRANスタイルのFORMAT文だろう。

その1

答え

「少なくとも1文字はアルファベットを含む列」「アルファベットを1文字も含まない列」を保証する制約については、オセロット・ソフトウェア社から文字集合の観点から考えられた、いかにもSQLらしい解答が寄せられた。一方、「アルファベットだけを含む列」については、SUBSTRING関数とBETWEEN述語をCHECK句の中で大量に使う、本当にひどいやり方しか私には考えつかなかった。何しろCHECK句だけで、スキーマ定義全体よりも大きくなってしまったんだ。

```
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	

これら3つのCHECK制約を使うには、あなたの使っているRDBMSが、大文字と小文字を区別するSQL-92の機能をサポートしていることが前提となる。アルファベットは大文字と小文字で異なる値を持っているが、ほかの文字はそうではない。この特性を利用することで、ある列がアルファベットを含むか含まないかを制御できるのだ。

その2

答え 「アルファベットだけを含む列」を保証するには、やはりLIKE述語に適用する正規表現パーサのようなRDBMSの独自拡張機能を使わないと難しい。

```
all_alpha VARCHAR(6) NOT NULL
CHECK (TRANSLATE (all_alpha USING one_letter_translation) =
'xxxxxx')
```

one_letter_translationは、「すべての文字を“x”へ写像する」という変換定義に付けた名前だ。TRANSLATE関数とその変換定義は標準SQLに含まれているが、まだ一般的ではない。変換定義を行う構文は次のようなものだが、ここでは詳細は割愛する。

```
<変換定義> ::=
  CREATE TRANSLATION <変換定義名> FOR <変換対象となる文字の集合>
  TO <変換後の文字の集合> FROM <変換対象データ>;
```

その3

答え 標準SQLの正規表現述語はPOSIX構文に基づいており、次のように書ける[訳注1]。

```
all_alpha VARCHAR(6) NOT NULL
CHECK (all_alpha SIMILAR TO '[:ALPHA:]+')
```

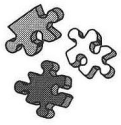
だが念のため、あなたが使っているRDBMSの仕様を調べたほうが確実であろう。

訳注1：[:ALPHA:]は[a-zA-Z]とも書ける。

パ|ズ|ル|
6

ホテルの予約

重複する期間の抽出（その2）



スコット・ガンマンズは、CompuServeのWATCOM[訳注1]フォーラムに次のような問題（全く同じではないが）を投稿してきた。

まず、自分は「ホテルSQL」の受付係だと想像してほしい。そして、次のようなHotelテーブルがあるとする。

```
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));
```

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：宿泊客名	

現時点では、CHECK句によって「到着前に出発はできない」という整合性制約が設定されているけれども、これではまだ足りない。このテーブルにさらに、「前の宿泊客の出発日が次の客の到着日と衝突する場合、その部屋には予約を追加できない」というルールも適用する方法を考えてみよう。つまり、ダブルブッキングを防ぎたいのだ。

その1

答え

次に示す解法は、CHECK句の中でかなり複雑なSQLを使えるRDBMSでなければ動かない。実際のところ、多くのRDBMSは次のSQLをサポートしていないだろう。

```
CREATE TABLE Hotel
(room_nbr INTEGER NOT NULL,
 arrival_date DATE NOT NULL,
 departure_date DATE NOT NULL,
 guest_name CHAR(30),
 PRIMARY KEY (room_nbr, arrival_date),
```

訳注1：サイベースのRDBMS製品「SQL Anywhere Studio」の旧称。

```
CHECK (departure_date >= arrival_date),
CHECK (NOT EXISTS
        (SELECT *
           FROM Hotel AS H1, Hotel AS H2
          WHERE H1.room_nbr = H2.room_nbr
            AND H1.arrival_date BETWEEN H2.arrival_date
                                    AND H2.departure_date)));
```

その2

答え 別解はテーブル設計を見直すことである。日付と部屋の組に対して1行を与えると、Hotelテーブルは次のようになる。

```
CREATE TABLE Hotel
(room_nbr INTEGER NOT NULL,
 occupy_date DATE NOT NULL,
 guest_name CHAR(30) NOT NULL,
 PRIMARY KEY (room_nbr, occupy_date, guest_name));
```

occupy_date：宿泊日

こうすればCHECK句は不要になるが、ディスク容量を多く消費する可能性があるし、いくぶん冗長になる。今日の記憶媒体の安価さを考えれば、ディスク容量は問題にならないかもしれないが、冗長性はいかなるときも問題だ。また、ある部屋を予約するためのINSERT文を複数発行する場合には、その日付に“歯抜け”がないことを保証する方法を考えねばならない。

ここだけの話だが、SQL-92のフルレベルには、2つの期間がオーバーラップするか否かをテストするOVERLAPS述語が定義されている。いわば、現行のRDBMSが持つBETWEEN述語の時間版だ[訳注2]。しかし、この述語を使えるRDBMS製品は今のところごく少数しかない。

訳注2：正確には、BETWEEN述語が時間軸上のある1点が指定した期間内に含まれるかどうかをテストするのに対し、OVERLAPS述語は期間同士が重なりあうか否かをテストする。

その3

答え

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、答えその1に対し、「H1.arrival_date BETWEEN H2.arrival_date AND H2.departure_dateという述語では、前の宿泊客の出発日に到着する客の予約が入れられない。そもそも、問題文のダブルブッキングを避けるルールは非現実的だ」という批判を寄せてきた。

しかし、OracleではCHECK制約の中にサブクエリを入れられないし、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。そこで彼は、期間の制約を施行するためにWITH CHECK OPTION付きのビューを利用した。

```
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;
```

こうすると、例えば次のようなINSERT文を実行した場合でも、HotelStaysビューの結果に現れないことから、Roe氏の予約を追加できなかったことが分かる。

```
INSERT INTO HotelStays
VALUES (1, '2008-01-01', '2008-01-03', 'Coe');
-- HotelStaysビューにはCoe氏だけが表示される
```

```
INSERT INTO HotelStays
VALUES (1, '2008-01-03', '2008-01-05', 'Doe');
-- HotelStaysビューにはCoe氏のほか、Doe氏も表示される
```

```
INSERT INTO HotelStays
VALUES (1, '2008-01-02', '2008-01-05', 'Roe');
-- Roe氏の予約（INSERT）は失敗し、
-- HotelStaysビューにはCoe氏しか表示されない
```

パ|ズ|ル|
7 ファイルのバージョン管理
順序を入れ子集合で表す

1995年11月、スティーブ・ティルソンはこんな問題を送ってきた。

「あなたに、このパズルを解いていただきたいのです。恐らく、今の私は『木を見て森を見ず』の状態でしょう。しかし、この問題を膨大な自己参照を使う破目に陥ることなく、エレガントに解答することは、実に至難の業だと思います。このパズルには6つの問題が含まれていますが、私の目下の疑問は『明らかな循環参照を、テーブル設計の段階で排除する方法があるか否か』です。

あなたはある組織において、文書ファイルのバージョンを一括管理するポートフォリオを、検索または参照のために保存しなければならないとします[訳注1]。本来、ファイルにはいろいろな属性がありますが、このパズルで必要なのは少しだけです。

```
CREATE TABLE Portfolios
(file_id INTEGER NOT NULL PRIMARY KEY,
 issue_date DATE NOT NULL,
 superseded_file_id INTEGER NOT NULL
   REFERENCES Portfolios(file_id),
 supersedes_file_id INTEGER NOT NULL
   REFERENCES Portfolios(file_id));
```

Portfolios：ポートフォリオテーブル　　file_id：ファイルID
issue_date：ファイルの発行日　　superseded_file_id：先行バージョンのファイルID
supersedes_file_id：後続バージョンのファイルID

問題は次の6つです。

- あるバージョンの後続バージョンを追跡できること
- あるバージョンの先行バージョンを追跡できること
- 過去のバージョンを復活できること（これを行うには復活したバージョンをその連鎖に組み込む必要があり、そのために循環参照が避けられなくなっている）
- issue_date列によってあるファイルバージョンの発行日を突き止められること（た

訳注1：ここで言うポートフォリオは、一般的にイメージされる金融用語ではなく、紙を何枚も綴った「バインダー」のようなものを想像してほしい。ファイルも同様だ。

だし、過去のバージョンが復活していた場合、発行日をどちらにするかという厄介な問題が生じる）

- SELECT文にどんなバージョンが含まれていようと、最新バージョンは選択できること
- バージョンの連鎖を調べる監査証跡を再作成できること

以上、よろしくお願いします。」

その1

答え

スティーブはまだ、ポインタチェイン[訳注2]と手続き型言語の考え方にとらわれている。だが、SQLで考えるにはそれではダメだ！　仕様にある「後続」と「先行」という語が、この問題が順序番号を扱うものであることを如実に示している。こういうときには、番号の代わりに入れ子集合の性質をうまく使おう。

まず、各ファイルバージョンの全情報を保持するテーブルを作る。

```
CREATE TABLE Portfolios
(file_id INTEGER NOT NULL PRIMARY KEY,
 そのほかの列 ...);
```

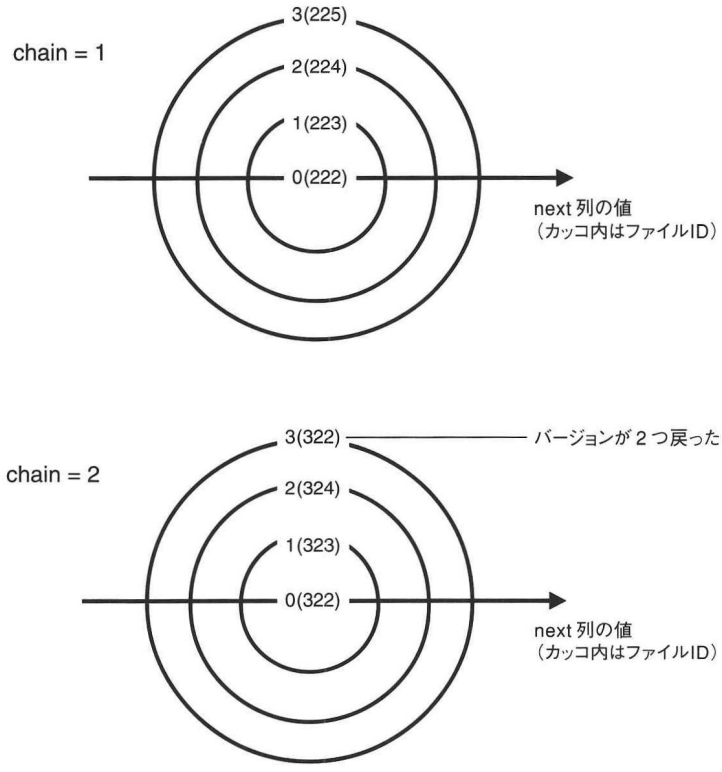
次に、バージョンの連鎖情報を持つテーブルを作る。chain列とnext列が重要なポイントである。

```
CREATE TABLE Succession
(chain INTEGER NOT NULL,
 next INTEGER DEFAULT 0 NOT NULL CHECK (next >= 0),
 file_id INTEGER NOT NULL REFERENCES Portfolios(file_id),
 suc_date DATE NOT NULL,
 PRIMARY KEY(chain, next));
```

Succession：連鎖テーブル	chain：連鎖系列	next：後続バージョンのファイルID
file_id：ファイルID	suc_date：発行日	

訳注2：リスト構造データなどで、各データが次のデータを指し示すポインタを持つことで次々にデータを参照可能にする実装方法。RDBテーブルでこれを表現すると、RDBMSの独自拡張に頼らない限り、スティーブの言うように検索クエリが自己結合の嵐となる。典型的な例は、階層データを表現する隣接リストモデルのテーブル定義である。詳しくは『プログラマのためのSQL 第2版』（ピアソン・エデュケーション刊）の第28章を参照。

next列の値を、0を中心に数直線上に描かれる「円」の半径だとイメージしてほしい。円はファイルのバージョンを表している。連鎖（chain）の始点となるバージョンの円の半径（つまりnext列の値）は0である。続くバージョンは、0を取り囲む半径1の円になる。連鎖の3番目に来るバージョンは、さらにその円を取り囲む半径2の円になる（**図**[訳注3]）。これは、Portfoliosテーブルに新しいファイルを登録したときに、Successionテーブルにも後続バージョンを表す行を登録することと同義である。そのときにnext列に入る値は、その連鎖におけるnext列の最大値より1だけ大きな数字になる。こうしてできあがるのが、バージョンの連鎖を表す「入れ子集合[訳注4]」である！



図：next列の値による入れ子集合

訳注3：この図は訳者が追加。

訳注4：この例だと、chain=1とchain=2という2つの連鎖が存在しているので、同心円的な入れ子集合も2系統が存在することになる。自然数を入れ子集合で表現するアイデアは、一見すると突飛に感じるかもしれないが、SQLではランキングや累計算出、また木構造のデータを表現する手段としてよく利用される。パズル35も参照。

例として、220番台のfile_idを持つファイルと、330番台のfile_idを持つファイルのバージョン連鎖を示そう。どちらも、999番というバージョンを最後尾に持っている。

```
CREATE TABLE Portfolios
(file_id INTEGER NOT NULL PRIMARY KEY,
 stuff CHAR(15) NOT NULL);  -- ファイルの名前
```

```
INSERT INTO Portfolios
VALUES (222, 'stuff'),
       (223, 'old stuff'),
       (224, 'new stuff'),
       (225, 'borrowed stuff'),
       (322, 'blue stuff'),
       (323, 'purple stuff'),
       (324, 'red stuff'),
       (325, 'green stuff'),
       (999, 'yellow stuff');
```

```
CREATE TABLE Succession
(chain INTEGER NOT NULL,
 next INTEGER NOT NULL,
 file_id INTEGER NOT NULL REFERENCES Portfolios(file_id),
 suc_date DATE NOT NULL,
 PRIMARY KEY(chain, next));
```

```
INSERT INTO Succession
VALUES (1, 0, 222, '2007-11-01'),
       (1, 1, 223, '2007-11-02'),
       (1, 2, 224, '2007-11-04'),
       (1, 3, 225, '2007-11-05'),
       (1, 4, 999, '2007-11-25'),
       (2, 0, 322, '2007-11-01'),
       (2, 1, 323, '2007-11-02'),
       (2, 2, 324, '2007-11-04'),
       (2, 3, 322, '2007-11-05'),
       (2, 4, 323, '2007-11-12'),
       (2, 5, 999, '2007-11-25');
```

では、スティーブの質問に1つずつ答えていこう。

- SELECT文にどんなバージョンが含まれていようと、最新バージョンは選択できること

これは次のクエリでOKだ[訳注5]。

```
SELECT DISTINCT P1.file_id, stuff, suc_date
  FROM Portfolios AS P1, Succession AS S1
 WHERE P1.file_id = S1.file_id
   AND next = (SELECT MAX(next)
                 FROM Succession AS S2
                WHERE S1.chain = S2.chain);

 file_id   stuff          suc_date
=====================================
     999   'yellow stuff' '2007-11-25'
```

複数の連鎖が同一ファイルへ統合された場合に対処するため、SELECT DISTINCTオプションを使う必要がある点に注意しよう。

■ バージョンの連鎖を調べる監査証跡を再作成できること

これも軽くクリアできる[訳注6]。

```
SELECT chain, next, P1.file_id, stuff, suc_date
  FROM Portfolios AS P1, Succession AS S1
 WHERE S1.file_id = P1.file_id
 ORDER BY chain, next;

 chain   next   file_id   stuff            suc_date
=======================================================
     1      0       222   'stuff'          '2007-11-01'
     1      1       223   'old stuff'      '2007-11-02'
     1      2       224   'new stuff'      '2007-11-04'
     1      3       225   'borrowed stuff' '2007-11-05'
     1      4       999   'yellow stuff'   '2007-11-25'
     2      0       322   'blue stuff'     '2007-11-01'
     2      1       323   'purple stuff'   '2007-11-02'
     2      2       324   'red stuff'      '2007-11-04'
     2      3       322   'blue stuff'     '2007-11-05'
     2      4       323   'purple stuff'   '2007-11-12'
     2      5       999   'yellow stuff'   '2007-11-25'
```

訳注5：クエリの実行結果は訳者が追加。

訳注6：クエリの実行結果は訳者が追加。

■　あるバージョンの先行バージョンを追跡できること[訳注7]

```
SELECT S1.file_id, ' superseded ', S2.file_id,
       ' on ', S1.suc_date
  FROM Succession AS S1, Succession AS S2
 WHERE S1.chain = S2.chain
   AND S1.next = S2.next + 1
   AND S1.file_id = :my_file_id; -- すべてのバージョンについて調べたい
                                 のなら、この条件は削除する
```

■　過去のバージョンを復活できること（これを行うには復活したバージョンをその連鎖に組み込む必要があり、そのために循環参照が避けられなくなっている）

```
BEGIN
-- 新規バージョンのための行を作る
INSERT INTO Portfolios VALUES (1000, 'sticky stuff');

-- 連鎖の中の任意のold_file_idをnew_file_idとして追加する
INSERT INTO Succession (chain, next, file_id, suc_date)
VALUES ((SELECT DISTINCT chain
           FROM Succession AS S1
          WHERE S1.file_id = :old_file_id),
        (SELECT MAX(next) + 1
           FROM Succession AS S1
          WHERE S1.chain = (SELECT DISTINCT chain
                              FROM Succession AS S2
                             WHERE file_id = :old_file_id)),
       :new_file_id, :new_suc_date);
END;
```

ここでの問題は、1つのバージョンが複数の既存バージョンの後続となることと、逆に複数のバージョンが1つの既存バージョンの後続となることを許した点である。つま

訳注7：例えば、999番のバージョンに先行するのは、225と323だ。:my_file_idを999としてこのクエリを実行すると、次のような結果が得られる。

```
S1.file_id   superseded   S2.file_id    on    suc_date
============================================================
    999      'superseded'        225    'on'  '2007-11-25'
    999      'superseded'        323    'on'  '2007-11-25'
```

反対に、後続するバージョンを調べる場合は「S1.next = S2.next + 1」を「S1.next = S2.next - 1」に変えればよい。

り、連鎖が一直線に並ぶとは限らないわけだ。もし、:old_file_idが2つ以上の連鎖に含まれていた場合、このコードではうまくいかない。新しい後続バージョンのfile_idやchainの番号に新しい番号を与えることでこの問題を解決することもできるが、SQL文が汚くなるし、今の私にはそれをするだけの時間もない。ぜひ、各自でトライしてみてほしい。

- issue_date列によってあるファイルバージョンの発行日を突き止められること（ただし、過去のバージョンが復活していた場合、発行日をどちらにするかという厄介な問題が生じる）

この入れ子集合のスキーマを使えば、何も悩む点はない。任意のfile_idについてSELECT文を発行して日付を調べればよいし、一連の連鎖を確認したければnext列を見れば済む。スティーブは、suc_date列の値がnext列の値に対応して増加していかなければならない、とは言っていなかったが、あるいはそういう要請もあるだろう。その場合には、CHECK句を追加することで対処できる。

パ|ズ|ル|
8

プリンタの割り当て

値の範囲に応じた結果を返す

1996年9月12日、ヨーゲシュ・チャチャはある問題に突き当たり、それをCompuServeに送ってきた。

彼の職場では、ユーザがいつも間違ったプリンタを使ってしまうことに困っていた。そこで彼は、各ユーザが使用すべきプリンタを導くテーブルをシステムに追加することにした。そのテーブルはこんな感じだ。

```
CREATE TABLE PrinterControl
(user_id CHAR(10), -- NULLは空きプリンタを意味する
 printer_name CHAR(4) NOT NULL PRIMARY KEY,
 printer_description CHAR(40) NOT NULL);
```

PrinterControl：プリンタ管理テーブル	user_id：ユーザID
printer_name：プリンタ名	printer_description：備考

運用ルールは次の2つである。

1. テーブルに登録されているユーザは、自分が使うべきprinter_nameのプリンタを使うこと
2. テーブルに登録されていないユーザは、user_idがNULLの共有プリンタ（Common Printer）を使うこと

例えば、PrinterControlテーブルは次の状態だとしよう。

```
PrinterControl

user_id    printer_name   printer_description
=======================================================
'chacha'   'LPT1'         'First floor's printer'
'lee'      'LPT2'         'Second floor's printer'
'thomas'   'LPT3'         'Third floor's printer'
NULL       'LPT4'         'Common printer for new user'
NULL       'LPT5'         'Common printer for new user'
```

chachaが帳票を印刷するときには、LPT1だけを使うことが許される。一方、celkoという名前のユーザは、LPT4かLPT5のどちらかを使うことになる。前者のケースでは、クエリを1つ発行すれば1行だけが選択されるので問題ない。だが後者のケースでは2行が選択されてしまい、その結果をそのまま使うことはできない。

さて、この問題を1つのクエリで解くことはできるだろうか？

その1

答え 私ならば、問題はデータにあると言いたい。user_id列を見てほしい。この名前は一意であるべきだが、複数行にNULLが格納されている。また、もう1つ現実的な問題がある。LPT4とLPT5の片一方だけが使われすぎないよう、うまく負荷分散したいのだ。

変てこなクエリを書く前に、ここはテーブル定義を変更しよう。

```
CREATE TABLE PrinterControl
(user_id_start  CHAR(10) NOT NULL,
 user_id_finish CHAR(10) NOT NULL,
 printer_name   CHAR(4) NOT NULL,
 printer_description CHAR(40) NOT NULL,
 PRIMARY KEY (user_id_start, user_id_finish));
```

PrinterControl：プリンタ管理テーブル
user_id_start：ユーザID（範囲開始）
user_id_finish：ユーザID（範囲終了）
printer_name：プリンタ名
printer_description：備考

すると、サンプルデータは次のようになる。

```
PrinterControl

user_id_      user_id_     printer_   printer_
start         finish       name       description
============================================================
'chacha'      'chacha'     'LPT1'     'First floor's printer'
'lee'         'lee'        'LPT2'     'Second floor's printer'
'thomas'      'thomas'     'LPT3'     'Third floor's printer'
'a'           'mzzzzzzz'   'LPT4'     'Common printer #1'
'n'           'zzzzzzzz'   'LPT5'     'Common printer #2'
```

クエリはこうだ。

```
SELECT MIN(printer_name)
  FROM PrinterControl
 WHERE :my_id BETWEEN user_id_start AND user_id_finish;
```

ここでの“トリック”は、名前の開始値と終了値である。これは“a”から“zzzzzzzz”までの文字列を、いくつかの範囲に分割する役割を果たす。範囲は自由に決められる。先の分割の仕方だと、celkoというユーザはLPT4だけを使うことができる。なぜなら、“celko”という文字列は“a”と“mzzzzzzz”の間に収まるからだ。同様に、normanはLPT5だけを使える。どんなユーザIDが使われるかを知っていれば、分割の境界を注意深く設定することで、2台のプリンタにかかる負荷をほぼ半々にできるだろう。

この解答では、共有プリンタは常に大きいLPT番号を持っていると仮定している。chachaがこのテーブルを検索した場合、彼は(LPT1, LPT4)という結果集合を得るが、その中から最小値であるLPT1を選択する。RDBMSのオプティマイザが賢ければ、主キーのインデックスを使って、極値関数の実行速度を向上させるだろう。

その2

答え

リチャード・レムレーは、こんな別解を思いついた。

```
SELECT COALESCE(MIN(printer_name),
       (SELECT MIN(printer_name)
          FROM PrinterControl AS P2
         WHERE user_id IS NULL))
  FROM PrinterControl AS P1
 WHERE user_id = :user_id;
```

これは見た目以上にトリッキーだ。ユーザ celkoで検索した場合、外側のWHERE句はuser_id = 'celko'となる。だが、そんなユーザは登録されていないので、PrinterControlテーブルのコピーであるP1からは1行も返らない、と思うのではないだろうか。

ところがそうはならない。なぜなら、確かに、

```
SELECT col
  FROM SomeTable WHERE 1 = 2;
```

というクエリは1行も返さないが、

```
SELECT MAX(col)
  FROM SomeTable WHERE 1 = 2;
```

というクエリは、1行だけ結果を返すのである。結果に含まれているのはNULLだ。これは空集合に集約関数を適用した場合に見られる奇妙な特性である。したがって、次のクエリはうまく動く。

```
SELECT COALESCE(MAX(col), something_else)
  FROM SomeTable
 WHERE 1 = 2;
```

WHERE句はMAX(col)の値を決めるためだけに使われ、行を返すか否かを決定することはない。それはSELECT句の仕事なのだ。集約されたMAX(col)はNULLになり、それが返される。それゆえ、COALESCE関数も正しく動く。

このクエリの欠点は、共有プリンタから選ぶときに常に同じプリンタ（LPT4）を返すため、負荷分散にならないことだ。これを防ぐには、NULLをゲストユーザで置き換えるようなUPDATE文を追加すればよい。そうすれば、複数のプリンタに分散される。

その3

答え 答えその2の欠点を安易に修正したのが、次のクエリである。

```
SELECT COALESCE(MIN(printer_name),
       (SELECT DISTINCT CASE
                        WHEN :user_id < 'n'
                        THEN 'LPT4'
                        ELSE 'LPT5' END
          FROM PrinterControl
         WHERE user_id IS NULL))
  FROM PrinterControl
 WHERE user_id = :user_id;
```

この解の欠点は、共有プリンタの選択がすべてクエリの中で処理されており、テーブルを全く参照しないところである。そもそもこのやり方をとるなら、サブクエリ内のCASE式より後ろにあるFROM句とWHERE句はすべて削除してもかまわない。ただし、それは「データベース内にプリンタについての情報を記録しない」ということを意味

する。プリンタを追加したり削除したりするときはデータベースではなく、このクエリを変更することになる。そこが情報を保存している場所だからだ。これはよい設計とは言えない。

その4

答え　テーブル設計を見直す方法は、もう1つある。未登録のゲストユーザにも割り当て可能かどうかを保持するフラグ列を持つのだ。そこに、割り当て可能なプリンタ（共有プリンタ）には'Y'を、登録ユーザしか使えないプリンタには'N'をセットする。

```
CREATE TABLE PrinterControl
(user_id CHAR(10), -- NULLは空きプリンタを意味する
 printer_name CHAR(4) NOT NULL PRIMARY KEY,
 assignable_flag CHAR(1) DEFAULT 'Y' NOT NULL
       CHECK (assignable_flag IN ('Y', 'N')),
 printer_description CHAR(40) NOT NULL);
```

PrinterControl：プリンタ管理テーブル　user_id：ユーザID　printer_name：プリンタ名
assignable_flag：割り当て可能フラグ　printer_description：備考

そして、テーブルを次のようにUPDATEする。これは、割り当て可能な共有プリンタのうちの1つを:guest_idのユーザに割り当てている。

```
UPDATE PrinterControl
   SET user_id = :guest_id
 WHERE printer_name
       = (SELECT MIN(printer_name)
            FROM PrinterControl
           WHERE assignable_flag = 'Y'
             AND user_id IS NULL);
```

また、運用を続けるうちに、ゲストユーザをすべてNULLクリアしなければならない場面にも遭遇するだろう。それは次のクエリで行う。

```
UPDATE PrinterControl
   SET user_id = NULL
 WHERE assignable_flag = 'Y';
```

パ|ズ|ル|
9

席空いてますか？

テーブルサイズを最小限に抑える

突然だが、あなたは1000席の座席を持つレストランの経営者だ。ウェイターが客を席に案内すると、あなたはそのことを座席テーブル（一瞬「テーブル・テーブル」と言いそうになったが、分かりにくいのでやめた）に記録する。同様に、客が食事を終えると、その席が空いたことを記録する。

そうなると欲しいのは、空席の一覧を「開始席番と終端席番がひとかたまりになった形」で出力するクエリである。そうそう、1つ注意が必要なのは、このデータベースが載せられているマシンがメインフレームではなく、PDAであることだ。

そこで問題なのだが、「使用するメモリ容量を可能な限り小さく」という条件付きで、座席テーブルとクエリを考えてほしい。座席番号は整数だとする。

最初に思いつく方法は、「空き／使用」フラグの列を座席テーブルに追加することだろう。空席を調べるクエリでは、このフラグを利用するわけだ。だがこの方法では、レストラン全体で「1つの整数列（座席番号）と1文字の文字型の列（フラグ）×1000行」を要することになる。確かにこれでもうまくいくが、最小のメモリ使用量という要件は満たせていない。残念！

その1

答え

余計な列を持たないように、フラグを座席番号の正負で表現するのはどうだろう。もっとも、2つの属性を1つの列に折り込むことは、関係モデルの観点からは勧められない。しかも、1000行を必要とするという点は変わらない。

このアプローチで席を“使用中”にするには、次のSQL文を実行する。

```
UPDATE Seats
   SET seat_nbr = -seat_nbr
 WHERE seat_nbr = :my_seat;
```

Seats：座席テーブル　　seat_nbr：座席番号

状態を“空席”に戻すときにも、同じUPDATE文で行える。また、閉店時間に席の状態をリセットするには、「SET seat_nbr = ABS(seat_nbr)」とすればよい。

その2

答え　第2の方法として考えられるのは、使用中の座席を保持する列だけを持つテーブルを追加し、空席テーブルとの間で席番号を移動させることだ。こうすると、両テーブルを合わせて1000行になる。ちょっとひねった答えだが、もっとひねったのが次の答えだ。

その3

答え　用意するテーブルは1つだけだ。そこに、0から1001までの座席番号を格納する行を作成する（0と1001はあらかじめ挿入しておく。0番と1001番の席は存在しないが、席の終端を示す“番兵”として機能し、コードを簡単にしてくれる）。

客が席に着いたら、その番号の行をテーブルに挿入する。逆に席が空いたら、その番号を削除する。この方法だと、すべて空席の場合にはテーブルにダミーの2行（0番と1001番）しか残らず、満席の場合（最大件数時）でも1002行（2004バイト）で済む。

次のビューは、このテーブルから座席番号が連続して抜けている部分[訳注1]、つまり空席の“かたまり”を見つけ出し、その一番若い座席番号を返す。

```
CREATE VIEW Firstseat (seat)
AS SELECT (seat + 1)
     FROM Restaurant
    WHERE (seat + 1) NOT IN (SELECT seat FROM Restaurant)
      AND (seat + 1) < 1001;
```

Restaurant：レストランの座席テーブル　　seat：座席番号

同様に、次のビューは空席のかたまりの一番最後の座席番号を返す。

```
CREATE VIEW Lastseat (seat)
AS SELECT (seat - 1)
     FROM Restaurant
    WHERE (seat - 1) NOT IN (SELECT seat FROM Restaurant)
      AND (seat - 1) > 0;
```

では、この2つのビューを使って、空席のかたまりを一覧表示しよう。

訳注1：このような1つ以上連続する欠番のことを「ギャップ」とも言う。パズル57および58も参照。

```
SELECT F1.seat AS start, L1.seat AS finish,
       ((L1.seat - F1.seat) + 1) AS available
  FROM Firstseat AS F1, Lastseat AS L1
 WHERE L1.seat = (SELECT MIN(L2.seat)
                    FROM Lastseat AS L2
                   WHERE F1.seat <= L2.seat);
```

このクエリでは、各かたまりの空席数も表示している。これが分かると、ウェイターがグループ客を案内するときに便利だろう。

その4

答え　リチャード・レムレーは、答えその3で示したビューとクエリを1つにまとめた。このクエリ1つで、空席のかたまりを表示できる。

```
SELECT (R1.seat + 1) AS start,
       (MIN(R2.seat) - 1) AS finish
  FROM Restaurant AS R1
       INNER JOIN
       Restaurant AS R2
    ON R2.seat > R1.seat
 GROUP BY R1.seat
HAVING (R1.seat + 1) < MIN(R2.seat);
```

その5

答え　答えその4のバリエーションとして、SQL-99規格の新機能であるOLAP関数を使う解も考えられる。この方法だと、座席数など追加の情報も多少得られる。

```
SELECT X.seat, X.rn,
       (seat - rn) AS available_seat_cnt
  FROM (SELECT seat, ROW_NUMBER() OVER (ORDER BY seat)
          FROM Restaurant) AS X(seat, rn)
 WHERE rn <> seat;
```

available_seat_cntはseat未満の空席数である。これは、レストランの座席がいくつかのグループに分かれている場合などに便利かもしれない。

パ|ズ|ル|
10

年金おくれよ

連続と直近を表現する

1994年11月にカナダ在住のプログラマ、ルーク・ティモウスキがCompuServeのMicrosoft Accessフォーラムに面白い問題を投稿してきた。当時、彼は年金基金に関するデータの処理に取り組んでいて、SQL-92で書けば次のようなテーブルを扱っていた。

```
CREATE TABLE Pensions
(sin CHAR(10) NOT NULL,
 pen_year INTEGER NOT NULL,
 month_cnt INTEGER DEFAULT 0 NOT NULL
      CHECK (month_cnt BETWEEN 0 AND 12),
 earnings DECIMAL(8,2) DEFAULT 0.00 NOT NULL);
```

Pensions：年金テーブル	sin：社会保険番号	pen_year：年金保険料の納付年
month_cnt：月数	earnings：年収	

SINはカナダで納税者を識別するために使われている社会保険番号（Social Insurance Number）で、米国のSSN（Social Security Number：社会保障番号）に相当する。また、pen_year列は年金保険料の納付年を、month_cnt列はその年に社員が働いた月数を、earnings列はその年の総収入を保持している。

問題は、各社員の直近60ヶ月の総収入を求めるというものだ。この数値は連続していなければならず、社員1人1人の年金を算出するために使われる。さかのぼる年は、最短で5年（各年のmonth_cntが全部12だった場合）、最長で60年（各年のmonth_cntが全部1だった場合）だ。ただし、丸4年働いた後、5年目は働かなかった（つまり全く働かなかった年がある）社員は年金の受給資格を一切得られない。

この問題が手強いのは、「直近」と「連続する」をSQLで表現することが難しいからだ。

ヒント

社員が全く働かなかった年についてもテーブルに行を挿入しておくと、クエリが簡単になるだけでなく、新しい情報が得られたときにもUPDATEするレコードが存在して便利だ。

その1

答え 次に示すビューは、

(1) 社員が働いた(= month_cntが0より大きい)
(2) month_cntの総計が60以上である

という2つの条件を満たす、連続する期間の始点と終点を得ている。

```
CREATE VIEW PenPeriods (sin, start_year, end_year, earnings_tot)
AS SELECT P0.sin, P0.pen_year, P1.pen_year,
          (SELECT SUM(earnings) -- 期間単位の総収入
             FROM Pensions AS P2
            WHERE P2.sin = P0.sin
              AND P2.pen_year BETWEEN P0.pen_year
                                  AND P1.pen_year)
     FROM Pensions AS P0, Pensions AS P1
    WHERE P1.sin = P0.sin -- 支払期間を作るための自己結合
      AND P1.pen_year >= (P0.pen_year - 4)
      AND 0 < ALL (SELECT month_cnt        -- 連続する月
                     FROM Pensions AS P3
                    WHERE P3.sin = P0.sin
                      AND P3.pen_year BETWEEN P0.pen_year
                                          AND P1.pen_year)
      AND 60 <= (SELECT SUM(month_cnt)     -- 60ヶ月以上
                   FROM Pensions AS P4
                  WHERE P4.sin = P0.sin
                    AND P4.pen_year BETWEEN P0.pen_year
                                        AND P1.pen_year);
```

SELECT句にサブクエリ式を入れるのは、SQL-92から使用可能な"トリック"だが、今では多くのRDBMS製品が実装している。

注意が必要なのは、このクエリが60ヶ月以上の期間すべてを返すことである。私たちが本当に欲しいのは、直近のend_yearだったはずだ。これを求めるには、いま定義したビューに対してMAX(end_year)で条件を指定すればよい。

```
SELECT *
  FROM PenPeriods AS P0
 WHERE end_year = (SELECT MAX(end_year)
                     FROM PenPeriods AS P1
                    WHERE P1.sin = P0.sin);
```

コードは少し汚くなるが、SQL-92ではHAVING句を使って同じことが書ける。やろうと思えば、これらのサブクエリをEXISTS述語を使って結合することもできるだろう。

練習問題として、最後のサブクエリに「P0.pen_yearとP1.pen_yearの間に、P4.pen_yearより大きい年が存在しない」という条件を追加して、やはり60ヶ月以上連続する期間の総収入を求めてみてほしい。

その2

答え

答えその1の最後の問いかけに応じてCompuServe経由で送られてきた改良版の答えの多くは、私の解法をベースとしていた。ところが、最良の解答はリチャード・レムレーが送ってきた、私の解法とは全く異なるアプローチによるものだった。彼の解は、Pensionsテーブルの3つのコピーP0、P1、P2（この順で時間順に並んでいる）を使う。

```
SELECT P0.sin,
       P0.pen_year AS start_year,
       P2.pen_year AS end_year,
       SUM(P1.earnings) AS sumofearning
  FROM Pensions AS P0, Pensions AS P1, Pensions AS P2
 WHERE P0.month_cnt > 0
   AND P1.month_cnt > 0
   AND P2.month_cnt > 0
   AND P0.sin = P1.sin
   AND P0.sin = P2.sin
   AND P0.pen_year BETWEEN P2.pen_year - 59 AND (P2.pen_year - 4)
   AND P1.pen_year BETWEEN P0.pen_year AND P2.pen_year
 GROUP BY P0.sin, P0.pen_year, P2.pen_year
HAVING SUM(P1.month_cnt) >= 60
   AND (P2.pen_year - P0.pen_year) = (COUNT(*) - 1);
```

レムレーいわく「もしmonth_cnt=0という行を許さなければ、問題はもっと簡単になります。この行を除外するために、無駄なWHERE条件を3つも使っています！」よいデータ設計は人生を簡単にしてくれるという、もう1つの実例と言わねばなるまい！

この解答の見事な点は2つある。1つ目は、5～60年の範囲（それぞれ、month_cntが60に達するのに必要な最小年数と最大年数）でどれだけ連続しているかを調べるためにBETWEEN述語を使っていること。2つ目は、期間に含まれる年が連続していることを保証するために、集約キーをHAVING句の条件で使っていることだ。

私がWATCOM SQL 4.0[訳注1]でこのクエリの実行計画を見積もってみたところ、レムレーの解は私のオリジナルの解より4倍良好な結果が出た。だが実際に実行してみると、差はそれ以上であった。これは多分、レムレーの解が3テーブル間の自己結合（普通ならとても高コストだ）を含んでいたせいでプランナ[訳注2]がだまされたからだろう。

その3

答え

1999年、ジャヴィド・ダウベゴヴィッチは本書の初版に次のようなメールをくれた。

「どちらの解もそれぞれに見事です（あなたのSQL-92版も、リチャードのSQL-89版も）。正直なところ、私の解は複雑すぎて全体的にエレガントではありませんが、方向としてはリチャードの解にとても近いものです。一番大きな問題は、収入の総和を求めることでした。ですが、もし直近のend_yearを求めて1つの解に決めたい場合は、あなたが書いたクエリ（SELECT文）を次のように変える必要があります。なお、このクエリを動かすためには、リチャードの解を使ってPensionsViewビューを作る[訳注3]必要があります」

```
SELECT P0.sin, P0.end_year,
       MAX(P0.start_year) AS laststart_year,
       MIN(P0.sumofearnings) AS minearnings,
       MIN(P0.sumofmonth_cnt) AS minmonth_cnt,
       MIN(P0.start_year) AS firststart_year,
       MAX(P0.sumofearnings) AS maxearnings,
       MAX(P0.sumofmonth_cnt) AS maxmonth_cnt
  FROM PensionsView AS P0
 WHERE end_year = (SELECT MAX(end_year)
                     FROM Pensions AS P1
                    WHERE P1.sin = P0.sin)
 GROUP BY P0.sin, P0.end_year;
```

訳注1：サイベースのRDBMS製品「SQL Anywhere Studio」の旧称。

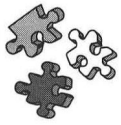
訳注2：WATCOM SQLのオプティマイザ。

訳注3：SELECT句に「SUM(P1.month_cnt) AS sumofmonth_cnt」を加える必要がある。

パズル 11

作業依頼

HAVING句の力（その1）



ジェンク・エルソイは、CompuServeのGuptaフォーラムにこんな質問を送ってきた。

ある工場において、プロジェクトの内容が作業依頼書に記録されているとしよう。そこには、通過すべき一連の工程が書かれている。step_nbrで管理される各工程は、“完了”か“前工程（複数の場合もある）の完了を待っている”かのいずれかの状態をとる。使用するテーブルは次のとおりだ。

```
CREATE TABLE Projects
(workorder_id CHAR(5) NOT NULL,
 step_nbr INTEGER NOT NULL
     CHECK (step_nbr BETWEEN 0 AND 1000),
 step_status CHAR(1) NOT NULL
     CHECK (step_status IN ('C', 'W')), -- Cは完了、Wは待機
 PRIMARY KEY (workorder_id, step_nbr));
```

Projects：プロジェクトテーブル	workorder_id：作業依頼ID
step_nbr：工程番号	step_status：工程ステータス

サンプルデータは、例えば次のようになる。

```
Projects

workorder_id    step_nbr    step_status
=======================================
'AA100'                0    'C'
'AA100'                1    'W'
'AA100'                2    'W'
'AA200'                0    'W'
'AA200'                1    'W'
'AA300'                0    'C'
'AA300'                1    'C'
```

彼の要望は、step_nbrが0（工程0番）で、かつstep_statusが 'C' であり、さらにその作業（同じworkorder_id）内で0番を除く全工程のstep_statusが 'W' である依頼 —— 要するに0番の工程のみ完了している依頼 —— を取り出すことだった。上のサンプルなら 'AA100' だけが答えになる。

その1

答え したいことは実に明快であるが、答えを見つけるためには、このクエリの仕様の1つ「その作業では、0番を除く全工程のstep_statusが 'W' である」を、主語と目的語を入れ替えて考える必要がある。「その作業では、'W' が0番を除く全工程のstep_statusである」と考えれば、答えはすぐに出てくる。

```
SELECT workorder_id
  FROM Projects AS P1
 WHERE step_nbr = 0
   AND step_status = 'C'
   AND 'W' = ALL (SELECT step_status
                    FROM Projects AS P2
                   WHERE step_nbr <> 0
                     AND P1.workorder_id = P2.workorder_id);
```

その2

答え 別の考え方をしてみよう。「特定のstep_nbrについてstep_statusがある性質を持っている作業依頼の集合（= step_nbrの集合）を探している」と考えるのだ。そうした場合、SUM関数の中で特性関数[訳注1]を使うことで、集合の全要素が仕様の条件を満たすか否かを知ることができる。もし条件を満たすのであれば、特性関数の戻り値を合計した値が、集合の要素数と一致するはずだ。

```
SELECT workorder_id
  FROM Projects
 GROUP BY workorder_id
HAVING SUM(CASE WHEN step_nbr <> 0 AND step_status = 'W' THEN 1
                WHEN step_nbr = 0  AND step_status = 'C' THEN 1
                ELSE 0 END) = COUNT(step_nbr);
```

CASE式が使えない環境でも、ちょっとした計算をすれば同じことができる[訳注2]。

訳注1：特性関数（characteristic function）とは、ある集合の要素がその集合の特定の部分集合に含まれるかどうかを決定する関数で、含まれるなら1を、含まれないなら0を返す。その部分集合を定めるという意味で定義関数とも呼ばれる。

訳注2：SIGN(step_nbr)は、step_nbrが0の場合のみ0を、ほかの場合は1を返す。POSITION(文字列1, 文字列2)は、文字列2の中での文字列1の場所を表す数値を返す。この場合は、step_statusが 'W' か 'C' かの判定に使っている。

```
SELECT workorder_id
  FROM Projects AS P1
 GROUP BY workorder_id
HAVING SUM(SIGN(step_nbr) * POSITION('W' IN step_status)
        + (1 - SIGN(step_nbr)) * POSITION('C' IN step_status))
       = COUNT(step_nbr);
```

先のクエリはテーブルのコピーを1つしか持たず、スキャンは1回で済むため、処理速度も最高になるはずだ。また、CASE式には処理を最適化するための細工を施してある。CASE式のWHEN句は書かれた順番でテストされるので、最もマッチしそうな条件を最初のほうに並べるのだ。

標準SQLでは要求されていないが、ANDが結んでいる項がすべて結合条件（2つのテーブルの列を結ぶ）であるか、あるいはすべて検索引数（1つのテーブルの列を定数と比較する —— Search ARGumentsを略してSARGとも呼ぶ）である場合、書かれた順で実行されることがやはり多い。したがって、SARGの初めに最小サイズのデータ型を扱う条件を置くことで、パフォーマンスを改善できる。例えば、整数の比較は大きなCHAR(n)型データの比較よりも速い。

その3

答え

答えその2の方法では、サブクエリを使っていない。コロンビアのフランシスコ・モレノは、その別バージョンを送ってくれた。オリジナルのクエリはOracleのDECODE関数を使っていたが、ここではSQL-92での書き方に直したものを紹介する。

```
SELECT workorder_id
  FROM Projects
 GROUP BY workorder_id
HAVING COUNT(*)      -- あるworkorder_idについての全行数
       = COUNT(CASE WHEN step_nbr = 0 AND step_status = 'C'
                    THEN 1
                    ELSE NULL END)  -- 0番で「完了」の行数
       + COUNT(CASE WHEN step_nbr <> 0 AND step_status = 'W'
                    THEN 1
                    ELSE NULL END); -- 0番以外で「待機」の行数
```

ここでは、比較演算の左辺をCOUNT(*)だけにして、式の中に入れないようにしている。こう書くことで、パフォーマンスも少し向上するだろう。

私の生徒の1人であるステファン・グナイストは、こんなに簡単で素晴らしい解を見つけた[訳注3]。

```
SELECT workorder_id
  FROM Projects
 WHERE step_status = 'C'
 GROUP BY workorder_id
HAVING SUM(step_nbr) = 0;
```

この解は、テーブル定義のNOT NULL制約とCHECK制約を利用したもので、結合を一切必要としない。私たちはすぐにDML（データ操作言語）にばかり目を向けがちだが、SQLが実はDDL（データ定義言語）とDMLの組み合わせなのだということを例証した名答だ。

訳注3：もし、作業依頼でグルーピングした集合の中に、作業状態が'C'である行が1つもなければ、SUM(step_nbr)は0ではなくNULLを返す。SUM(step_nbr)が0を返すのは、step_nbr=0である行の作業状態が'C'で、残りの行の作業状態がすべて'W'の場合だけである。

パ|ズ|ル|
12

訴訟の進行状態

最大値の集合からその最小値を取り出す

レオナルド・C・メダルは、CompuServeにこんな問題を送ってきた。

いま病院に対して、患者たちが訴訟を起こしているとしよう。それを記録しているのがClaimsテーブルだ。1つの訴訟には、複数の被告人（普通は医師）が存在し得る。

```
Claims

claim_id   patient_name
=======================
      10   'Smith'
      20   'Johns'
      30   'Brown'
```

Claims：訴訟テーブル	claim_id：訴訟ID	patient_name：患者名

被告人は、Defendantsテーブルに記録される。

```
Defendants

claim_id   defendant_name
=========================
      10   'Johnson'
      10   'Mayer'
      10   'Dow'
      20   'Baker'
      20   'Mayer'
      30   'Johnson'
```

Defendants：被告テーブル	claim_id：訴訟ID	defendant_name：被告名

被告人は、今までに起きた訴訟事件の履歴と結び付けられており、訴訟の進行状態は次のように記録されている。

```
LegalEvents

claim_id   defendant_name   claim_status   change_date
=========================================================
      10   'Johnson'        'AP'           '1994-01-01'
      10   'Johnson'        'OR'           '1994-02-01'
      10   'Johnson'        'SF'           '1994-03-01'
      10   'Johnson'        'CL'           '1994-04-01'
      10   'Mayer'          'AP'           '1994-01-01'
      10   'Mayer'          'OR'           '1994-02-01'
      10   'Mayer'          'SF'           '1994-03-01'
      10   'Dow'            'AP'           '1994-01-01'
      10   'Dow'            'OR'           '1994-02-01'
      20   'Mayer'          'AP'           '1994-01-01'
      20   'Mayer'          'OR'           '1994-02-01'
      20   'Baker'          'AP'           '1994-01-01'
      30   'Johnson'        'AP'           '1994-01-01'
```

LegalEvents：訴訟事件テーブル　claim_id：訴訟ID　defendant_name：被告名
claim_status：訴訟状態　change_date：更新日

訴訟の状態は、法律で定められた一定の順序で変化する。その順序を示すのが次のテーブルである。

```
ClaimStatusCodes

claim_status   claim_status_desc          claim_seq
=================================================
'AP'           'Awaiting review panel'            1
'OR'           'Panel opinion rendered'           2
'SF'           'Suit filed'                       3
'CL'           'Closed'                           4
```

ClaimStatusCodes：訴訟状態コードテーブル　claim_status：訴訟状態
claim_status_desc：説明　claim_seq：連番
'Awaiting review panel'：審理中　'Panel opinion rendered'：評決
'Suit filed'：上訴　'Closed'：結審

ある訴訟における各被告の訴訟状態（claim_status）には一番最近のもの、つまり連番（claim_seq）の最大値を持つ訴訟状態を使う。なお、法律上の理由から、日付でソートした場合の訴訟の進行と連番でソートした場合の進行は、一致するとは限らない。

また、訴訟自体の進行状態（claim_status）には、その訴訟に関与している被告全員の訴訟状態のうち、連番が最小のものを使う。要するに、最大値（各被告の訴訟状態）を集めた集合の中から最小値（各訴訟の進行状態）を選ぶというわけだ。上のサンプルデータならば、次のような結果が答えになる。

```
claim_id   patient_name   claim_status
=======================================
      10   'Smith'         'OR'
      20   'Johns'         'AP'
      30   'Brown'         'AP'
```

あなたには各訴訟の進行状態を取得し、それを表示してもらいたい。

その1

答え　メダルの答えは問題文の条件をコードに直訳したクエリで、1文で書かれている。

```
SELECT C1.claim_id, C1.patient_name, S1.claim_status
  FROM claims AS C1, ClaimStatusCodes AS S1
 WHERE S1.claim_seq IN
           (SELECT MIN(S2.claim_seq)
              FROM ClaimStatusCodes AS S2
             WHERE S2.claim_seq IN
                       (SELECT MAX(S3.claim_seq)
                          FROM LegalEvents AS E1,
                               ClaimStatusCodes AS S3
                         WHERE E1.claim_status = S3.claim_status
                           AND E1.claim_id = C1.claim_id
                         GROUP BY E1.defendant_name));
```

その2

答え 以下に示す解答は、フランシスコ・モレノから寄せられた。SQL-92構文の結合を使うことによって、サブクエリを使わないようにしたものだ。

求める結果を得るためのクエリは次のとおり。

```
SELECT C1.claim_id, C1.patient_name,
       CASE MIN(S1.claim_seq) WHEN 2 THEN 'AP'
                              WHEN 3 THEN 'OR'
                              WHEN 4 THEN 'SF'
                              ELSE 'C1' END
  FROM ((Claims AS C1
         INNER JOIN
         Defendants AS D1
         ON C1.claim_id = D1.claim_id)
        CROSS JOIN
        ClaimStatusCodes AS S1)
       LEFT OUTER JOIN
       LegalEvents AS E1
       ON C1.claim_id  = E1.claim_id
          AND D1.defendant_name = E1.defendant_name
          AND S1.claim_status = E1.Claim_status
 WHERE E1.claim_id IS NULL
 GROUP BY C1.claim_id, C1.patient_name;
```

パ|ズ|ル|
13

2人かそれ以上か、それが問題だ

CASE式の中に集約関数を組み込む（その1）

ブレンダン・キャンベルは1996年の5月、CompuServeのOracleユーザのグループフォーラムに面白い問題を投稿してきた。彼はこの問題を本に載せる許可をくれるだけでなく、彼が考えた常識外れなPL/SQLの解を、悪い見本として載せることにも承諾してくれた。つまり、科学の発展のために、自らの恥と不名誉を世界に供してくれたのだ。献体をするのは簡単だ —— もう死んでいるのだから。本当に難しいのは、自らの尊厳を差し出すことだ。だが、ここでは解を公開してよいという、彼の勇気をたたえるだけにとどめておこう。

このパズルで私たちが考えるのは、レポート作成プログラムへ結果を渡すために、講座とそれを受講している生徒、それに教師の名前（1人とは限らない）を表示するクエリである。それだけなら何ということはないが、難点が1つある。レポート用紙には、教師の名前を印刷するスペースが2人分しかないのだ。

もし教師が1人しかいなかった場合には、その教師名を1番目のteacher_name列に表示し、2番目の列は空白かNULLにする。教師がちょうど2人いた場合には、両方の名前を昇順で表示する。教師が3人以上いた場合には、1人目の教師だけを最初の列に表示して、2番目の列には '--More--' と表示する。

必要なデータは、次のテーブルに含まれている。

```
CREATE TABLE Register
(course_nbr INTEGER NOT NULL,
 student_name CHAR(10) NOT NULL,
 teacher_name CHAR(10) NOT NULL,
 ...);
```

Register：受講登録テーブル	course_nbr：講座番号
student_name：生徒名	teacher_name：教師名

ブレンダンが考えたオリジナルの解は長さが70行もあったが、純粋なSQLでの答えは12行程度の1文で済む。

その1

答え 1つの方法は、極値関数[訳注1]とUNIONを使うことだ。

```
SELECT R1.course_nbr, R1.student_name, MIN(R1.teacher_name), NULL
  FROM Register AS R1
 GROUP BY R1.course_nbr, R1.student_name
HAVING COUNT(*) = 1
UNION
SELECT R1.course_nbr, R1.student_name,
       MIN(R1.teacher_name), MAX(R1.teacher_name)
  FROM Register AS R1
 GROUP BY R1.course_nbr, R1.student_name
HAVING COUNT(*) = 2
UNION
SELECT R1.course_nbr, R1.student_name,
       MIN(R1.teacher_name), '--More--'
  FROM Register AS R1
 GROUP BY R1.course_nbr, R1.student_name
HAVING COUNT(*) > 2;
```

それでは、例によって詳しく見ていこう。

最初のSELECT文は、教師が1人だけという場合のcourse_nbrとstudent_nameの組み合わせを取り出している。ではなぜ、MIN関数でうまくいくのか分かるだろうか。

それは、1人しかいないのなら、その教師名がデフォルトで最小値になるからだ。私は失われた値としてNULLを使うのが好きだが、文字列定数を使ってもかまわない。

2番目のSELECT文は、教師がちょうど2人という場合のcourse_nbrとstudent_nameの組み合わせを取り出している。2人しかいないので、MIN関数とMAX関数で名前を昇順に並べることができる。

3番目のSELECT文は、教師が3人以上いる場合のcourse_nbrとstudent_nameの組み合わせを取り出している。最初の教師名を取得するためにMIN関数を使い、2番目の列にはレポートの仕様に従って定数'--More--'を指定している。

訳注1：MAX関数とMIN関数のこと。

その2

答え　リチャード・レムレーは、私が公表した解をうまく料理してくれた。SQL-92にあるCASE式の中に極値関数を入れる構文を利用して、後半の2つのSELECT文を最初のSELECT文の中に折り込んだのだ。

```
SELECT course_nbr, student_name, MIN(teacher_name),
       CASE COUNT(*) WHEN 1 THEN NULL
                     WHEN 2 THEN MAX(teacher_name)
                     ELSE '--More--' END
  FROM Register
 GROUP BY course_nbr, student_name;
```

この単純CASE式は、次のような検索CASE式で書き換えることもできる。

```
CASE WHEN COUNT(*) = 1 THEN NULL
     WHEN COUNT(*) = 2 THEN MAX(teacher_name)
     ELSE '--More--'
END
```

さらに次の解を見ると、条件により結果の表示の仕方を変える問題では、SELECT句のリスト中で使えるCASE式が本当に便利だということが分かるだろう。

その3

答え　そもそも、作成しようとしているレポートのレイアウト自体に問題がある――本来、教師の名前は全員分リストアップされたほうがよい。course_nbrとstudent_nameも、同じ値が繰り返し出力されないようにしたい。つまり、それらの列で同じ値がまた出てきたときには、そこは空白にしておくべきなのだ。COBOLやほかの帳票作成用プログラムでは簡単に書けるが、SQL-92では次ページに示すような形になる。

```
SELECT CASE WHEN teacher_name
              = (SELECT MIN(teacher_name)
                   FROM Register AS R1
                  WHERE R1.course_nbr = R0.course_nbr
                    AND R1.student_name = R0.student_name)
            THEN course_nbr
            ELSE NULL END AS course_nbr_hdr,
       CASE WHEN teacher_name
              = (SELECT MIN(teacher_name)
                   FROM Register AS R1
                  WHERE R1.course_nbr = R0.course_nbr
                    AND R1.student_name = R0.student_name)
            THEN student_name
            ELSE NULL END AS student_name_hdr,
       teacher_name
  FROM Register AS R0
 ORDER BY course_nbr, student_name, teacher_name;
```

ただ、やはりこのコードはあまり誉められたものではない。結果が第1正規形を満たさないからだ。これは、フロントエンドのプログラムですべきことをデータベースでやってしまった悪い例である。

パ|ズ|ル|
14

電話とFAX

外部結合の上手な使いこなし

今、あなたは自分のオフィスで新しいデータベース印刷システムを使い、社員の自宅の電話番号を一覧表にして打ち出そうとしていると想像してほしい。使用するテーブルは以下のものである。

```
CREATE TABLE Personnel
(emp_id INTEGER PRIMARY KEY,
 first_name CHAR(20) NOT NULL,
 last_name CHAR(20) NOT NULL);
```

Personnel：社員テーブル　　emp_id：社員ID　　first_name：名　last_name：姓

```
CREATE TABLE Phones
(emp_id INTEGER NOT NULL,
 phone_type CHAR(3) NOT NULL
    CHECK (phone_type IN ('hom', 'fax')),
 phone_nbr CHAR(12) NOT NULL,
 PRIMARY KEY (emp_id, phone_type),
 FOREIGN KEY (emp_id) REFERENCES Personnel(emp_id));
```

Phones：電話テーブル　　emp_id：社員ID　　phone_type：電話タイプ
phone_nbr：電話番号

phone_type列の'hom'と'fax'というコードは、番号が電話番号かFAX番号かを表す。

あなたは各社員の姓名と電話番号、FAX番号を1行にまとめて全員分印刷したいと考えている。番号が分からない場合にはNULLを表示する。電話番号とFAX番号ともに不明ならば、両方の番号列がNULLになる。

なお、念のため言っておくと、Phonesテーブルの外部キー制約は、社員以外の番号を表示しないためのものだ。

その1

答え この問題には、考えるべきことがたくさんある。最初に思いつく方法は、自宅の電話に関する情報を単独で作ることである。情報は全社員を網羅したいので、外部結合が必要だ。

```
CREATE VIEW HomePhones
(last_name, first_name, emp_id, home_phone)
AS SELECT E1.last_name, E1.first_name, E1.emp_id, H1.phone_nbr
     FROM (Personnel AS E1 LEFT OUTER JOIN Phones AS H1
             ON E1.emp_id = H1.emp_id
            AND H1.phone_type = 'hom');
```

これと同様に、FAXの情報についてもビューを作ることができるだろう。

```
CREATE VIEW FaxPhones (last_name, first_name, emp_id, fax_phone)
AS SELECT E1.last_name, E1.first_name, E1.emp_id, F1.phone_nbr
     FROM (Personnel AS E1 LEFT OUTER JOIN Phones AS F1
              ON E1.emp_id = F1.emp_id
             AND F1.phone_type = 'fax');
```

あとは、これら2つのビューを結合するだけだ。

```
SELECT H1.last_name, H1.first_name, home_phone, fax_phone
  FROM HomePhones AS H1, FaxPhones AS F1
 WHERE H1.emp_id = F1.emp_id;
```

また、こういう問題にはCOALESCE関数が役に立つ。この関数は引数の式のリストを左から右へ調べていき、最初に見つかったNULLでない値を返してくれる。

```
SELECT COALESCE(H1.last_name, F1.last_name),
       COALESCE(H1.first_name, F1.first_name),
       home_phone, fax_phone
  FROM HomePhones AS H1 FULL OUTER JOIN FaxPhones AS F1
    ON H1.emp_id = F1.emp_id;
```

その2

答え

答えその1のクエリは正しく動くのだが、欠点が1つある。このクエリを実行する前におそらく2つのビューを実体化することになるため、処理にとても時間がかかるのだ。そこで、一歩退いて答えその1のクエリをよく観察してみる。すると、FaxPhonesとHomePhonesの2つのビューは、どちらもPersonnelテーブルと外部結合されていることが分かる。それならば、これら2つのビューからPersonnelテーブルを抜き出して、FROM句で2つのテーブルと結合しよう。

```
SELECT E1.last_name, E1.first_name, H1.phone_nbr AS Home,
       F1.phone_nbr AS FAX
  FROM (Personnel AS E1 LEFT OUTER JOIN Phones AS H1
          ON E1.emp_id = H1.emp_id
         AND H1.phone_type = 'hom')
       LEFT OUTER JOIN Phones AS F1
            ON E1.emp_id = F1.emp_id
           AND F1.phone_type = 'fax';
```

このクエリでは、3つのテーブルを一度に取り出すので、答えその1のクエリよりも多少は速くなるはずだ。なお、Sybase、Oracle、Guptaの独自拡張の構文では外部結合のネストを扱えないため、このクエリは簡単には書けない。

その3

答え

リズ・クレイボーン・コスメティクス社のデータベースアナリストであるキショーレ・ガンジーは、2つのビューを使わない解法を提案してきた。彼の答えはOracleの構文を使っていたが、それをSQL-92構文に直したものが次のクエリである。

```
SELECT E1.emp_id, E1.first_name, E1.last_name,
       MAX(CASE WHEN P1.phone_type = 'hom'
                THEN P1.phone_nbr
           ELSE NULL END) AS home_phone,
       MAX(CASE WHEN P1.phone_type = 'fax'
                THEN P1.phone_nbr
           ELSE NULL END) AS fax_phone
  FROM Personnel AS E1 LEFT OUTER JOIN Phones AS P1
    ON P1.emp_id = E1.emp_id
 GROUP BY E1.emp_id, E1.first_name, E1.last_name;
```

CASE式では、電話かFAXかに応じて番号を正しい列に配置している。その後に、MAX関数とGROUP BY句を使って1行にまとめるという流れだ。

その4

答え 最後の解は、コロンビアのフランシスコ・モレノから届けられた。

```
SELECT P1.last_name, P1.first_name,
       (SELECT T1.phone_nbr
          FROM Phones AS T1
         WHERE T1.emp_id = P1.emp_id
           AND T1.phone_type = 'hom') AS home_phone,
       (SELECT T2.phone_nbr
          FROM Phones AS T2
         WHERE T2.emp_id = P1.emp_id
           AND T2.phone_type = 'fax') AS fax_phone
  FROM Personnel AS P1;
```

これも正解だが、スカラサブクエリ[訳注1]は、あまり速くは動かないだろう。

訳注1：名前のとおりスカラ値、すなわち単一の値を返すサブクエリ。条件に一致する行がなかった場合にはNULLを返すため、この解答のように外部結合の代用として広く利用されている。

パ|ズ|ル|
15

現在の給料と昇給前の給料

極値関数（MAX/MIN）の一般化

ジャック・ウェルズがこのややこしい問題をCompuServeに送ってきたのは、1996年の7月のことである。ジャックが当時置かれていた状況をひとことで言えば、「3GL[訳注1]を使う人々と一緒に働かなくてはならないSQLプログラマ」だった。彼らは社員情報の帳票作成に従事しており、その仕事の一環として、各社員の「現在の給料」と「昇給前（現在の額になる直前）の給料」についての情報を得たいと思っていた。そのレポートには、社員の昇給日と給料を記載する必要がある。

もし、給料と昇給日を単純に1行ずつ結果セットへ出力し、それをホストプログラムで適宜フォーマットするというのなら、話はとても簡単だ。実際、皆さんが初めて書くプログラムとは、このような処理を行うものではないだろうか。

だが残念なことに、言い添えておかねばならないことがある。実は、アプリケーションプログラマたちがどうしようもない怠け者ぞろいで、「社員ごとに、現在と昇給前の給料および昇給日を1行で返すようにSQL文を書いてくれ」と言ってきたのだ。あとはただ1行ずつカーソルで読み出し、印刷するだけでレポートはでき上がる。彼らは実作業と呼べるような仕事を何もしなくて済むわけだ。

ジャックは、この問題をフェビアン・パスカル（www.dbdebunk.comを主宰）にも相談した[訳注2]。パスカル氏の返答は「そんなクエリは書けない」というものだった。「真のリレーショナル言語ならば可能かもしれない。しかし、SQLはそうではないので不可能だ。たとえSQL-92でもできない」

言ってくれるじゃないか。こんな話を聞くと、ふつふつと闘志が湧いてきてしまう！

そうそう、もう1つ言い忘れていたことがある。この問題を考えるにあたって、1つ制限があった。それは、ジャックがOracleを使っていることだ。Oracleは当時、SQL-92に対応していなかった（正しい外部結合の構文もなく、一般的なスカラサブクエリ式なども使えなかった）。そのため、クエリはSQL-89に準拠したものであることとする。

テストデータには、次のものを使おう。

訳注1：第3世代言語（3rd Generation Language）。手続き型言語一般を指す。

訳注2：フェビアン・パスカル氏は、米国で活躍するルーマニア生まれのシステムコンサルタント。関係モデルの理論的厳密さにこだわり、ベンダや商業誌の商業主義を告発する辛口批評で知られる。著書に『Practical Issues in Database Management』（Addison-Wesley刊）など。

```
CREATE TABLE Salaries
(emp_name CHAR(10) NOT NULL,
 sal_date DATE NOT NULL,
 sal_amt DECIMAL(8,2) NOT NULL,
 PRIMARY KEY (emp_name, sal_date));
```

Salaries：給料テーブル	emp_name：社員名	sal_date：昇給日	sal_amt：給料

```
INSERT INTO Salaries
VALUES ('Tom',   '1996-06-20', 500.00),
       ('Tom',   '1996-08-20', 700.00),
       ('Tom',   '1996-10-20', 800.00),
       ('Tom',   '1996-12-20', 900.00),
       ('Dick',  '1996-06-20', 500.00),
       ('Harry', '1996-07-20', 500.00),
       ('Harry', '1996-09-20', 700.00);
```

Tomには3回、Harryには1回の昇給があり、Dickはまだ昇給のない新米である。

その1

答え 手始めに、簡単な問題から考えてみよう。この解答では、私が個人的に「一般極値関数（generalized extrema）」とか「top(n)関数」と呼んでいるクエリを使う。そのクエリをビューにすると、次のようになる。

```
CREATE VIEW Salaries2 (emp_name, sal_date, sal_amt)
AS SELECT S0.emp_name, S0.sal_date, MAX(S0.sal_amt)
     FROM Salaries AS S0, Salaries AS S1
    WHERE S0.sal_date <= S1.sal_date
      AND S0.emp_name = S1.emp_name
    GROUP BY S0.emp_name, S0.sal_date
   HAVING COUNT(*) <= 2;

emp_name  sal_date      sal_amt
==============================
'Dick'    '1996-06-20'   500.00
'Harry'   '1996-07-20'   500.00
'Harry'   '1996-09-20'   700.00
'Tom'     '1996-10-20'   800.00
'Tom'     '1996-12-20'   900.00
```

Salariesテーブルのコピー S1 は、各社員について昇給が2回以下という部分集合を作るのに使われている。MAX関数は、そこからsal_amt列の値を取り出すための“トリック”である。こうして、1回の昇給につき1行の形で直近2回の昇給日と給料を社員ごとに出力できる。アプリケーションプログラマがそれほど怠け者でないなら、このテーブルを渡せば、レポート作成のためのフォーマットはアプリケーション側でしてくれるだろう。

その2

答え

実際に解くべき問題は、もっと難しい。これをSQL-89の範囲内で記述するための1つの方法は、問題を2つのケースに分けることだ。

1. 1回だけ給料の設定があった社員
2. 2回以上給料の設定があった社員

すべての社員が、このどちらか一方のケースにのみ該当することは明らかだ。そこで、これら2つの集合をUNIONでつなげる方法が考えられる。

```
SELECT S0.emp_name, S0.sal_date, S0.sal_amt, S1.sal_date,
       S1.sal_amt
  FROM Salaries AS S0, Salaries AS S1
 WHERE S0.emp_name = S1.emp_name
   AND S0.sal_date =
           (SELECT MAX(S2.sal_date)
              FROM Salaries AS S2
             WHERE S0.emp_name = S2.emp_name)
   AND S1.sal_date =
           (SELECT MAX(S3.sal_date)
              FROM Salaries AS S3
             WHERE S0.emp_name = S3.emp_name
               AND S3.sal_date < S0.sal_date)
UNION ALL
SELECT S4.emp_name, MAX(S4.sal_date), MAX(S4.sal_amt),
       NULL, NULL
  FROM Salaries AS S4
 GROUP BY S4.emp_name
HAVING COUNT(*) = 1;
```

```
emp_name   sal_date       sal_amt   sal_date       sal_amt
=========================================================
'Tom'      '1996-12-20'    900.00   '1996-10-20'    800.00
'Harry'    '1996-09-20'    700.00   '1996-07-20'    500.00
'Dick'     '1996-06-20'    500.00    NULL            NULL
```

DB2プログラマなら、これがSQL-92標準の外部結合演算子がDB2に実装されていなかったころに、外部結合を実現するために使われていた方法の1つであることに気がつくだろう。最初のSELECT文が最も難しい。Salariesテーブルに対して自己結合を行っているのだが、コピーS0は現在の給料の情報源であり、コピーS1は直前の給料の情報源である。2つ目のSELECT文は、テーブルに1行しか存在しない社員を取得する単純な集約クエリである。2つの結果集合には共通部分がないので、UNIONの代わりにUNION ALLを使って余計なソートを省くことができる。

その3

答え もっと良い解を見つけてほしいという私の挑戦に対して、いくつかの返事があった。スミス・バーニー社のリチャード・レムレーは、SQL-92に基づく解法を送ってきた。この解ではビューを使わずに済むよう、サブクエリテーブル式をうまく利用している。共通表式[訳注3]をサポートする製品なら、AとBのサブクエリはビューに変換できる。

```
SELECT B.emp_name, B.maxdate, Y.sal_amt, B.maxdate2, Z.sal_amt
  FROM (SELECT A.emp_name, A.maxdate, MAX(X.sal_date) AS maxdate2
          FROM (SELECT W.emp_name, MAX(W.sal_date) AS maxdate
                  FROM Salaries AS W
                 GROUP BY W.emp_name) AS A
            LEFT OUTER JOIN Salaries AS X
              ON A.emp_name = X.emp_name
             AND A.maxdate > X.sal_date
        GROUP BY A.emp_name, A.maxdate) AS B
    LEFT OUTER JOIN Salaries AS Y
        ON B.emp_name = Y.emp_name AND B.maxdate = Y.sal_date
    LEFT OUTER JOIN Salaries AS Z
        ON B.emp_name = Z.emp_name AND B.maxdate2 = Z.sal_date;
```

訳注3：共通表式（Common Table Expression）はSQL-99で標準化された機能で、FROM句のインラインビューをWITH句を使って「外出し」にする技術。何度も参照されるインラインビューを表通表式にすることで、コードを簡潔にできる。具体的な書式は、本パズルの答えその7にあるsalaryRanksビューの定義を参照。

その4

答え　マイク・コンウェイは、Oracleを使った解答を思いついた。私はそれをSQL-92構文に直そうとしたのだが、いまひとつ中途半端な結果に終わってしまった。難しかったのは、当時のOracleが標準の外部結合構文をサポートしていなかったため、正しい結果を得るには結合を実行する順番に注意が必要だったことである。オラクル社のアソシエイトアプリケーションエンジニアであるシェド・カディールは、答えその1で作ったビューを使って、私の解法を改良してくれた。

```
SELECT S1.emp_name, S1.sal_date, S1.sal_amt, S2.sal_date,
       S2.sal_amt
  FROM Salaries2 AS S1, salaries2 AS S2 -- ビューを使う
 WHERE S1.emp_name = S2.emp_name
   AND S1.sal_date > S2.sal_date
UNION ALL
SELECT emp_name, MAX(sal_date), MAX(sal_amt), NULL, NULL
  FROM Salaries2
 GROUP BY emp_name
HAVING COUNT(*) = 1;
```

もしかしたら、RDBMSによってはUNION ALLを使うにあたって、SELECT句の2つのNULLをCAST(NULL AS DATE)とCAST(NULL AS DECIMAL(8,2))に変え、列のデータ型を厳密にそろえる必要があるかもしれないので、注意してほしい。

その5

答え　ジャックは、WWW.dbdebunk.comに紹介のあるクリス・デイトの著書で定義されている関係代数演算子を使った解答を考えついた[訳注4]。しかし、ここで紹介するのはやめておく。理由は、①オリジナルの問題には「Oracleで動く解」という制約がある、②その関係代数を実装したRDBMSがない、という2つである。関係代数に基づいた「Tutorial D」という実験的な言語があるのだが、広く利用されているものではない。

訳注4：この問題は、『プログラマのためのSQL 第2版』（ピアソン・エデュケーション刊）でも取り上げられている。

また、この解には紛らわしいデータを作るという問題点があった。直前の給料のレコードを持たないすべての社員に対し、直前の給料として0.00を、日付として'1900-01-01'を割り当てるのだ。だが、実際には「ゼロ」と「値がない」ということは論理的に異なるし、宇宙が1900年に始まったわけでもない。

フェビアン・パスカルは、この解答について次のようにコメントした。「ずいぶん昔のことだし、正確な状況や、私の回答が適切に表現されていたか、あるいは（特にセルコによって）理解されていたか、思い出せない。私が推測するに、この手の問題は、クエリを適用するテーブルの詳細な定義や、テーブルとクエリに影響を及ぼすビジネスルールについての情報がないと解くことができないだろう。クリス・デイトに、この解答の感想を聞いてみるとしよう。」

クリス・デイトは、彼が個人的に作った言語[訳注5]を使って、ジャックのよりずっとコンパクトな解を送ってくれた。それについての彼の評価は「冗長で退屈だが、本質的には明快」というものだった。続けて彼は、「セルコの解が正しいかどうか、私は知らないし、知りたいとも思わない」と述べていた。

その解の外部結合を使っていた箇所をCOALESCE関数で代用したのが、次に示す解法だ。これはアンドレイ・オデゴフが送ってくれた。

```
SELECT S1.emp_name, S1.sal_date AS curr_date,
       S1.sal_amt AS curr_amt,
       CASE WHEN S2.sal_date <> S1.sal_date
            THEN S2.sal_date END AS prev_date,
       CASE WHEN S2.sal_date <> S1.sal_date
            THEN S2.sal_amt  END AS prev_amt
  FROM Salaries AS S1 INNER JOIN Salaries AS S2
    ON S2.emp_name = S1.emp_name
   AND S2.sal_date =
         COALESCE((SELECT MAX(S4.sal_date)
                     FROM Salaries AS S4
                    WHERE S4.emp_name = S1.emp_name
                      AND S4.sal_date < S1.sal_date),
                  S2.sal_date)
 WHERE NOT EXISTS(SELECT *
                    FROM Salaries AS S3
                   WHERE S3.emp_name = S1.emp_name
                     AND S3.sal_date > S1.sal_date);
```

訳注5：「個人的な言語」とは、Tutorial Dのことと思われる。この言語の具体的な内容は『C.J.Dateのデータベース実践講義』（オライリー・ジャパン刊）から知ることができる。

その6

答え 給料設定日の全組み合わせを作り、それにフィルタをかけるという方法もある。

```
CREATE VIEW SalaryHistory
(emp_name, curr_date, curr_amt, prev_date, prev_amt)
AS SELECT S0.emp_name,
          S0.sal_date AS curr_date,
          S0.sal_amt AS curr_amt,
          S1.sal_date AS prev_date,
          S1.sal_amt AS prev_amt
     FROM Salaries AS S0
          LEFT OUTER JOIN Salaries AS S1
            ON S0.emp_name = S1.emp_name
           AND S0.sal_date > S1.sal_date;
```

次に、このビューを自己結合のクエリで使う。

```
SELECT S0.emp_name,S0.curr_date, S0.curr_amt,
       S0.prev_date, S0.prev_amt
  FROM SalaryHistory AS S0
 WHERE S0.curr_date
          = (SELECT MAX(curr_date)
               FROM SalaryHistory AS S1
              WHERE S0.emp_name = S1.emp_name)
   AND (S0.prev_date = (SELECT MAX(prev_date)
                          FROM SalaryHistory AS S2
                         WHERE S0.emp_name = S2.emp_name)
                           OR S0.prev_date IS NULL);
```

けっこう複雑な方法だが、SalaryHistoryビューはほかの統計を計算する際にも便利だろう。

その7

答え MarkC600は、SQL Serverのニュースグループでビューを使った異なるアプローチを提案してきた。外部結合をSQL:2003規格にあるRANK関数で代用するのである。次のクエリをよく調べて、考え方のパターンがどう変わるか見てほしい。

```
WITH salaryRanks(emp_name, sal_date, sal_amt, pos)
AS (SELECT emp_name, sal_date, sal_amt,
           RANK() OVER (PARTITION BY emp_name
                            ORDER BY sal_date DESC)
      FROM Salaries)
SELECT C.emp_name, C.sal_date AS curr_date,
       C.sal_amt AS curr_amt,
       P.sal_date AS prev_date, P.sal_amt AS prev_amt
  FROM SalaryRanks AS C LEFT OUTER JOIN SalaryRanks AS P
    ON P.emp_name = C.emp_name
   AND P.pos = 2 WHERE C.pos = 1;
```

その8

答え　もう1つ、SQL:2003で書いた解を紹介しよう。ディーター・ネースから寄せられた、OLAP関数とCASE式を使うものだ。

```
SELECT S1.emp_name,
       MAX(CASE WHEN rn = 1 THEN sal_date ELSE NULL END)
         AS curr_date,
       MAX(CASE WHEN rn = 1 THEN sal_amt ELSE NULL END)
         AS curr_amt,
       MAX(CASE WHEN rn = 2 THEN sal_date ELSE NULL END)
         AS prev_date,
       MAX(CASE WHEN rn = 2 THEN sal_amt ELSE NULL END)
         AS prev_amt
  FROM (SELECT emp_name, sal_date, sal_amt,
               RANK() OVER (PARTITION BY emp_name
                                ORDER BY sal_date DESC)
          FROM Salaries) AS S1 (emp_name, sal_date, sal_amt, rn)
 WHERE rn < 3
 GROUP BY S1.emp_name;
```

1つの考え方として、まず社員ごとに昇給データ（行）を日付によってランク付けし、その中から直近の日付を持つ2行を拾い出すという方法がある。別の考え方として、最初に出力対象となる行をすっかり組み立てておき、あとはそこから実際に欲しい行を見つけるだけ、という方法もある。このクエリでは出力の元となる行を最初に見つけ、最後にそれらをまとめ上げる、という方法をとっている。

テーブルが使われるのは1度だけで、自己結合もないが、RANK関数は裏でソートを必要とすることに注意しよう。もっとも、列データを行単位で記憶領域に格納するSQL

エンジンや、社員名をグループ化するようなインデックスを持っているRDBMSなら、この程度のソートでパフォーマンスに影響は出ないだろう。

その9

答え　ディーター・ネースは、OLAPと共通表式を使う解をもう1つ送ってきてくれた（実行テストはTeradata上で行ったが、SQL Server 2005でも動く）。

```
WITH CTE (emp_name, sal_date, sal_amt, rn)
AS (SELECT emp_name, sal_date, sal_amt ,
           ROW_NUMBER() OVER (PARTITION BY emp_name
                              ORDER BY sal_date DESC) AS rn
                              --行に連番を付与する
      FROM Salaries)
SELECT O.emp_name,
       O.sal_date AS curr_date, O.sal_amt AS curr_amt,
       I.sal_date AS prev_date, I.sal_amt AS prev_amt
  FROM CTE AS O LEFT OUTER JOIN CTE AS I
    ON O.emp_name = I.emp_name
   AND I.rn = 2
 WHERE O.rn = 1;
```

最後に、Teradataが持つSQL:2003のOLAP関数を使う方法も紹介しておこう。

```
SELECT emp_name, curr_date, curr_amt, prev_date, prev_amt
  FROM (SELECT emp_name, sal_date AS curr_date,
               sal_amt AS curr_amt,
               MIN(sal_date) OVER (PARTITION BY emp_name
                                   ORDER BY sal_date DESC
                                   ROWS BETWEEN 1 FOLLOWING
                                   AND 1 FOLLOWING) AS prev_date,
               MIN(sal_amt)  OVER (PARTITION BY emp_name
                                   ORDER BY sal_date DESC
                                   ROWS BETWEEN 1 FOLLOWING
                                   AND 1 FOLLOWING) AS prev_amt,
               ROW_NUMBER() OVER (PARTITION BY emp_name
                                  ORDER BY sal_date DESC) AS rn
          FROM Salaries) AS DT
  WHERE rn = 1;
```

パ|ズ|ル|16

主任とアシスタント

参照整合性制約の正しい設定

ARIのジェラルド・メンコーは、1994年4月にこの問題をCompuServeに投稿した。そのとき、ARIでは、データベースをParadoxからWATCOM SQL（現在はSybaseに統合）に変えたところだった。旧データベースからのデータ移行は、Paradoxのテーブルを1つ1つWATCOM SQLのテーブルに移すというやり方で行われた —— 正規化や整合性ルールなどは一切考慮されず、単純に列名とデータ型をコピーしたのだ。ああ、私はSQLの導師として、正規化をしない者には地獄めぐりが待ち受けていることを、彼に教えてやるべきだった。だがすでに時遅し。そして、彼らがやらかしたのと同じ過ちは、時と場所を問わず、世界中の至るところで犯されている。

そのシステムは、いくつかの仕事に就いている社員とその所属チームを調べるためのものだった。1つの仕事には社員から主任技師が1人と、アシスタント技師が1人割り当てられる。使うテーブルは次のものである。

```
CREATE TABLE Jobs
(job_id INTEGER NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 ... );
```

Jobs：仕事テーブル　　job_id：仕事ID　　start_date：開始日

```
CREATE TABLE Personnel
(emp_id INTEGER NOT NULL PRIMARY KEY,
 emp_name CHAR(20) NOT NULL,
 ... );
```

Personnel：社員テーブル　　emp_id：社員ID　　emp_name：社員名

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL,
 mech_type INTEGER NOT NULL,
 emp_id INTEGER NOT NULL,
 ... );
```

Teams：チームテーブル	job_id：仕事ID	mech_type：技師タイプ
emp_id：社員ID		

あなたにまずやってほしいのは、Teamsテーブルに整合性チェックを追加することだ。これについては、正規化やほかのテーブルは気にしなくてよい。

次に、job_idで識別されるすべての仕事について、(もしいれば) 主任技師 (Primary) とアシスタント技師 (Assistant) をリストアップするクエリを作ってもらいたい。ヒントは、Jobsテーブルからはすべての仕事を選択できるが、Teamsテーブルからはチームの割り当てられた仕事しか選択できないことである。また、1人の人物が、ある仕事の主任技師とアシスタント技師を兼務することもあり得る。

その1

答え

まず、参照整合性制約を付与しよう。おそらく、Teamsテーブルは外部キーによってほかのテーブルと関係付けておくべきだろう。どんなときにも、次のようにデータベーススキーマ内のコードで整合性をチェックするのはよい考えだ。

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL REFERENCES Jobs(job_id),
 mech_type CHAR(10) NOT NULL
   CHECK (mech_type IN ('Primary', 'Assistant')),
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id)
 ...);
```

では、次の問題に移ろう。主任技師だけを取り出したいなら、慣れたSQLプログラマはすぐに左外部結合を使おうとするだろう。

```
SELECT Jobs.job_id, Teams.emp_id AS "primary"
  FROM Jobs LEFT OUTER JOIN Teams
             ON Jobs.job_id = Teams.job_id
 WHERE Teams.mech_type = 'Primary';
```

同じような外部結合をPersonnelテーブルとTeamsテーブルを結合するために使うこともできるが、その場合、「チームの各技師について、主任技師とアシスタント技師を見つけるために2つの独立した外部結合を行い、その結果を1つのテーブルにまとめる必

要がある」という問題が出る。やろうと思えばできなくはないが、恐ろしく深いネストになる自己外部結合を1つのSELECT文で書くことになる。多分、理解できるコードにはならないし、それ以前に読む気も起きないだろう。

ビューを使って同じレポートを作ることもできるが、次のクエリを使えば、そんな手間は全くかからない。

```
SELECT Jobs.job_id,
       (SELECT emp_id
          FROM Teams
         WHERE Jobs.job_id = Teams.job_id
           AND Teams.mech_type = 'Primary') AS "primary",
       (SELECT emp_id
          FROM Teams
         WHERE Jobs.job_id = Teams.job_id
           AND Teams.mech_type = 'Assistant') AS assistant
  FROM Jobs;
```

ASに続くprimaryがダブルクォーテーションで囲まれているのは、これが主キーを意味するSQL-92の予約語だからだ。ダブルクォーテーションは、囲んだ語が識別子であることを示す。ちなみに、シングルクォーテーションで囲んだ場合には、その語は文字列として扱われる。

このクエリの“トリック”は、外側のSELECT句で、2つの独立したスカラサブクエリを使えることだ。社員の名前を追加したければ、内側のSELECTをちょっと変更するだけでよい。

```
SELECT Jobs.job_id,
       (SELECT emp_name
          FROM Teams, Personnel
         WHERE Jobs.job_id = Teams.job_id
           AND Personnel.emp_id = Teams.emp_id
           AND Teams.mech_type = 'Primary') AS "primary",
       (SELECT emp_name
          FROM Teams, Personnel
         WHERE Jobs.job_id = Teams.job_id
           AND Personnel.emp_id = Teams.emp_id
           AND Teams.mech_type = 'Assistant') AS Assistant
  FROM Jobs;
```

1つの仕事に主任技師とアシスタント技師を兼務している社員がいた場合は、2つの

列に同じ名前が現れる。もし1つの仕事に2人以上の主任技師またはアシスタント技師がいた場合は、当然のことながらエラーになる。主任技師やアシスタント技師がいない仕事では、スカラサブクエリは空の結果を返し、それはNULLになる。このクエリの結果は、外部結合で求めようとした結果と一致する。

その2

答え　米国カリフォルニアのチーコ社で働くスキップ・リースは、Teamsテーブルに次のようなルールを適用したいと考えた。

ルール1：1つのjob_idは、1人または0人の主任技師を持つ
ルール2：1つのjob_idは、1人または0人のアシスタント技師を持つ
ルール3：1つのjob_idは、必ず主任技師またはアシスタント技師を1人以上持つ

ルール3は、「担当のチームメンバーがいない仕事は存在し得ない」という意味だ。言われてみれば、確かにこれは妥当なルールである。

Jobsテーブルに仕事を追加する前には、必ずTeamsテーブルにチーム情報が追加されていなければならない。これは、参照整合性制約を使うことで可能だ。ルール1とルール2は、「job_id」と「mech_type」の2列を主キーとすることで実現できる。そうすれば、同じmech_typeについて、job_idは1つしか登録できない。

```
CREATE TABLE Jobs
(job_id INTEGER NOT NULL PRIMARY KEY REFERENCES Teams (job_id),
 start_date DATE NOT NULL,
 ... );
```

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL,
 mech_type CHAR(10) NOT NULL
    CHECK (mech_type IN ('Primary', 'Assistant')),
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
 PRIMARY KEY (job_id, mech_type));
```

言いにくいことだが、この問題にはちょっとした落とし穴がある。SQL-92では、REFERENCES句は参照されるテーブルのユニークキーまたは主キーを参照しなければならないとされている。つまり、参照は列数もデータ型も並び順も同じ列同士の間で行われ

なければならない。Teamsテーブルについて見ると、(job_id, mech_type) は主キーなのでこれを利用することは可能だが、job_idだけを参照することはできない。

これに対処するには、job_id列に一意制約を付ければよい。

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL UNIQUE,
 mech_type CHAR(10) NOT NULL
    CHECK (mech_type IN ('Primary', 'Assistant')),
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
 PRIMARY KEY (job_id, mech_type));
```

しかし、このテーブルが表現するエンティティ (=仕事) はjob_idで識別されるのだから、むしろjob_idを主キーにして、次のように書いたほうが自然ではないだろうか。

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL PRIMARY KEY,
 mech_type CHAR(10) NOT NULL
    CHECK (mech_type IN ('Primary', 'Assistant')),
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
 UNIQUE (job_id, mech_type));
```

現実的な問題として、RDBMSの主キーはデータの記憶容量やアクセス方法に影響を及ぼすので、その定義を変えるとパフォーマンスにも違いが出るだろう。

だが、ちょっと待て！ この定義はjob_idが一意であることを要求しているため、1つの仕事で「主任」と「アシスタント」を兼務する技師が登録できない。これは問題だ。

その3

答え

主任兼アシスタントの技師を持つというのは、チームの属性だ。したがって、Teamsテーブルの定義をこう修正しよう。

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL REFERENCES Jobs(job_id),
 primary_mech INTEGER NOT NULL REFERENCES Personnel(emp_id),
 assist_mech  INTEGER NOT NULL REFERENCES Personnel(emp_id),
 CONSTRAINT at_least_one_mechanic
   CHECK(COALESCE (primary_mech, assist_mech) IS NOT NULL),
 ...);
```

Teams：チームテーブル	job_id：仕事ID	primary_mech：主任技術者
assist_mech：補佐技術者		

だが、これではまだ不十分だ。仕事を担当できるのは、肩書きのある技師だけに限定したい。

```
CREATE TABLE Personnel
(emp_id INTEGER NOT NULL PRIMARY KEY,
 emp_name CHAR(20) NOT NULL,
 mech_type CHAR(10) NOT NULL
     CHECK (mech_type IN ('Primary', 'Assistant')),
 UNIQUE (emp_id, mech_type),
 ...);
```

Personnel：社員テーブル	emp_id：社員ID	emp_name：社員名
mech_type：技師タイプ		

これに合わせて、Teamsテーブルも変えよう。

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL REFERENCES Jobs(job_id),
 primary_mech INTEGER NOT NULL,
 primary_type CHAR(10) DEFAULT 'Primary' NOT NULL
     CHECK (primary_type = 'Primary'),
 assist_mech INTEGER NOT NULL ,
 assist_type CHAR(10) DEFAULT 'Assistant' NOT NULL
     CHECK (assist_type = 'Assistant') ,
 CONSTRAINT fk_primary FOREIGN KEY (primary_mech, primary_type)
     REFERENCES Personnel(emp_id, mech_type),
 CONSTRAINT fk_assist FOREIGN KEY (assist_mech, assist_type)
     REFERENCES Personnel(emp_id, mech_type),
 CONSTRAINT at_least_one_mechanic
     CHECK(COALESCE (primary_mech, assist_mech) IS NOT NULL)) ;
```

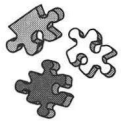
Teams：チームテーブル	job_id：仕事ID	primary_mech：主任技師
primary_type：主任タイプ	assist_mech：アシスタント技師	assist_type：補佐タイプ

これで、ようやくうまくいくはずだ。

パ|ズ|ル|
17

人材紹介会社

関係除算と標準形



ラリー・ウェイドは、1996年2月にMicrosoft Accessフォーラムへこんな問題を投稿してきた。

彼は人材紹介会社の社長で、紹介依頼や、就業希望者とそのスキルについてのデータベースを持っていた。ラリーがしたかったのは、人材紹介を依頼された仕事に対して、それをこなせるスキルを持った就業希望者をこのデータベースから見つけ出すことだった。例えば、「製造と在庫管理または会計のスキルを持った登録者をすべて見つけ出す」という具合だ。なお、紹介依頼は、求める人材のスキルをブール（真偽）式で表した形になっている。

最初に、就業希望者のスキルについてのテーブルを作ろう。なお、彼らの個人情報は別テーブルに保存されていると思ってほしい。この問題では特に気にする必要はない。

```
CREATE TABLE CandidateSkills
(candidate_id INTEGER NOT NULL,
 skill_code CHAR(15) NOT NULL,
 PRIMARY KEY (candidate_id, skill_code));
```

CandidateSkills：就業希望者スキルテーブル　　candidate_id：就業希望者ID
skill_code：スキルコード

```
INSERT INTO CandidateSkills
VALUES (100, 'accounting'),
       (100, 'inventory'),
       (100, 'manufacturing'),
       (200, 'accounting'),
       (200, 'inventory'),
       (300, 'manufacturing'),
       (400, 'inventory'),
       (400, 'manufacturing'),
       (500, 'accounting'),
       (500, 'manufacturing');
```

'accounting'：会計　　'inventory'：在庫管理　　'manufacturing'：製造

解答として分かりやすいのは、フロントエンドのプログラムの中で次のようなSQL文を、人材の紹介依頼ごとに動的に作ることだろう。

```
SELECT DISTINCT C1.candidate_id, 'job_id #212' -- 仕事IDコードの定数
  FROM CandidateSkills AS C1,                  -- スキル1つにつき1テーブル
       CandidateSkills AS C2,
       CandidateSkills AS C3
 WHERE C1.candidate_id = C2.candidate_id
   AND C1.candidate_id = C3.candidate_id
   AND -- 以下で紹介依頼を表す式を作る
       (    C1.skill_code = 'manufacturing'
        AND C2.skill_code = 'inventory'
         OR C3.skill_code = 'accounting');
```

腕のよいプログラマなら、このクエリ用の入力画面を作るのに1週間もかからないと思う。それから、この動的に生成されたSQL文をjob_idコードと同名のビューとして保存する。きれいで手早い答えだ！ ただし、この解法には、膨大な数の非常に遅いクエリを保持しなければならないという難点がある。ほかによいアイデアはないだろうか？

そうそう、言い忘れていたが、ラリーの会社が扱うべき職種は25万を超える。彼の会社は「DOT (Dictionary of Occupational Titles)」という、アメリカ政府が統計用に使う職種のコード体系を利用しているのだ。

その1

答え

職種の多さを気にしなくてよいのなら、問題はぐっと簡単になる。それぞれの職種に必要なスキルをビットで表現することにし、整数値を使ってビットを0または1にセットするのだ。

```
'accounting' = 1        (2進数の001)
'inventory'  = 2        (2進数の010)
'manufacturing' = 4     (2進数の100)
```

この方法では、例えば、('inventory' AND 'manufacturing') は (2 + 4) = 6で表せる。惜しむらくは、25万種の職種が相手では、このやり方が通じないことだ。

また、この解法でまず問題になるのは、検索条件の構文解析をどうするかである。「'manufacturing' AND 'inventory' OR 'accounting'」は、「('manufacturing' AND

'inventory') OR 'accounting'」と「'manufacturing' AND ('inventory' OR 'accounting')」のどちらを意味するのだろう？ ここでは、ANDのほうが優先度が高いと考えることにしよう。

その2

答え 別解は、すべてのクエリを選言標準形に変換することである。選言標準形とは、各条件をANDでつなげた節を、さらにORでつなげた形である[訳注1]。

では、適任者を見つけたい人材紹介依頼を管理するテーブルを作ってみよう。

```
CREATE TABLE JobOrders
(job_id INTEGER NOT NULL,
 skill_group INTEGER NOT NULL,
 skill_code CHAR(15) NOT NULL,
 PRIMARY KEY (job_id, skill_group, skill_code));
```

JobOrders：人材の紹介依頼テーブル	job_id：紹介依頼のあった仕事ID
skill_group：スキルグループ	skill_code：スキルコード

skill_groupコードは、「そのグループに含まれるすべてのスキルが必要とされる」ことを意味している。標準形では、グループ内のスキルはANDで結ばれる。また、ある紹介依頼の中にあるskill_groupは、同じjob_idを持つ別のskill_groupとORで結ばれる。

それでは、次のような一連の依頼を標準形にしてJobOrdersテーブルに登録しよう。

```
Job 1 = ('inventory' AND 'manufacturing') OR 'accounting'
Job 2 = ('inventory' AND 'manufacturing') OR ('accounting'
        AND 'manufacturing')
Job 3 = 'manufacturing'
Job 4 = ('inventory' AND 'manufacturing' AND 'accounting')
```

これらは、次のように変換される。

訳注1：標準形には、この問題で使われる選言標準形（disjunctive canonical form）と裏返しの関係にある連言標準形や、量化まで考慮した冠頭標準形などがある。論理式の形式が規則的なため、今回のように機械的に論理式を処理したいときに便利だ。

```
INSERT INTO JobOrders
VALUES (1, 1, 'inventory'),
       (1, 1, 'manufacturing'),
       (1, 2, 'accounting'),
       (2, 1, 'inventory'),
       (2, 1, 'manufacturing'),
       (2, 2, 'accounting'),
       (2, 2, 'manufacturing'),
       (3, 1, 'manufacturing'),
       (4, 1, 'inventory'),
       (4, 1, 'manufacturing'),
       (4, 1, 'accounting');
```

クエリには、関係除算を利用する[訳注2]。skill_codeとskill_groupの組み合わせが被除数、就業希望者のスキルが除数である。同一のjob_idに含まれるskill_group同士はORで結ばれているため、skill_groupがどれか1つでも一致したjob_idが選択される。

```
SELECT DISTINCT J1.job_id, C1.candidate_id
  FROM JobOrders AS J1 INNER JOIN CandidateSkills AS C1
    ON J1.skill_code = C1.skill_code
 GROUP BY candidate_id, skill_group, job_id
HAVING COUNT(*) >= (SELECT COUNT(*)
                      FROM JobOrders AS J2
                     WHERE J1.skill_group = J2.skill_group
                       AND J1.job_id = J2.job_id);

job_id   candidate_id
=====================
     1            100
     1            200
     1            400
     1            500
     2            100
     2            400
     2            500
     3            100
     3            300
     3            400
     3            500
     4            100
```

訳注2：関係除算については、パズル21で主題として取り上げているので、そちらを参照。

紹介依頼や就業希望者が変わっても、クエリは変わらない。このクエリをビューにすれば、就業希望者のいない仕事や、仕事のない就業希望者を見つけるのにも使える。

その3

答え 別解はスミス・バーニー社のリチャード・レムレーから届いた。彼は、相関サブクエリを使わないSQL-92準拠の解を思いついた。次のようなクエリだ。

```
SELECT J1.job_id, C1.candidate_id
  FROM (SELECT job_id, skill_group, COUNT(*) AS grp_cnt
          FROM JobOrders
         GROUP BY job_id, skill_group) AS J1 CROSS JOIN
       (SELECT R1.job_id, R1.skill_group, S1.candidate_id,
               COUNT(*) AS candidate_cnt
          FROM JobOrders AS R1, CandidateSkills AS S1
         WHERE R1.skill_code = S1.skill_code
GROUP BY R1.job_id, R1.skill_group, S1.candidate_id) AS C1
 WHERE J1.job_id = C1.job_id
   AND J1.skill_group = C1.skill_group
   AND J1.grp_cnt = C1.candidate_cnt
 GROUP BY J1.job_id, C1.candidate_id;
```

FROM句のサブクエリテーブル式は共通表式で置き換えてもよいが、それでパフォーマンスが上がるかどうかは分からない。また、テーブル式C1とJ1をほかの場所で使うことがなければ、それらをビューにするメリットはあまりないだろう。

このクエリは3つのGROUP BYを持っているので、その点でも相関サブクエリに比べて速く動くかどうか、断言できない。だが、集約されたテーブルは元テーブルのインデックスを一切使うことができないため、このアプローチのほうが遅くなる危険はあると思う。

パズル 18

ダイレクトメール

行同士を比較してDELETEする

ダイレクトメールを送りつけるために、顧客の住所を管理しているテーブルがある。このテーブルには、同じ住所（address）に住む顧客を1家族としてまとめるためのfam列がある。fam列がなぜ必要かというと、メールはあくまで1家族につき1通だけにしたいからだ。なお、このfam列には、その住所の世帯主の行に振られた主キー（con_id）の値が格納される。

テーブルのレイアウトは次のとおりである。

```
Consumers

conname    address   con_id    fam
==================================
'Bob'      'A'            1   NULL
'Joe'      'B'            3   NULL
'Mark'     'C'            5   NULL
'Mary'     'A'            2      1
'Vickie'   'B'            4      3
'Wayne'    'D'            6   NULL
```

さて、あなたには、同世帯（addressが同じ）の顧客のうち、fam列がNULLである人の行を削除してほしい。ただし、fam列がNULLであっても、1世帯に1人だけという顧客は残したい。上のサンプルで言えば、BobとJoeは削除するが、MarkとWayneは残す、ということだ。

その1

答え

次に示すDELETE文は処理に高い負荷がかかるが、要求を素直にSQL化できている。

```
DELETE FROM Consumers
 WHERE fam IS NULL      -- 自分のfam列はNULLで
   AND EXISTS           -- かつ、ほかに次のような家族がいる
       (SELECT *
          FROM Consumers AS C1
         WHERE C1.con_id <> Consumers.con_id    -- 自分以外で
           AND C1.address = Consumers.address   -- 同じ住所に住み
           AND C1.fam IS NOT NULL);        -- fam列がNULLでない人
```

その2

答え　だが、もう少し考えれば、削除すべき対象は世帯についてCOUNT(*)が1より大きくなる行だということが分かる。

```
DELETE FROM Consumers
 WHERE fam IS NULL        -- fam列がNULL
   AND (SELECT COUNT(*)
          FROM Consumers AS C1
         WHERE C1.address = Consumers.address) > 1;
```

この解の“トリック”は、COUNT(*)がNULL行もカウントしてくれることである。

その3

答え　フランシスコ・モレノは、答えその1の別解を送ってくれた。

```
DELETE FROM Consumers
 WHERE fam IS NULL       -- fam列がNULL
   AND EXISTS (SELECT *
                 FROM Consumers AS C1
                WHERE C1.fam = Consumers.con_id);
```

パ|ズ|ル|
19

セールスマンの売上ランキング

上位3位を取り出す

この問題には、1995年5月に開催されたデータベースワールドで出会った。ある人が、IBMのパビリオンから私のところへ持ってきたのだ。IBMではホワイトボードを用意し、DB2のエキスパートにこの問題を解かせようとしていたが、その人は途方に暮れてしまっていたのである。

まず、セールスマンの名前と売上を記録する次のテーブルが与えられている。

```
CREATE TABLE SalesData
(district_nbr INTEGER NOT NULL,
 sales_person CHAR(10) NOT NULL,
 sales_id INTEGER NOT NULL,
 sales_amt DECIMAL(5,2) NOT NULL);
```

SalesData：売上データテーブル	district_nbr：地区番号	sales_person：セールスマン
sales_id：セールスID	sales_amt：売上高	

問題とは、「上司の求めに応じて、各地域で上位3位の好成績を上げたセールスマンのレポートを作成すること」である。サンプルデータには次のものを使おう。

```
SalesData

distnct_nbr   sales_person   sales_id   sales_amt
=============================================
          1   'Curly'               5        3.00
          1   'Harpo'              11        4.00
          1   'Larry'               1       50.00
          1   'Larry'               2       50.00
          1   'Larry'               3       50.00
          1   'Moe'                 4        5.00
          2   'Dick'                8        5.00
          2   'Fred'                7        5.00
          2   'Harry'               6        5.00
          2   'Tom'                 7        5.00
          3   'Irving'             10        5.00
          3   'Melvin'              9        7.00
          4   'Jenny'              15       20.00
          4   'Jessie'             16       10.00
          4   'Mary'               12       50.00
          4   'Oprah'              14       30.00
          4   'Sally'              13       40.00
```

その1

答え 困ったことに、私たちに与えられた要求にはいくつかの問題がある。まず、欲しいのは上位3位の売上（誰が達成したかは考慮しない）なのか、それとも上位3位のトップセールスマンなのかがはっきりしないのだ。もし前者なら、地区1（distnct_nbr = 1）について見ればLarryが上位3位まで独占している。しかし、上位3人のセールスマンはLarry、Moe、Harpoだ。

また地区2（distnct_nbr = 2）のように、3人以上の人物が全く同じ売上を上げていた場合はどう判断すればよいのだろう？　地区3（distnct_nbr = 3）のように活動しているセールスマンが3人に満たない場合には、レポートから除外するのか含めるのか？ ——これは業務用システムではなくただのパズルなので、「この上司は売上を誰が達成したかを考慮せずに、上位3位の売上を知りたがっている」と決めてしまおう。すると、次のクエリでOKだ。

```
SELECT S0.district_nbr, S0.sales_person,
       S0.sales_id, S0.sales_amt
  FROM SalesData AS S1, SalesData AS S0
 WHERE S0.district_nbr = S1.district_nbr
   AND S0.sales_amt <= S1.sales_amt
 GROUP BY S0.district_nbr, S0.sales_person, S0.sales_id,
          S0.sales_amt
HAVING COUNT(*) <= 3;
```

ただし、この解法だとHAVING句が、地区2のように売上（sales_amt）の同じセールスマンばかりが4人以上いる地区を除外してしまう[訳注1]。そのため、結果は次のようになる。

訳注1：地区2では、各セールスマンから見て、自分以上に売上が高いという同地区のセールスマンが（自分を含め）3人以下になることがない。つまり、HAVING COUNT(*) <= 3を満たすことがなく、地区2のセールスマンは全員除外される。

```
district_nbr   sales_person   sales_id   sales_amt
==================================================
           1   'Larry'               1       50.00
           1   'Larry'               2       50.00
           1   'Larry'               3       50.00
           3   'Irving'             10        5.00
           3   'Melvin'              9        7.00
           4   'Mary'               12       50.00
           4   'Oprah'              14       30.00
           4   'Sally'              13       40.00
```

地区ごとに売上が上位3位に入るセールスマンを得るためにはどうすればよいのだろう？ そのためには、クエリを次のように直せば大丈夫だ。

```
SELECT S0.district_nbr, S0.sales_person
  FROM SalesData AS S1, SalesData AS S0
 WHERE S0.district_nbr = S1.district_nbr
   AND S0.sales_amt <= S1.sales_amt
 GROUP BY S0.district_nbr, S0.sales_person
HAVING COUNT(DISTINCT S1.sales_person) <= 3
 ORDER BY S0.district_nbr, S0.sales_person
```

すると、次のような結果が得られる。売上が上位3位に入るセールスマンをすべて取得している点に注目してほしい。

```
district_nbr   sales_person
===========================
           1   'Harpo'
           1   'Larry'
           1   'Moe'
           3   'Irving'
           3   'Melvin'
           4   'Mary'
           4   'Oprah'
           4   'Sally'
```

地区1では、売上上位3位が現れる。一方、競争の緩い地区3には2人しか現れない。

その2

答え SQL-99で追加されたOLAP関数を使うと、人生はもっと単純になる。

```
SELECT DISTINCT S1.district_nbr, S1.sales_person
  FROM (SELECT district_nbr, sales_person,
               DENSE_RANK() OVER (PARTITION BY district_nbr
                                  ORDER BY sales_amt DESC)
          FROM SalesData) AS S1 (district_nbr, sales_person,
                                 rank_nbr)
 WHERE S1.rank_nbr <= 3;
```

Teradata、Oracle、DB2、SQL Server 2005がこうしたOLAP関数をサポートしている。ランキングの対象とするものによって、使用するOLAP関数も変わってくる。

RANK関数は、パーティション内の行に連番を付与する。重複値があった場合、同じ値には同じ番号が与えられるが、後に続く番号が"飛び石"になる。

DENSE_RANK関数も同じくパーティション内の行に連番を与えるが、こちらは重複値がある場合にも飛び石は生じない。

ROW_NUMBER関数は、パーティション内の各行に一意な連番を割り振る。重複値は考慮しない。

パーティション内でORDER BY句が指定されなければ、番号は場当たり的に割り振られる。例えば、fooについて2つの重複値を持つ次のパーティションに対しては、次のような結果になる。

```
foo   ROW_NUMBER()   RANK()   DENSE_RANK()
=========================================
'A'              1        1              1
'A'              2        1              1
'A'              3        1              1
'B'              4        4              2
'B'              5        4              2
```

パ|ズ|ル|
20 テスト結果
HAVING句の力（その2）

1995年5月、シャンカー氏がCompuServeのSybaseフォーラムにある問題を投稿してきた。問題は、テストの結果を保存しているTestResultsテーブルに関するものだ。

各テストは何段階かのステップ（test_step）で行われ、TestResultsテーブルにはテストの進捗状況として、test_stepごとにその完了日が記録される。test_stepは必ずしも順番どおりには並んでおらず、しかも1つのテストが複数のtest_stepを持っていることもある。だから、「読解スキル」のテストが5つのtest_stepを持っていたり、「数学スキル」が6つのtest_stepを持っていたりするかもしれない。なお、test_stepの番号は1から始まり、必要な数まで振られているとしよう。

```
CREATE TABLE TestResults
(test_name CHAR(20) NOT NULL,
 test_step INTEGER NOT NULL,
 comp_date DATE,                -- NULLは未完了を意味する
 PRIMARY KEY (test_name, test_step));
```

このテーブルから、「すべてのステップが完了しているテストを高速に見つけ出す」クエリを書いてほしい。

その1

答え

まず考えられるのは、次のような“明白な”解答だろう。

```
SELECT DISTINCT test_name
  FROM TestResults AS T1
 WHERE NOT EXISTS (SELECT *
                     FROM TestResults AS T2
                    WHERE T1.test_name = T2.test_name
                      AND T2.comp_date IS NULL);
```

このクエリの意味は、「完了していないtest_stepが1つも存在しないテストを探せ[訳注1]」である。さて、ほかに解を思いつくだろうか？

訳注1：“明白”とは言うものの、EXISTS述語を使い慣れていない人には、この二重否定は簡単ではないだろう。このクエリは、「すべてのtest_stepが完了している」→「完了していないtest_stepが1つも存在しない」という同値変換を利用している。この変換は、述語論理の量化理論の応用である（SQLのEXISTSは述語論理の存在量化子に相当する）。

その2

答え ロイ・ハーヴェイは全く異なるアプローチで、もっと簡単な名答を見つけた。

```
SELECT test_name
  FROM TestResults
 GROUP BY test_name
HAVING COUNT(*) = COUNT(comp_date);
```

このSQL文がなぜうまく動くかというと、COUNT(*)がcomp_date列のNULLを数える（要するに全行を数える）のに対して、COUNT(comp_date)は集計の前にNULLを除外するからである。

これは、ある集合を別の集合と比較したいときに使えるうまい“トリック”だ。このトリックを活かし、完了していないテストがいくつあるかを調べられるように改良したのが、次のクエリである。

```
SELECT test_name,
       COUNT(*) AS test_steps_needed ,
      (COUNT(*) - COUNT(comp_date)) AS test_steps_missing
  FROM TestResults
 GROUP BY test_name
HAVING COUNT(*) <> COUNT(comp_date);
```

もし、完了していないテストのリストだけが欲しくて、ステップがいくつ残っているのかを知る必要がなければ、次のように短く書くこともできる。

```
SELECT DISTINCT test_name
  FROM TestResults
 WHERE comp_date IS NULL;
```

SQL文を書くときには、各行をどう処理するかに気を取られたりせず、集合全体がどのように振る舞うかを考えることが重要なのだ[訳注2]。

訳注2：もし手続き型言語で書くなら、「テーブル（ファイル）から1行ずつ読み出して完了日がNULLかどうかを判断する」という、一種のコントロールブレイク処理になるだろう。しかしSQLの場合、処理の基本単位は行ではなく「行の集合」であり、集合が全体としてどのような特性を持つのかを調べることが重要である。HAVING句はそのための強力な武器になる。次のパズル21でも、再びHAVING句が活躍する。

パ|ズ|ル|
21

飛行機と飛行士

関係除算の使い方／考え方

飛行機のパイロットを一覧するテーブルと、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。ここから、待機中の飛行機すべてを操縦できるパイロットを全員選択したい。

```
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

```
INSERT INTO PilotSkills
VALUES ('Celko', 'Piper Cub'),
       ('Higgins', 'B-52 Bomber'),
       ('Higgins', 'F-14 Fighter'),
       ('Higgins', 'Piper Cub'),
       ('Jones', 'B-52 Bomber'),
       ('Jones', 'F-14 Bomber'),
       ('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) PRIMARY KEY);
```

Hangar：格納庫テーブル　　plane：飛行機

```
INSERT INTO Hangar
VALUES ('B-1 Bomber'),
       ('B-52 Bomber'),
       ('F-14 Fighter');
```

結果は次のようになるはずだ。

```
PnotSkills DIVIDED BY 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。最初にエドガー・F・コッド[訳注1]が定義した関係除算の定義によれば、必要とされる以上の行を持っていても問題はない。

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[訳注2]。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。

その1

答え

この問題の古典的な解法は、ほとんどの教科書に載っている。よく読まれているクリス・デイトが著した古典的教科書でも、解答のひな型が紹介されている。この問題に当てはめて言うと、「パイロットの名前集合（商）を得るために、PilotSkillsテーブル（被除テーブル）をHangarテーブル（除テーブル）で割る」というものだ。

```
SELECT DISTINCT pilot
  FROM PilotSkills AS PS1
 WHERE NOT EXISTS
         (SELECT *
            FROM Hangar
           WHERE NOT EXISTS
                    (SELECT *
                       FROM PilotSkills AS PS2
```

訳注1：RDBの基礎理論の提唱者。

訳注2：本書では、関係除算で除数に相当するテーブルを「除テーブル」、被除数に相当するテーブルを「被除テーブル」と呼ぶことにする。なお、関係除算については『プログラマのためのSQL 第2版』（ピアソン・エデュケーション刊）の「19.2 関係除算」も参照。また『Joe Celko's Analytics & OLAP』（Morgan Kaufmann刊）でも、バスケット解析への応用として関係除算の多様な方法が解説されている。

```
                    WHERE (PS1.pilot = PS2.pilot)
                      AND (PS2.plane = Hangar.plane)));
```

さて、これまでに学んできた“トリック”を使って、別解を見つけられるだろうか。

その2

答え　パズル20「テスト結果」の答えその2を見返してほしい。実は、そこで紹介したロイ・ハーヴェイのトリックが、ここでも使えるのだ。覚えたトリックをほかの場所で試してみるのは大切なことだ。

各パイロットが格納庫内を巡って、自分が操縦できる飛行機にふせんを貼っていくとしよう。貼り付けたふせんの数と格納庫内の飛行機の数が一致すれば、パイロットは格納庫内にあるすべての飛行機を操縦できるということである。これをクエリで書くとこうなる。

```
SELECT Pilot
  FROM PilotSkills AS PS1, Hangar AS H1
 WHERE PS1.plane = H1.plane
 GROUP BY PS1.pilot
HAVING COUNT(PS1.plane) = (SELECT COUNT(*) FROM Hangar);
```

WHERE句では、パイロットでグループ化して集計する前に、PilotSkillsテーブルに登録されている飛行機のリストを、格納庫に存在する機体だけに絞り込んでいる。もし、このテーブルの機体リストがHangarテーブルの部分集合であることが最初から分かっているなら、WHERE句をなくして、2つのCOUNT(x)の式をCOUNT(DISTINCT x)に置き換えることも可能だ。

また、答えその1のように、入れ子のEXISTS述語を使う方法を広めたのがデイトの教科書だとすれば、このCOUNT(*)を使うやり方は私が広めたと言っても過言ではあるまい。面白いことに、これら2つのアプローチにははっきりとした違いがある。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。このとき、答えその1で示したSELECT文はパイロット全員を返すが、先ほど示したSELECT文は空集合を返す。

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、先ほどのSELECT文と同じ動作をする除算演算子を定義している。となれば、デイトもこちらを正しい答え

と認めてくれるだろう。

その3

答え 関係除算にはもう1種類ある。それが「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);
```

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」である。「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とは言っていない。

『DB2 Magazine』の1996年冬号に、シェリル・ラーセンによる「強力なSQL：基礎を越えて」という記事[訳注3]が掲載された。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。

訳注3：「Powerful SQL: Beyond the Basics」 http://www.db2mag.com/db_area/archives/1996/q4/9601lar.shtml

パズル 22

大家の悩み

複雑な外部結合（その1）

カレン・ギャラガーは、マンションの住人のうち、家賃を支払った人物についてのレポートを作ろうとしていた。彼女が考えたSQL文（Microsoft Accessのものから書き換えた）は、次のようなものだ[訳注1]。

```
SELECT *
  FROM Units AS U1 LEFT OUTER JOIN
         (Tenants AS T1 LEFT OUTER JOIN RentPayments AS RP1
            ON T1.tenant_id = RP1.tenant_id)
         ON U1.unit_nbr = T1.unit_nbr
 WHERE U1.complex_id = 32
   AND U1.unit_nbr = RP1.unit_nbr
   AND T1.vacated_date IS NULL
   AND ((RP1.payment_date >= :my_start_date
         AND RP1.payment_date < :my_end_date)
        OR RP1.payment_date IS NULL)
 ORDER BY U1.unit_nbr, RP1.payment_date;
```

Units：部屋テーブル	complex_id：マンションID	unit_nbr：部屋番号
Tenants：借り主テーブル	tenant_id：借り主ID	vacated_date：退去日
RentPayments：家賃支払いテーブル	payment_date：家賃が支払われた日	

彼女が求めていたのは、RentPaymentsテーブルのうち、payment_dateがある期間内に含まれている行と、部屋と借り主の組み合わせと結合する“空行”（つまり結合しない行）をまとめたレポートである。ところがこのクエリだと、WHERE句に記述されているRentPaymentsテーブルについての条件を削除しなければ、RentPaymentsテーブルの“空行”を得られなかった。さて、どこに間違いがあったのだろう？ それを見つけて正しいクエリに直してほしい。

訳注1：原著には参照しているテーブルの定義が記されていないが、参考までに妥当と思われるUnitsテーブル、Tenantsテーブル、RentPaymentsテーブルの定義を掲げておく。

```
CREATE TABLE Tenants
(tenant_id INTEGER,
 unit_nbr  INTEGER,
 vacated_date DATE,
 PRIMARY KEY (tenant_id,
              unit_nbr));
```

```
CREATE TABLE Units
(complex_id INTEGER,
 unit_nbr   INTEGER,
 PRIMARY KEY (complex_id,
              unit_nbr));
```

```
CREATE TABLE RentPayments
(tenant_id INTEGER,
 unit_nbr  INTEGER,
 payment_date DATE,
 PRIMARY KEY (tenant_id,
              unit_nbr));
```

その1

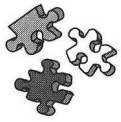
答え ヒントは、変わるものは何か、変わらないものは何かを考えて外部結合を重ねることである。部屋（Units）と借り主（Tenants）の組み合わせから見ると、借り主は時間の経過とともに入れ替わるが、部屋は不変だ。したがって、外部結合において全件表示されるべきは部屋のほうである。部屋と借り主の組み合わせが得られたら、家賃の支払い（Rent Payments）についても同じように考えてほしい。借り主が部屋に入居していても家賃が支払われない可能性がある（家賃の支払いは不変ではない）、という結論に達するだろう。

```
SELECT *    -- 実際に使うときは、必要な列だけ指定すること
  FROM (Units AS U1 LEFT OUTER JOIN Tenants AS T1
         ON U1.unit_nbr = T1.unit_nbr
        AND T1.vacated_date IS NULL
        AND U1.complex_id = 32)
          LEFT OUTER JOIN RentPayments AS RP1
            ON (T1.tenant_id = RP1.tenant_id
           AND U1.unit_nbr = RP1.unit_nbr)
 WHERE RP1.payment_date BETWEEN :my_start_date AND :my_end_date
    OR RP1.payment_date IS NULL;
```

(T1.tenant_id = RP1.tenant_id AND U1.unit_nbr = RP1.unit_nbr) という述語は、「ある1人の借り主がある1つの部屋に家賃を支払っている」という意味だ。このクエリは、1人の借り主が複数の部屋を借りている場合にもうまくいく。また、「部屋を借りていない人からは家賃を徴収しない」というルールも参照整合性制約によって適用できると考えてよい。なお、BETWEEN述語を使うとコードが簡単になり可読性と保守性が高まるのだが、代わりに契約の失効日を調節する必要が生じる[訳注2]。

訳注2：BETWEENは両端の日付を含むため、問題文にあるSQL文のような「:my_end_dateを含まない」という条件にはならない。そのため条件に使用する日付を変える必要がある、ということだろう。

パ|ズ|ル|
23 雑誌と売店
手続き型から宣言型へ考え方を切り換える（その1）



これは、1994年11月にキース・マクレガーがCompuServeのSybaseフォーラムに投稿した問題だ。ある日、エンドユーザの1人が、次のようなクエリを彼のところに持ち込んできた。マクレガーは3日近く試行錯誤したが、解法の手がかりすら得られなかった。COBOLとフラットファイルを使うのであれば30分で解ける問題なのだが、それをSQLで行うにはどうすればよいのか、彼は皆目見当がつかなかったという。

実はこの問題は、手続き型の考え方から宣言的な考え方へ頭を切り替えるのに格好の例題である。次のような、雑誌販売を管理するためのテーブルがあるとしよう[訳注1]。

```
CREATE TABLE Titles
(product_id INTEGER NOT NULL PRIMARY KEY,
 magazine_sku INTEGER NOT NULL,
 issn INTEGER NOT NULL,
 issn_year INTEGER NOT NULL);

product_id   magazine_sku   issn   issn_year
==========================================
         1          12345      1        2006
         2           2667      1        2006
         3          48632      1        2006
         4           1107      1        2006
         5          12345      2        2006
         6           2667      2        2006
         7          48632      2        2006
         8           1107      2        2006
```

Titles：雑誌テーブル　product_id：製品番号　issn_year：ISSNの発行年
magazine_sku：雑誌のSKU番号（SKUは在庫管理の最小単位の意）
issn：ISSN（国際標準逐次刊行物番号。雑誌や新聞などを識別できる）

```
CREATE TABLE Newsstands
 (stand_nbr INTEGER NOT NULL PRIMARY KEY,
  stand_name CHAR(20) NOT NULL);
```

訳注1：TitlesテーブルとSalesテーブルのサンプルデータは、答えその7で解法を紹介されているクズネツォーフ氏によるもの。Newsstandsテーブルのデータは、訳者が追加。

```
stand_nbr   stand_name
=========================
        1   'Newsstands1'
        2   'Newsstands2'
        3   'Newsstands3'
        4   'Newsstands4'
```

Newsstands：売店テーブル　　stand_nbr：売店番号　　stand_name：売店名

```
CREATE TABLE Sales
(product_id   INTEGER NOT NULL REFERENCES Titles(product_id),
 stand_nbr    INTEGER NOT NULL REFERENCES Newsstands(stand_nbr),
 net_sold_qty INTEGER NOT NULL,
 PRIMARY KEY(product_id, stand_nbr));

product_id   stand_nbr   net_sold_qty
=====================================
         1           1              1  -- Newsstands1
         2           1              4          〃
         3           1              1          〃
         4           1              1          〃
         5           1              1          〃
         6           1              2          〃
         7           1              1          〃
         3           2              1  -- Newsstands2
         4           2              5     (条件を満たす)
         8           2              6          〃
         1           3              1  -- Newsstands3
         2           3              3     (条件を満たす)
         3           3              3          〃
         4           3              1          〃
         5           3              1          〃
         6           3              3          〃
         7           3              3          〃
         1           4              1  -- Newsstands4
         2           4              1          〃
         3           4              4          〃
         4           4              1          〃
         5           4              1          〃
         6           4              1          〃
         7           4              2          〃
```

Sales：売上テーブル　　product_id：製品番号　　stand_nbr：売店番号
net_sold_qty：売上部数

彼が選択（SELECT）する必要があったのは、次の条件をどちらか一方でも満たしているすべての売店である。

1. SKUが2667番と48632番の2つの雑誌について、net_sold_qty（売上部数）の平均がともに2より大きい（一方でも2以下であれば、その売店は選択しない）。
2. SKUが1107番の雑誌について、net_sold_qtyの平均が5より大きい（この条件を満たすなら、1.の条件を満たすか否かにかかわらず選択する）。

その1

答え

基本的な情報を得るために、まず3つのテーブルを結合したビューを作ろう。このビューは、後でほかのレポートを作るときにも役立つはずだ。

```
CREATE VIEW MagazineSales(stand_name, magazine_sku, net_sold_qty)
AS SELECT Newsstands.stand_name, Titles.magazine_sku,
          net_sold_qty
     FROM Titles, Sales, Newsstands
    WHERE Sales.stand_nbr   = Newsstands.stand_nbr
      AND Titles.product_id = Sales.product_id;
```

最初に試したのは、次のような見るに堪えないクエリだった。

```
SELECT stand_name
  FROM MagazineSales AS M0
 GROUP BY stand_name
HAVING -- 条件を満たす2つ
     ((SELECT AVG(net_sold_qty)
         FROM MagazineSales AS M1
        WHERE M1.stand_name = M0.stand_name
          AND magazine_sku = 1107) > 5)
  OR ((SELECT AVG(net_sold_qty)
         FROM MagazineSales AS M2
        WHERE M2.stand_name = M0.stand_name
          AND magazine_sku IN (2667, 48632)) > 2)
 AND NOT -- 条件を満たさない2つ
     ((SELECT AVG(net_sold_qty)
         FROM MagazineSales AS M3
        WHERE M3.stand_name = M0.stand_name
          AND magazine_sku = 2667) < 2
```

```
        OR
        (SELECT AVG(net_sold_qty)
           FROM MagazineSales AS M4
          WHERE M4.stand_name = M0.stand_name
            AND magazine_sku = 48632) < 2);
```

さて、このクエリを簡略化または改良できるだろうか？

ド・モルガンの法則が有効かもしれない。ディシジョンテーブルを使うのもよいだろう。

その2

1995年4月、米国イリノイ州クラレンドンヒルズで個人コンサルタントとして活躍中のカール・C・フェダールから、次のような解答が提案された。この解答では平均販売数のビューを作り、それに対して「両方の条件のしきい値を超えていること」という条件のEXISTS述語を使うことで、問題をとても簡単なものにしている。

平均販売数のビューは次のものだ。

```
CREATE VIEW MagazineSales
(stand_nbr, magazine_sku, avg_qty_sold)
 AS SELECT Sales.stand_nbr, Titles.magazine_sku,
           AVG(Sales.net_sold_qty)
      FROM Titles, Newsstands, Sales
     WHERE Titles.product_id = Sales.product_id
       AND Newsstands.stand_nbr = Sales.stand_nbr
       AND Titles.magazine_sku IN (1107, 2667, 48632)
     GROUP BY Sales.stand_nbr, Titles.magazine_sku;
```

これを使うと、クエリはぐっと短くなる。

```
SELECT DISTINCT N0.stand_name
  FROM MagazineSales AS M0, Newsstands AS N0
 WHERE N0.stand_nbr = M0.stand_nbr
   AND ((M0.magazine_sku = 1107 AND M0.avg_qty_sold > 5)
        OR (M0.magazine_sku = 2667 AND M0.avg_qty_sold > 2
            AND EXISTS (SELECT *
                          FROM MagazineSales AS Other
```

```
                                WHERE Other.magazine_sku = 48632
                                  AND Other.stand_nbr = M0.stand_nbr
                                  AND Other.avg_qty_sold > 2)));
```

古いバージョンのSybaseなどでは、結合後に集約したビューからは正しい結果が得られなかった。今日では、ビューは共通表式にすることもできる。

なお、ここではビューを使ったが、代わりに一時テーブルを使ってもよい。

その3

答え　ここで紹介する解答は、ドンカー・システム社でテクニカルサポートマネージャーを務めるアダム・トンプソンから寄せられた。彼は本書の初版を読んで、この解答を思いついたという。

まず、彼は雑誌タイトルに索引を付与した。

```
CREATE INDEX Titles_magazine_sku
ON Titles (magazine_sku, product_id);
```

これにより、特にデータ量が多い場合にはパフォーマンスが大きく改善されるだろう。ただ、インデックスは標準SQLの一部ではないので、本書で言及しない。

それから、彼はSQL-89に完全準拠する解答を得た。

```
SELECT DISTINCT N1.stand_name
  FROM Newsstands AS N1
 WHERE N1.stand_nbr IN (SELECT S1.stand_nbr
                          FROM Sales AS S1
                         WHERE S1.product_id IN
                           (SELECT T1.product_id
                              FROM Titles AS T1
                             WHERE magazine_sku = 1107)
                         GROUP BY S1.stand_nbr
                        HAVING AVG(S1.net_sold_qty) > 5)
    OR (N1.stand_nbr IN (SELECT S1.stand_nbr
                           FROM Sales AS S1
                          WHERE S1.product_id IN
                                (SELECT T1.product_id
                                   FROM Titles AS T1
                                  WHERE magazine_sku = 2667)
                          GROUP BY S1.stand_nbr
```

```
                    HAVING AVG(S1.net_sold_qty) > 2)
      AND N1.stand_nbr IN (SELECT S1.stand_nbr
                             FROM Sales AS S1
                            WHERE S1.product_id IN
                              (SELECT T1.product_id
                                 FROM Titles AS T1
                                WHERE magazine_sku = 48632)
                            GROUP BY S1.stand_nbr
                           HAVING AVG(S1.net_sold_qty) > 2));
```

彼はこのSELECT文をSQL Anywhere v.5.5でテストした。その際、先に示したインデックスのほかにもう1つインデックスを張っていたため、N1への全表検索（フルスキャン）は1回で済んでいたという。このクエリに対して1回の全表検索で済むというのは、かなり良好な動作だと思う。

また、相関サブクエリのGROUP BY句ではstand_nbr列だけが必要で、product_id列は省略できることに注意してほしい。これは、直前のWHERE句でグループ化の対象を1107番、2667番、48632番の雑誌に絞り込んでいるからだ。ただ、より一般的な解答としては、クエリ構文の仕様が今後拡張されたときのために「GROUP BY stand_nbr, product_id」という記述を認めたほうがよいだろう。SQL-92では、述語の中で複数の列を参照する式がすでに使えるのだが、それを実装したRDBMSはまだあまりない。

その4

答え 次に紹介する解答はGROUP BY句を使わない代わりに、CASE式を必要とする。

```
SELECT N1.stand_name
  FROM Sales AS S1, Titles AS T1, Newsstands AS N1
 WHERE T1.magazine_sku IN (2667, 48632, 1107)
   AND S1.product_id = T1.product_id
   AND S1.stand_nbr = N1.stand_nbr
 GROUP BY N1.stand_name
HAVING (AVG(CASE WHEN T1.magazine_sku = 2667
                 THEN S1.net_sold_qty ELSE NULL END) > 2
        AND AVG(CASE WHEN T1.magazine_sku  = 48632
                     THEN S1.net_sold_qty ELSE NULL END) > 2)
    OR AVG(CASE WHEN T1.magazine_sku  = 1107
                THEN S1.net_sold_qty ELSE NULL END) > 5;
```

その5

答え リチャード・レムレーは、1997年9月にSQL-92構文に沿った解答をいくつか提案してきた。

```
SELECT stand_name
  FROM (SELECT N1.stand_name
          FROM Sales AS S1
               INNER JOIN Titles AS T1
               ON S1.product_id = T1.product_id
                  INNER JOIN Newsstands AS N1
                  ON S1.stand_nbr = N1.stand_nbr
         WHERE T1.magazine_sku IN (2667, 48632, 1107)
         GROUP BY N1.stand_nbr, N1.stand_name
        HAVING (AVG(CASE WHEN T1.magazine_sku = 2667
                         THEN S1.net_sold_qty ELSE NULL END) > 2
           AND AVG(CASE WHEN T1.magazine_sku = 48632
                        THEN S1.net_sold_qty ELSE NULL END) > 2)
            OR AVG(CASE WHEN T1.magazine_sku = 1107
                        THEN S1.net_sold_qty ELSE NULL END) > 5);
```

しかし、彼はこの解答を書いていて、次のような点に気づいたという。

1. 売店（Newsstands）テーブルはこの問題と何の関係もない。これは、条件を満たすstand_nbrが決定した後で売店の名前を調べるためだけに使われている。これを前面に持ってくるのは紛らわしく、解答を複雑にするだけだ。

2. 売店名（stand_name）の検索は、条件を満たすstand_nbrにつき、1回しか実行する必要がない。それならば、売店名の検索はstand_nbrが決まった後に、最後に行うべきである。もし、Salesテーブルに1万行あったら、この解答では売店テーブルに対して1万回の結合をすることになる。最初の解答では1回のINNER JOINだけだった。あなたはどちらがより適切だと思うか？

3. この解答では、GROUP BY句に売店名の列を含める必要がある。SELECT句の中に売店名を入れるためなのだが、これは以下のような理由から推奨できない。
   - stand_name列は集約とは無関係であり、論理的には不要である。
   - SELECT句やHAVING句、ORDER BY句に列を追加したいと思ったら、

GROUP BY句にもその列を追加する必要があるが、これは非論理的だ。というのも、GROUP BY句には問題を解決するために集約が必要な列だけを含めるべきであって、それ以外の不要な列を入れるべきではないからだ。GROUP BY句に不要な列があると、その句の目的が不明確になり、クエリを読みにくくしてしまう。

- GROUP BY句に不要な列を追加することは、大きなパフォーマンス低下の危険を潜在的に抱え込むことになる。stand_nbrだけなら、1つの整数を見るだけで済む。だが、SELECT句に1ダースの列を追加すれば、同じ数の列をGROUP BY句にも追加せざるを得ない。その結果、1ダースの列を使って1万行の集約を行うよう、サーバに強制することになるだろう。おそらくは数百バイトにのぼるであろう複数列の組み合わせによる集約と、1列の整数列のみの集約——パフォーマンスはどの程度違うだろうか？
- メンテナンスが非常に面倒になる。本来グルーピングに必要ないはずの列を、SELECTに追加したいという理由でGROUP BY句にも追加せねばならないのだから。

その6

答え 1999年7月、フランシスコ・モレノはSQL-92の集合演算子と、ちょっとした代数をうまく使った解答を提案してくれた[訳注2]。

```
SELECT stand_name
  FROM Newsstands AS N1
 WHERE 1 =
   ANY ((SELECT SIGN(AVG(net_sold_qty) - 2)
           FROM Sales AS S1
          WHERE product_id IN (SELECT product_id
                                 FROM Titles
                                WHERE magazine_sku = 2667)
                AND S1.stand_nbr = N1.stand_nbr
        INTERSECT
        SELECT SIGN(AVG(net_sold_qty) - 2)
          FROM Sales AS S2
```

訳注2：この解答では、まずINTERSECTによって、2667番の雑誌の平均売上数が2より大きい売店と、48632の雑誌の平均売り上げ数が2より大きい売店の共通集合を取る。その後にUNIONで、1107番の雑誌の平均売り上げ数が5より大きい売店を加えている。

```
 WHERE product_id IN (SELECT product_id
                        FROM Titles
                       WHERE magazine_sku = 48632)
   AND S2.stand_nbr = N1.stand_nbr)
UNION
SELECT SIGN(AVG(net_sold_qty) - 5)
  FROM Sales AS S3
 WHERE product_id IN (SELECT product_id
                        FROM Titles
                       WHERE magazine_sku = 1107)
   AND S3.stand_nbr = N1.stand_nbr);
```

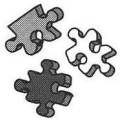
その7

答え クズネツォーフ氏は、こんなシンプルな解答を考え出した。

```
SELECT stand_nbr
  FROM (SELECT stand_nbr,
               AVG(CASE WHEN magazine_sku = 2667
                        THEN net_sold_qty END),
               AVG(CASE WHEN magazine_sku = 48632
                        THEN net_sold_qty END),
               AVG(CASE WHEN magazine_sku = 1107
                        THEN net_sold_qty END)
          FROM Sales, Titles
         WHERE Sales.product_id = Titles.product_id
         GROUP BY stand_nbr
        ) AS T (stand_nbr, avg_2667, avg_48632, avg_1107)
 WHERE avg_1107 > 5 OR (avg_2667 > 2 AND avg_48632 > 2);
```

ひとこと言っておくと、CASE式でELSE NULLを省略することは文法的に間違いではないが、私は将来の変更や追加に備えて省略せずに書くほうが好きだ。そうしておけば、後で見たときにここでNULLが生成されることを見落とさずに済む。

パ|ズ|ル|
24 10個のうち1つだけ
擬似配列の扱い方



レガシーシステムをSQLデータベースへ移行している最中、アラン・フランクマンはある問題に突き当たった。次のようなテーブルがあるとする。

```
CREATE TABLE MyTable
(keycol INTEGER NOT NULL,
 f1 INTEGER NOT NULL,
 f2 INTEGER NOT NULL,
 f3 INTEGER NOT NULL,
 f4 INTEGER NOT NULL,
 f5 INTEGER NOT NULL,
 f6 INTEGER NOT NULL,
 f7 INTEGER NOT NULL,
 f8 INTEGER NOT NULL,
 f9 INTEGER NOT NULL,
 f10 INTEGER NOT NULL);
```

f1～f10の列は、1つの配列をテーブルに展開するために作ったものだ。彼がしたかったのは、f1～f10列のうち1つだけゼロでない列がある行を選択することである。

あなたには、これを実現する方法をできる限り多く考えてほしい。パフォーマンスは気にしなくてよいから、とにかくたくさん。

その1

答え

SybaseなどのRDBMSでは、SIGN関数を使うことができる[訳注1]。この関数は、引数が負なら−1を、0なら0を、正なら1を返す。もし、f1列～f10列には0以上の数値が格納されるのであれば、次のように簡単に書ける。

```
SELECT *
  FROM MyTable
 WHERE SIGN(f1) + SIGN(f2) + ... + SIGN(f10) = 1;
```

もし、負数も含まれるなら、SIGN(ABS(fn))とすればよい。

訳注1：SIGN関数は標準SQLには入っていないが、ほとんどのRDBMSで使える。

また、SIGN関数とABS関数の組み合わせと同じ処理を、SQL-92のCASE式を用いて次のように書くこともできる。

```
CASE WHEN x <> 0 THEN 1 ELSE 0 END
```

その2

答え　配列をまさに配列の形で擬似的に表現するというのなら、このテーブルを第1正規形（1NF）にするべきだ[訳注2]。

```
CREATE TABLE Foobar
(keycol INTEGER NOT NULL,
 i INTEGER NOT NULL CHECK (i BETWEEN 1 AND 10),
 f INTEGER NOT NULL,
 PRIMARY KEY (keycol, i));
```

追加のiという列は、配列の添え字に相当する。そうすると問題は、「ゼロでない非キー列を1つだけ持つエンティティを探す」ことから、「ゼロであるf列をちょうど9個持つエンティティを探す」ことに変換される。これで問題がたちどころに簡単になった。

```
SELECT keycol
  FROM Foobar
 WHERE f = 0
 GROUP BY keycol
HAVING COUNT(*) = 9;
```

Foobarテーブルの代わりに、同じ構造のビューを作ってもかまわないが、優秀なオプティマイザでないと処理が非常に遅くなるだろう。

```
CREATE VIEW Foobar (keycol, f)
AS SELECT keycol, f1
     FROM MyTable
    WHERE f1 <> 0
```

訳注2：著者はいわゆる「行持ち」の形式を第1正規形と呼んでいるが、この用語法は正確ではない。問題として提示されたテーブルは繰り返し項目を持つが、すでに第1正規形を満たしてはいる。

```
    UNION
    SELECT keycol, f2 FROM MyTable WHERE f2 <> 0
    UNION
     ...
    UNION
    SELECT keycol, f10 FROM MyTable WHERE f10 <> 0 ;
```

その3

答え　3つ目の方法は、まだ一般には利用できないSQL-92の機能に依存している。まずはコードを見てもらおう。

```
SELECT *
  FROM MyTable
 WHERE (f1, f2, ... , f10) IN
          (VALUES (f1, 0, 0, 0, 0, 0, 0, 0, 0, 0),
                  (0, f2, 0, 0, 0, 0, 0, 0, 0, 0),
                  ...
                  (0, 0, 0, 0, 0, 0, 0, 0, 0, f10))
   AND (f1 + f2 + ... f10) <> 0;
```

SQL-92では、述語との比較の中で行構築子またはテーブル構築子を使って、処理に使うだけの一時的な行やテーブルを作成できる。IN述語は、同一性述語（「=」のこと）をORでつなげた場合と同じ働きをする。行方向の比較は1つずつ行われ、対応するすべての値が等しいときにだけ真になる。

その4

答え　「ゼロでない列が1つだけある」ということは、「残り9つの列がすべてゼロ」ということと同値だ。

```
SELECT *
  FROM MyTable
 WHERE 0 IN (VALUES (f2 + f3 + ... f10), -- f1を抜く
                    (f1 + f3 + ... f10), -- f2を抜く
                    ...
                    (f1 + f2 + ... f9))  -- f10を抜く
   AND (f1 + f2 + ... f10) <> 0;
```

その5

答え　1999年1月にトレヴァー・ドワイヤーは、彼が実際に遭遇した類似の問題をCompuServeに投稿してきた。違うのは、彼のテーブルはゼロの代わりにNULLを含んでおり、少なくとも1つはNULLでない列を含む行を見つけることが課題だったことだ。これはSQL-92では非常に簡単に書ける。

```
SELECT *
  FROM MyTable
 WHERE COALESCE(f1, f2, f3, f4, f5, f6, f7, f8, f9, f10)
       IS NOT NULL;
```

COALESCE関数はリストの中から最初のNULLでない値を返す。もしリスト全体がNULLだけしか含んでいなければ、NULLを返す。

冒頭の問題が、リストの中の式に変換関数を使うことでこの問題に置き換わることは明らかだ。なお、NULLIF関数は、列の値が0ならNULLを返す。

```
COALESCE (NULLIF (f1, 0), NULLIF (f2, 0), ..., NULLIF (f10, 0))
```

パ|ズ|ル|
25 マイルストーン
行と列を入れ替える

もともとは少し違う形だったが、このパズルはブライアン・ヤングから持ち込まれた。

彼は、顧客の注文（my_order）に対しどんな各サービス（service_type）をいつ提供するのか（マイルストーン）を連綿と記録するためのシステムを担当していた。いわばサービス提供のスケジュール管理システムであり、記録される提供日はサービスのタイプによって変わってくる。だから、経営者が各店舗のスケジュールを横断的に見たいと要求してきたのはもっともなことだと、私も認めねばならない。だが、そういうことは本来フロントエンドツールの表示用関数で実現するべきであって、データベース上で処理するべきではない。それなのに経営者たちは、スケジュールを表示するときに業務コード（service_type）を指定できたらいいな、などとも思っていた。

ブライアンは、スティーブ・ロチが書いたSQL Serverの解説本の中に、この問題を解くためのうまい方法を見つけた。ただし、そのテクニックを使って正しく結果を得るためにはSUM関数と整数の乗算を使う必要があり、日付に対してはうまく適用できなかった（名誉のために言っておくと、ロチのやり方は本当に見事なものだった！）。

このシステムのテーブルは、次のような構造をしている。

```
CREATE TABLE ServicesSchedule
(shop_id CHAR(3) NOT NULL,
 order_nbr CHAR(10) NOT NULL,
 sch_seq INTEGER NOT NULL CHECK (sch_seq IN (1,2,3)),
 service_type CHAR(2) NOT NULL,
 sch_date DATE,
 PRIMARY KEY (shop_id, order_nbr, sch_seq));
```

ServicesSchedule：サービススケジュールテーブル	shop_id：売店ID
order_nbr：注文番号	sch_seq：サービスの状況
service_type：サービスタイプ／業務コード	sch_date：サービス提供日

sch_seqは次のようにコード化されている。

(1 = 'processed') ←サービスを提供中
(2 = 'completed') ←サービスを完了
(3 = 'confirmed') ←サービスの確認済み

したがって、データは通常、次のようになる。

```
ServicesSchedule

shop_id   order_nbr     sch_seq   service_type   sch_date
=============================================================
'002'     '4155526710'       1   '01'           '1994-07-16'
'002'     '4155526710'       2   '01'           '1994-07-30'
'002'     '4155526710'       3   '01'           '1994-10-01'
'002'     '4155526711'       1   '01'           '1994-07-16'
'002'     '4155526711'       2   '01'           '1994-07-30'
'002'     '4155526711'       3   '01'           NULL
```

service_typeが'01'の場合には、ブライアンたちは次のようなフォーマットでデータを表示させたかったはずだ。

```
order_nbr       processed       completed       confirmed
=========================================================
'4155526710'    '1994-07-16'    '1994-07-30'    '1994-10-01'
'4155526711'    '1994-07-16'    '1994-07-30'    NULL
```

その1

答え

もし、使っているRDBMSがSQL-92対応ではなくSQL-89対応であっても、自己結合を使えば希望の形でデータを取り出せる。

```
SELECT DISTINCT S1.order_nbr, S1.sch_date,  S2.sch_date,
                S3.sch_date
  FROM ServicesSchedule S1, ServicesSchedule S2,
       ServicesSchedule S3
 WHERE S1.service_type = :my_tos   -- 業務コードをセットする
   AND S1.order_nbr = :my_order    -- 注文番号をセットする
   AND S1.sch_seq = 1
   AND S2.order_nbr = S1.order_nbr
   AND S2.sch_seq = 2
   AND S3.order_nbr = S1.order_nbr
   AND S3.sch_seq = 3;
```

ただし、RDBMS製品によっては自己結合のコストが非常に高くなることがある。古い実装ではこれが最速の方法だとは思うが、ほかの方法を思いつくだろうか？

その2

答え　SQL-92なら、サブクエリ式を使えばあっという間に解ける。

```
SELECT DISTINCT S0.order_nbr,
       (SELECT sch_date
          FROM ServicesSchedule AS S1
         WHERE S1.sch_seq = 1
           AND S1.order_nbr = S0.order_nbr) AS processed,
       (SELECT sch_date
          FROM ServicesSchedule AS S2
         WHERE S2.sch_seq = 2
           AND S2.order_nbr = S0.order_nbr) AS completed,
       (SELECT sch_date
          FROM ServicesSchedule AS S3
         WHERE S3.sch_seq =3
           AND S3.order_nbr = S0.order_nbr) AS confirmed
  FROM ServicesSchedule AS S0
 WHERE service_type = :my_tos ; -- 業務コードをセットする
```

この"トリック"の難点は、オプティマイザによってはうまく最適化されないケースがあることだ。もしかしたら、答えその1の自己結合よりパフォーマンスが悪いかもしれない。

その3

答え　あるいは、UNION ALL演算子を使って、元テーブルのデータを横（1行）に並べた作業用テーブルを作ろうと試みた人もいるだろう。通常はあまりパフォーマンスの出ないやり方だが、元テーブルが非常に巨大な場合には、答えその2の自己結合より速いこともあり得る。

```
INSERT INTO Work (order_nbr, processed, completed, confirmed)
SELECT order_nbr, sch_date, NULL, NULL
  FROM ServicesSchedule AS S1
 WHERE S1.sch_seq = 1
   AND S1.order_nbr = :my_order
   AND service_type = :my_tos    -- 業務コードをセット
UNION ALL
SELECT order_nbr, NULL, sch_date, NULL
```

```
  FROM ServicesSchedule AS S2
 WHERE S2.sch_seq = 2
   AND S2.order_nbr = :my_order
   AND service_type = :my_tos    -- 業務コードをセット
UNION ALL
SELECT order_nbr, NULL, NULL, sch_date
  FROM ServicesSchedule AS S3
 WHERE S3.sch_seq = 3
   AND S3.order_nbr = :my_order
   AND service_type = :my_tos;   -- 業務コードをセット
```

場合によっては、この単純なUNION ALLの文を3つのINSERT文に分解する必要があるかもしれない。いずれにせよ、作業用テーブルができあがれば、最終的に欲しい結果を出力するクエリは簡単だ。

```
SELECT order_nbr, MAX(processed), MAX(completed), MAX(confirmed)
  FROM Work
 GROUP BY order_nbr;
```

MAX関数はグループ内のNULLでない最大値を選択する。グループ内の非NULL値は最初から1つだけ、という場合にはMAX関数は不要になる。

その4

答え

UNIONの繰り返しは、SQL-92ならCASE式で書き換えられることが多い。その方向で考えて作ったのが次の解だ。

```
SELECT order_nbr,
       (CASE WHEN sch_seq = 1
             THEN sch_date
             ELSE NULL END) AS processed,
       (CASE WHEN sch_seq = 2
             THEN sch_date
             ELSE NULL END) AS completed,
       (CASE WHEN sch_seq = 3
             THEN sch_date
             ELSE NULL END) AS confirmed
  FROM ServicesSchedule
 WHERE service_type = :my_tos
   AND order_nbr = :my_order;
```

ただし、これだと同じ注文番号のデータが同じ行に並ばない。この点を改善するには、次のようにGROUP BY句を加えればよい。

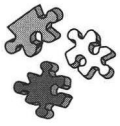
```
SELECT order_nbr,
       MAX(CASE WHEN sch_seq = 1
                THEN sch_date
                ELSE NULL END) AS processed,
       MAX(CASE WHEN sch_seq = 2
                THEN sch_date
                ELSE NULL END) AS completed,
       MAX(CASE WHEN sch_seq = 3
                THEN sch_date
                ELSE NULL END) AS confirmed
  FROM ServicesSchedule
 WHERE service_type = :my_tos
   AND order_nbr = :my_order
 GROUP BY order_nbr, service_type;
```

現在のSQLではこれがベストだろう。古いSQLで書かれたコードも、この形に変換することが可能だ。

パズル 26

DFD

存在しない組み合わせを見つける



トム・ブラッグは、CompuServeのCASEフォーラムにこういう問題を送ってきた（ここでは少し改変したものを示す）。

ここに、データフローダイアグラム（DFD）[訳注1]に関する情報を管理するテーブルがあるとしよう。列構成はダイアグラム名、そのダイアグラムに含まれるバブル（円）の名前、そしてバブル間を走るフロー（矢印）の名前だ。

```
CREATE TABLE DataFlowDiagrams
(diagram_name CHAR(10) NOT NULL,
 bubble_name CHAR(10) NOT NULL,
 flow_name CHAR(10) NOT NULL,
 PRIMARY KEY (diagram_name, bubble_name, flow_name));
```

DataFlowDiagrams：DFDテーブル	diagram_name：ダイアグラム名
bubble_name：バブル名	flow_name：フロー名

問題を説明するために、次のサンプルを使うとしよう。

```
DataFlowDiagrams

diagram_name   bubble_name   flow_name
=======================================
'Proc1'        'input'       'guesses'
'Proc1'        'input'       'opinions'
'Proc1'        'crunch'      'facts'
'Proc1'        'crunch'      'guesses'
'Proc1'        'crunch'      'opinions'
'Proc1'        'output'      'facts'
'Proc1'        'output'      'guesses'
'Proc2'        'reckon'      'guesses'
'Proc2'        'reckon'      'opinions'
 ...
```

ここから知りたいことは、「各ダイアグラムにあるバブルがどのフローと結び付いて**いないか**」である。これは失われたデータフローの有無など、ダイアグラムの妥当性を

訳注1：情報システムにおけるデータの流れを表現するための記法。データの処理を表す円の間を、データの流れを表す矢印でつなぐ。

チェックするルーチン作業の一部である。また、問題を簡単にするために、どのバブルもダイアグラム内にあるすべてのフローと結び付かなければならないとしよう。つまり、サンプルでは、(Proc1, input) というバブルには 'facts' フローが欠けており、(Proc1, output) というバブルには 'opinions' フローが欠けている、という具合だ。

その1

答え SQL-92なら、次のようなクエリを書くことができる。

```
SELECT diagram_name, bubble_name, flow_name
  FROM (SELECT F1.diagram_name, F1.bubble_name, F2.flow_name
          FROM DataFlowDiagrams AS F1
          CROSS JOIN
          DataFlowDiagrams AS F2) AS TMP
EXCEPT
SELECT F3.diagram_name, F3.bubble_name, F3.flow_name
  FROM DataFlowDiagrams AS F3;
```

要するに、可能なすべてのバブルとフローの組み合わせを作り、そこから実際に存在する組み合わせをEXCEPT演算子で“引き算”しているわけだ。

その2

答え SQL-92の別解には、次のようなクエリも考えられる。差集合演算を行うのに、EXCEPTの代わりにNOT INを使っている。

```
SELECT DISTINCT diagram_name, bubble_name, flow_name
  FROM (SELECT F1.diagram_name, F1.bubble_name, F2.flow_name
          FROM DataFlowDiagrams AS F1
          CROSS JOIN
          DataFlowDiagrams AS F2
         WHERE F2.flow_name NOT IN
              (SELECT F3.flow_name
                 FROM DataFlowDiagrams AS F3
                WHERE F3.diagram_name = F1.diagram_name
                  AND F3.bubble_name  = F1.bubble_name)) AS TMP
 ORDER BY diagram_name, bubble_name, flow_name;
```

その3

答え SQL-89で解こうとするなら、ビューを使わなければならないだろう。

```
-- すべてのフローを保持するビュー
CREATE VIEW AllDFDFlows (flow_name)
AS SELECT DISTINCT flow_name FROM DataFlowDiagrams;

flow_name
==========
'facts'
'guesses'
'opinions'
```

```
-- 元のテーブルの各行にすべてのフローを割り当てる
CREATE VIEW NewDFD
(diagram_name, bubble_name, flow_name, missingflow)
AS SELECT DISTINCT F1.diagram_name, F1.bubble_name, F1.flow_name,
                   F2.flow_name
     FROM DataFlowDiagrams AS F1, AllDFDFlows AS F2
    WHERE F1.flow_name <> F2.flow_name;
```

```
-- flow_name列に失われたフローが存在しない
-- (diagram_name, bubble_name) の組み合わせを表示する
SELECT DISTINCT diagram_name, bubble_name, missingflow
  FROM NewDFD AS ND1
 WHERE NOT EXISTS (SELECT *
                     FROM NewDFD AS ND2
                    WHERE ND1.diagram_name = ND2.diagram_name
                      AND ND1.bubble_name  = ND2.bubble_name
                      AND ND2.flow_name    = ND1.missingflow)
 ORDER BY diagram_name, bubble_name, missingflow;
```

DISTINCTを付けたのはやりすぎのような気もするが、実行速度を確認してほしい。ネットワークに負荷をかけてすべての行を持ってくるより、こちらのほうが速いはずだ。

パ|ズ|ル|
27 等しい集合を見つける
集合の相等性チェック

集合論は部分集合を表す2つの記号を持っている。1つが「馬の蹄」を横向きにしたもの（⊂）で、集合Aが集合Bに含まれることを意味する。真部分集合と呼ばれることもある。もう1つは、その下に水平線を書き加えたもの（⊆）で、これは「含まれるか、または等しい」を意味する。こちらは、単に部分集合とか包含演算子と呼ばれる。

標準SQLは、テーブル同士を比較する演算子をまったく持っていない。大学のリレーショナルデータベースの教科書には、標準SQLに存在しないCONTAINS述語なるものを載せているものもある。例えば、ビピン・C・デサイー著『Introduction to Database Systems』（West Group刊, 1990, ISBN0-314-66771-7）や、エルマスリとナバスの共著『Fundamentals of Database Systems』（Benjamin Cummings刊, 1989, ISBN0-8053-0145-3）は、そういう“チョンボ”をやっている一例だ。この述語は、かつてIBMの実験的なSQLシステム「System R」に存在していたが、実行コストが高くつくという理由で、後の実装からは除かれたのだ。

IN述語は要素として含むかどうかのテストをするだけで、部分集合を含むかどうかを調べることはできない。高校時代に習った集合論を思い出してほしい。要素関係を表す記号はギリシャ文字のイプシロンをかたどったような「∈」というものだった。要素関係は1つの要素に対して適用されるが、部分集合はそれ自身が集合であって要素ではないので適用されないのだ。

ところで、『Database Programming & Design』の1993年11月号に掲載されたクリス・デイトのパズル（「整合性の問題 Part Ⅱ」）は、部品とその供給業者を管理するテーブルを使って、数も種類も**全く**同じ部品を取り扱う供給業者を見つけるものだった。要するに、2つの等しい集合を見つけるという問題である。

ここでは、その問題に挑戦してほしい。サンプルデータには、デイトが作った有名なテーブルを使おう[訳注1]。

```
CREATE TABLE SupParts
(sno CHAR(2) NOT NULL,
 pno CHAR(2) NOT NULL,
 PRIMARY KEY (sno, pno));
```

訳注1：このテーブルの詳細は、デイトの著書『データベースシステム概論 第6版』（丸善刊）や『データベース実践講義』（オライリー・ジャパン刊）を参照。snoが供給業者の識別子（supplier no）、pnoが部品の識別子（part no）を表す。

SupParts：供給業者／部品テーブル	sno：供給業者番号	pno：部品番号

あなたは、この問題を解く方法をどれだけ思いつくだろうか？

その1

答え

1つの方法は、完全外部結合で供給業者の組み合わせを作ることである。この組み合わせ結果には、両供給業者で共通しない部品を含む行まで現れるが、内部結合では現れない行の供給業者列のどちらかはNULLとなる。このことから、どの組み合わせがマッチしなかったのかが分かる。最後に、同じ部品を供給する業者のすべての組み合わせから、これらのマッチしなかった組み合わせを引き算すれば、求めるクエリは完成だ[訳注2]。

```
SELECT SP1.sno, SP2.sno
  FROM SupParts AS SP1 INNER JOIN SupParts AS SP2
    ON SP1.pno = SP2.pno
   AND SP1.sno < SP2.sno
EXCEPT
SELECT DISTINCT SP1.sno, SP2.sno
  FROM SupParts AS SP1 FULL OUTER JOIN SupParts AS SP2
    ON SP1.pno = SP2.pno
   AND SP1.sno < SP2.sno
 WHERE SP1.sno IS NULL
    OR SP2.sno IS NULL;
```

EXCEPT演算子は、集合論の差集合演算に該当する。ただ、この方法の実行速度は非常に遅いだろう。

その2

答え

2つの集合が互いに等しいことを証明する場合、「AがBを含み、かつ、BがAを含む」ことを示すのが普通である。標準SQLで「AがBを含む（AはBの部分集合である）」ことを示す場合には、一般に「Bに存在しない要素がAにも存在しない」ことを示す。そこから、次のような解答の叩き台となるSQL文が書ける。

訳注2：実際には、このクエリは答えその2と同様に、1つでも共通の部品を含む業者のペアを選択してしまう。

```
SELECT DISTINCT SP1.sno, SP2.sno
  FROM SupParts AS SP1, SupParts AS SP2
 WHERE SP1.sno < SP2.sno
   AND SP1.pno IN (SELECT SP22.pno
                     FROM SupParts AS SP22
                    WHERE SP22.sno = SP2.sno)
   AND SP2.pno IN (SELECT SP11.pno
                     FROM SupParts AS SP11
                    WHERE SP11.sno = SP1.sno);
```

おっと、このSQL文ではまだ望む結果を得られない。1つでも共通する部品がある供給業者の組み合わせをすべて選択してしまうからだ。

その3

答え 答えその2で述べた伝統的な踏線で望む結果を得るためには、NOT EXISTS述語を使えばよい。

```
SELECT DISTINCT SP1.sno, SP2.sno
  FROM SupParts AS SP1, SupParts AS SP2
 WHERE SP1.sno < SP2.sno
   AND NOT EXISTS  -- SP1にはあるがSP2にはない部品
            (SELECT SP3.pno
               FROM SupParts AS SP3
              WHERE SP1.sno = SP3.sno
                AND SP3.pno NOT IN
                      (SELECT pno
                         FROM SupParts AS SP4
                        WHERE SP2.sno = SP4.sno))
   AND NOT EXISTS  -- SP2にはあるがSP1にはない部品
            (SELECT SP5.pno
               FROM SupParts AS SP5
              WHERE SP2.sno = SP5.sno
                AND SP5.pno NOT IN
                      (SELECT pno
                         FROM SupParts AS SP4
                        WHERE SP1.sno = SP4.sno));
```

その4

答え 私は、部分集合を使わずに集合の相等性を示す別解はないものかと考えた。まず、同じ業者同士の組み合わせを排除しつつ、共通の部品をキーとして供給業者同士を結合した。すると、2つの集合の共通部分が得られる。ここで、共通部分の要素数が2つの集合の要素数と一致すれば、2つの集合は等しいということになる。

```
SELECT SP1.sno, SP2.sno
  FROM SupParts AS SP1 INNER JOIN SupParts AS SP2
    ON SP1.pno = SP2.pno
   AND SP1.sno < SP2.sno
 GROUP BY SP1.sno, SP2.sno
HAVING COUNT(*) =
  (SELECT COUNT(*) -- 1対1の対応が存在するか否かのテスト
     FROM SupParts AS SP3
    WHERE SP3.sno = SP1.sno)
   AND COUNT(*) =
     (SELECT COUNT(*)
        FROM SupParts AS SP4
       WHERE SP4.sno = SP2.sno);
```

SupPartsテーブルのsno列にインデックスを設定すれば結合演算が速くなるし、カウントもインデックスから直接得られる。

その5

答え このSQL文は答えその4と同じ考え方で書かれている。COUNT(SP1.sno || SP2.sno)が、2つのSupPartsテーブルに同じ数だけ存在する供給業者番号を見つけるための“トリック”だ。

```
SELECT SP1.sno, SP2.sno
  FROM SupParts AS SP1 INNER JOIN SupParts AS SP2
    ON SP1.pno = SP2.pno
   AND SP1.sno < SP2.sno
 WHERE (SELECT COUNT(pno)
          FROM SupParts AS SP3
         WHERE SP3.sno = SP1.sno) = (SELECT COUNT(pno)
                                       FROM SupParts AS SP4
```

```
                                        WHERE SP4.sno = SP2.sno)
 GROUP BY SP1.sno, SP2.sno
HAVING COUNT(SP1.sno || SP2.sno) = (SELECT COUNT(pno)
                                      FROM SupParts AS SP3
                                     WHERE SP3.sno = SP1.sno);
```

その6

答え　次に紹介するのは、フランシスコ・モレノから寄せられた答えその3の変形版である。NOT EXISTS述語の代わりに差集合演算（EXCEPT演算子）を使っている。モレノはOracleを使用していて、その差集合演算子（OracleではMINUS）はかなり高速に実行されるという。

```
SELECT DISTINCT SP1.sno, SP2.sno
  FROM SupParts AS SP1, SupParts AS SP2
 WHERE SP1.sno < SP2.sno
   AND NOT EXISTS  -- SP1にはあるがSP2にはない部品
           (SELECT SP3.pno
              FROM SupParts AS SP3
             WHERE SP1.sno = SP3.sno
            EXCEPT
            SELECT SP4.pno
              FROM SupParts AS SP4
             WHERE SP2.sno = SP4.sno)
   AND NOT EXISTS  -- SP2にはあるがSP1にはない部品
           (SELECT SP5.pno
              FROM SupParts AS SP5
             WHERE SP2.sno = SP5.sno
            EXCEPT
            SELECT SP6.pno
              FROM SupParts AS SP6
             WHERE SP1.sno = SP6.sno);
```

その7

答え　アレクサンダー・クズネツォーフは、昔ながらの「結合内の一致を数える」アプローチに、再び工夫を凝らした。

```
SELECT A.sno AS sno1, B.sno AS sno2
  FROM (SELECT sno, COUNT(*), MIN(pno), MAX(pno)
```

```
          FROM SupParts GROUP BY sno)
      AS A(sno, cnt, min_pno, max_pno)
      INNER JOIN
      (SELECT sno, COUNT(*), MIN(pno), MAX(pno)
          FROM SupParts GROUP BY sno)
      AS B(sno, cnt, min_pno, max_pno)
  -- 以下の4つの条件でほとんどの組み合わせが除外される
       ON A.cnt = B.cnt
      AND A.min_pno = B.min_pno
      AND A.max_pno = B.max_pno
      AND A.sno < B.sno
  -- 以下の高コストなSELECT文は、すべての組み合わせについて
  -- 実行する必要はない
WHERE A.cnt
        = (SELECT COUNT(*)
             FROM SupParts AS A1,
                  SupParts AS B1
            WHERE A1.pno = B1.pno
              AND A1.sno = A.sno
              AND B1.sno = B.sno);
```

大抵のオプティマイザは、各列のMIN関数やMAX関数の値を統計テーブルに持っているので、結果を取得するのに時間はかからない。それを利用している点が、このクエリのうまいところだ。

その8

答え 最後に、集合の相等性をテストする一般的な2つの方法を、集合論の記号を使って確認しておこう。

$$((A \subseteq B) = (B \subseteq A)) \Rightarrow (A = B)$$
$$((A \cup B) = (A \cap B)) \Rightarrow (A = B)$$

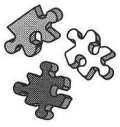
1つ目の等式は、まさに結合を使った比較の基礎となるものだ。2つ目の等式は、部分集合のレベルというより、集合のレベルで考えたものである。これを応用したクエリとしては、例えば次のようなテーブルAとテーブルBとの比較が考えられる。

```
SELECT DISTINCT 'not equal'  -- '等しくない'
  FROM ((SELECT * FROM  A
         UNION
         SELECT * FROM  B)
         EXCEPT
        (SELECT * FROM  A
         INTERSECT
         SELECT * FROM  B));
```

もしテーブルAとテーブルBが等しければ、空集合が返される。ただし、テーブルに重複行が存在する場合、集合演算子にALL句を付けるときには注意を払う必要がある。ありがたいことに、集合演算子は列レベルではなく行レベルで動作するため、このSQL文は、UNION互換[訳注3]なテーブルならどんなテーブルの組み合わせにも適用できる（つまりテンプレートとして使える）。比較したいテーブルが持っている列名は、全く知らなくてよい。

訳注3：UNION互換とは、UNION演算子で連結される2つのテーブルの間で「①列数が同じ」「②同じ位置に来る列のデータ型が同じ（または自動的に変換される）」であることを言う。②のデータ型の自動変換が行われるかどうかはRDBMSによる。

パ|ズ|ル|
28 正弦関数を作る
内挿法を行う



あなたの使っているRDBMSが、正弦（sine）関数を標準ライブラリに持っていないと仮定しよう。このとき、ラジアンに対する正弦値を計算することは可能だろうか？

その1

答え 単純にやるなら、ラジアンとそれに対応する正弦値を持つテーブルを作ればよい。

```
CREATE TABLE Sine
(x REAL NOT NULL,
 sin REAL NOT NULL);
```

```
INSERT INTO Sine
VALUES (0.00, 0.0000),
 ...
       (0.75, 0.6816),
       (0.76, 0.6889)
 ... ;
```

あとは表計算ソフトや充実した数学ライブラリを持つプログラミング言語の助けを借りて、テーブルにデータを入れるだけだ。このテーブルをスカラサブクエリ内で使うことで、ラジアンを入力とする正弦値が得られる。

```
(SELECT sin FROM Sine WHERE x = :myvalue)
```

もちろん、場合によってはこのテーブルはかなり巨大になってしまうだろうが、入力値の範囲が限られた関数であれば、比較的小さなサイズで済むだろう。これはこれで悪い方法ではない。ただ残念なことに、正弦関数は、すべての実数に対して定義された連続的な関数であるところが厄介だ。

その2

答え 答えその1のSELECT文で、:myvalueがテーブルになかった場合、このサブクエリは空、つまりNULLを返すことに気付いただろうか。これは良くない。

古い計算や三角法について書かれた本を読むと、計算機がなかった時代の人々がどのようにしてテーブルを使っていたかが分かる。彼らは「内挿法」と呼ばれる、なかなか味わい深い数学のテクニックを使っていた。

最も簡単な方法は線形内挿である。ある関数の2つの値、f(a)とf(b)が与えられた場合、その間に位置する第3の値を近似的に求めることができる。その公式は、

$$f(a) + (x - a) \times ((f(b) - f(a)) \div (b - a))$$

である。例えば、sin(0.754)を求めたい場合、次の2つの値を使う。

```
INSERT INTO Sine VALUES (0.75, 0.6816);
INSERT INTO Sine VALUES (0.76, 0.6889);
```

これを公式にあてはめてみよう。

```
0.6816 + (0.754 - 0.75) * ((0.6889 - 0.6816) / (0.76 - 0.75))
= 0.68452
```

実際の値は0.68456であるから、誤差は0.00004。ほとんどの場合、この推計値でも用をなすだろう。この"トリック"をクエリに組み込んだのが次の答えだ。

```
SELECT A.sin + (:myvalue - A.x)
       * ((B.sin - A.sin) / (B.x - A.x))
  FROM Sine AS A, Sine AS B
 WHERE A.x = (SELECT MAX(x) FROM Sine WHERE x <= :myvalue)
   AND B.x = (SELECT MIN(x) FROM Sine WHERE x >= :myvalue);
```

厳密には、関数の値域を0から2πまでに制限する条件が必要だが、まあ、それは些細なことだ。内挿法にはほかにも計算方法があるが、考え方は一緒である。

この問題が教える教訓は、SQLはテーブルと結合に対して有効に働くよう設計された言語であり、計算向きの言語ではないということだ。したがって、計算的な解法を探す前に、まずテーブルを使った解法を探すべきだ。

なお、内挿法の背後にある数学について知りたいという人には、J.F.シュテッフェンゼンの『Interpolation』(Dover Publications刊, 2006, ISBN978-0-486-45009-4)を推薦しておこう。

パズル 29

最頻値を求める

HAVING句の力（その3）

SQLで定義されている統計関数は単純平均、つまりAVGだけだ。これはよく使う統計ではあるが、統計関数はほかにもある。平均値（ミーン）、中央値（メジアン）、最頻値（モード）などもすべて、値の集合から「代表値」を求める方法だ。

最頻値とは、テーブルのある列において最も個数の多い値のことである。例えば、「給与名簿（Payroll）」という名前のテーブルに、check_nbr（給与小切手の番号）とそこに記載された給与額（check_amt）という2つの数値型の列があるとしよう。

```
CREATE TABLE Payroll
(check_nbr INTEGER NOT NULL PRIMARY KEY,
 check_amt DECIMAL(8,2) NOT NULL,
 ... );
```

Payroll：給料名簿テーブル　　check_nbr：給与小切手の番号　　check_amt：給与額

この給与名簿テーブルの中から給与額の最頻値と、そのレコード数を求めたい。では、そのクエリをSQL-89構文で書くにはどうすればよいだろう？ また、SQL-92構文やSQL-99構文ではどう書けるだろうか？

答え

その1

SQL-89には、SQL-92のような直交性[訳注1]がない。そのため、最善の方法はおそらくビューを使うことである。

```
CREATE VIEW AmtCounts
AS SELECT COUNT(*) AS check_cnt
     FROM Payroll
    GROUP BY check_amt;
```

そして、このビューを使って給与額の最頻値を検索する。

訳注1：直交性は非常に意味に幅のある多義的な用語だが、SQLに関連する意味としては、「演算の独立性」と考えてよい。SQL-92では、クエリの結果を1つの式と見なしてほかの演算の入力とするサブクエリ（表式）がサポートされ、演算の独立性（つまり直交性）が改善された。

```
SELECT check_amt, COUNT(*)
  FROM Payroll
 GROUP BY check_amt
HAVING COUNT(*) = (SELECT MAX(check_cnt)
                     FROM AmtCounts);
```

しかしこの解法だと、クエリの実行後にもビューをデータベースのスキーマに残してしまう。ほかの目的でこのビューを使う予定があるなら便利だが、そうでなければ邪魔だろう。ここはひとつ、ビューを使わずに1つのSQL文で実現したい。

その2

答え

SQL-92の直交性によって、ビューをテーブルサブクエリ式に折り込むことができるようになった。次のように書く。

```
SELECT check_amt, COUNT(*) AS check_cnt
  FROM Payroll
 GROUP BY check_amt
HAVING COUNT(*) = (SELECT MAX(check_cnt)
                     FROM (SELECT COUNT(*) AS check_cnt
                             FROM Payroll
                            GROUP BY check_amt));
```

一番内側のSELECT文は、1つ外側のSELECT文に集約されたテーブルを渡す前に、処理が完全に終わっている必要がある。その後に、真ん中のSELECT文がそこから最大値を見つけて、一番外側のSELECT文に渡す。この作業が行われる過程で、集約されたテーブルが解体される可能性も十分にある。

もしオプティマイザが優秀なら、最初に実行されるクエリを、最終的な解答を得るときに再利用するためにとっておくだろう。しかし、保証はできないので、実際の動作を確認してほしい。

その3

答え

今から示すSQL-92の別解は、前の解法とNULLの扱いが少し異なる。どんな違いがあるのか、分かるだろうか？

```
SELECT check_amt, COUNT(*) AS check_cnt
  FROM Payroll
 GROUP BY check_amt
HAVING COUNT(*) >= ALL (SELECT COUNT(*) AS check_cnt
                          FROM Payroll
                         GROUP BY check_amt);
```

この解法にメリットがあるとすれば、MAX関数を使っていないので、集約されたテーブルを複数のSELECT文の間で使い回せる可能性がより高い点だ。事実、サブクエリ内のSELECT文は、外側のSELECT文の射影である点に注目してほしい。

これらの解が、実際にあなたが使っているRDBMSでどの程度のパフォーマンスを発揮するのかは、各自で調べてみてほしい。

その4

答え

ご存知のように、最近では、将来的に実装するOLAP拡張の一部として最頻値を求めるための関数を用意しているRDBMSも多い。それを使えば、この問題を解くのは造作もない。サブクエリはあっさりとOLAP関数へ置き換えることができる[訳注2]。

```
SELECT check_amt,
       COUNT(*) OVER (PARTITION BY check_amt) AS check_cnt
  FROM Payroll;
```

もっとも、この書き方でパフォーマンスが向上するかどうかは、保証の限りでない。

訳注2：このクエリでは、check_amtごとの値の数を一覧表示するにとどまる。最終的には、check_cntが最も多いcheck_amtを選択しなければならない。

パ|ズ|ル|
30

買い物の平均サイクル

過去の直近の日付を求める

レイモンド・ピーターセンは、次のような質問をしてきた。

顧客の名前と購買日の2つの列だけを持つSalesテーブルがあるとする。ここから、各顧客の購買間隔が平均で何日かを1つのクエリで求める方法はあるだろうか？ なお、同じ顧客に対して同じ日に売上を2回以上記録することはないものとする。

```
CREATE TABLE Sales
(customer_name CHAR(5) NOT NULL,
 sale_date DATE NOT NULL,
 PRIMARY KEY (customer_name, sale_date));
```

Sales：販売テーブル	customer_name：顧客名	sale_date：販売日

次のデータは、1994年6月第1週の売上記録である。

```
Sales

customer_name   sale_date
============================
'Fred'          '1994-06-01'
'Mary'          '1994-06-01'
'Bill'          '1994-06-01'
'Fred'          '1994-06-02'
'Bill'          '1994-06-02'
'Bill'          '1994-06-03'
'Bill'          '1994-06-04'
'Bill'          '1994-06-05'
'Bill'          '1994-06-06'
'Bill'          '1994-06-07'
'Fred'          '1994-06-07'
'Mary'          '1994-06-08'
```

ここから分かることは、Fredは最初の買い物から次の買い物まで1日、その次に5日の間隔で買っていることが分かる。したがって、彼の平均購買間隔は3日だ。Maryは7日おきなので平均も7日、Billは規則的に毎日買っている。

その1

答え 真っ先にやろうとしたのは、顧客ごとに購買日の間隔日数を示す、手の込んだビューを作ることだった。このアプローチを採る場合には、初めにそれぞれの購買日に対する過去の直近の購買日を求める。

```
CREATE VIEW Lastsales
(customer_name, this_sale_date, last_sale_date)
AS SELECT S1.customer_name, S1.sale_date,
          (SELECT MAX(sale_date)
             FROM Sales AS S2
            WHERE S2.sale_date < S1.sale_date
              AND S2.customer_name = S1.customer_name)
     FROM Sales AS S1;
```

Lastsales：直近の販売ビュー　customer_name：顧客名　this_sale_date：今回の販売日
last_sale_date：直近の販売日

要するに、これは最大下界[訳注1]を求めるクエリである。ある購買日以前の購買日のうちで最大のもの（最も日付の新しいもの）が、過去の直近の購入日となるわけだ。

次に、現在の購買日と直近の購買日との間隔日数を持つビューを作る。これら2つのビューを1つのSQL文の中にまとめることもできなくはないが、読むに耐えないクエリになるし、うまく最適化されないだろう。ここではコードを簡単にするために、DAYS関数が使えるものとする。これは、2つの引数の日付の間隔を整数で返す[訳注2]。

```
CREATE VIEW SalesGap (customer_name, gap)
AS SELECT customer_name, DAYS(this_sale_date, last_sale_date)
     FROM Lastsales;
```

ここまで来れば、あとは1つのクエリで解ける。

訳注1：「下界」とは集合論の用語で、ある順序集合のどの要素よりも小さい要素のこと。例えば、{'1994-06-02', '1994-06-03', '1994-06-05'} という日付の集合に対しては、'1994-06-01' や '1994-05-20' などが下界になる。一般に、下界は複数あり得る。「最大下界」は、その名のとおり下界のうちで最大のもので、先の例では '1994-06-01' になる。「過去の直近日」は、日付における最大下界と言える。

訳注2：DAYS関数を持たないRDBMSでも、日付型の列同士を直接引き算すればよい。

```
SELECT customer_name, AVG(gap)
  FROM SalesGap
 GROUP BY customer_name;
```

2つのビューをAVG関数の引数に入れることもできるが、やったが最後、読解不能なクエリになってしまうし、処理は粘っこいハチミツのように遅くなるだろう。

答え

その2

答えその1を見せたのは、人間は1人で考えていると頭の中が暴走してしまう、という例を示したかったからだ。顧客の購買間隔の平均日数が求めるだけなら、わざわざ複雑なビューを作る必要などない。単純に、最初の購買日から直近の購買日までの経過日数を数えて、それを購買間隔の数で割ればよいのだ。

```
SELECT customer_name,
       DAYS(MAX(sale_date), MIN(sale_date)) / (COUNT(*) - 1)
       AS avg_gap
  FROM Sales
 GROUP BY customer_name
HAVING COUNT(*) > 1;
```

今日の日付と直近の購買日との間隔に時間まで考慮に入れなければ、購買間隔の数は(購買回数－1)なので、(COUNT(*)－1)で割っている。

HAVING句は、1回しか買い物をしなかった客を除外している。ただし、この1回限りの客も、SELECT句のMAX(sale_date)をCURRENT_DATEに変えることで結果に含められる。次のクエリはそのように書き直したものだ。

```
SELECT customer_name,
    CASE WHEN COUNT(*) > 1
         THEN DAYS(MAX(sale_date), MIN(sale_date)) / (COUNT(*)-1)
         ELSE DAYS(CURRENT_DATE, MIN(sale_date))
     END AS avg_gap
  FROM Sales
 GROUP BY customer_name;
```

うまいことに、答えその1とその2のどちらの方法も、同日に同一の顧客が2回以上買い物した場合にも対応できる。

パ|ズ|ル|
31

すべての製品を購入した顧客

関係除算の応用

ソフトウェアAG社[訳注1]は、1990年代の中ごろに「Esperant」という優秀なSQL作成ソフトを発表した。ユーザーは、キーボードと対話式の選択リストを使って英文を作り、Esperantがそれを一連のSQLクエリに変換する。

確かに、自然言語によるクエリというのは古いアイデアである。しかし、そうした製品の多くは、英語のフレーズがクエリとして動作するように、あらかじめいくらかプログラムしておく必要があった。それらと比べると、Esperantが単独で実行可能な処理の数には注目すべきものがあった。Esperantは、事前のプログラミングなしに関係除算やビューの作成を行えたし、複雑なトランザクションを組むこともできた。

ソフトウェアAG社のデモには、次に示すような顧客、注文、注文明細（およびその他もろもろ）の典型的なスキーマが含まれていた。

```
CREATE TABLE Customers
(customer_id INTEGER NOT NULL PRIMARY KEY,
 acct_balance DECIMAL (12, 2) NOT NULL,
 ... );
```

Customers：顧客テーブル　　customer_id：顧客ID　　acct_balance：売掛金残高

```
CREATE TABLE Orders
(customer_id INTEGER NOT NULL,
 order_id INTEGER NOT NULL PRIMARY KEY,
 ... );
```

Orders：注文テーブル　　customer_id：顧客ID　　order_id：注文ID

```
CREATE TABLE OrderDetails
(order_id INTEGER NOT NULL,
 item_id INTEGER NOT NULL,
 item_qty INTEGER NOT NULL,
 PRIMARY KEY (order_id, item_id),
 ... );
```

訳注1：ドイツの大手ソフトウェア会社。

OrderDetails：注文明細テーブル　order_id：注文ID　item_id：製品ID
item_qty：注文数量

```
CREATE TABLE Products
(item_id INTEGER NOT NULL PRIMARY KEY,
 item_qty_on_hand INTEGER NOT NULL,
 ... );
```

Products：製品テーブル　item_id：製品ID　item_qty_on_hand：現存数量

このテーブル群を使ったサンプル問題の1つに、すべての製品を購入した顧客全員の売掛金残高（acct_balance）の平均と、すべてではないがいくつかの製品を購入した顧客全員の売掛金残高（acct_balance）の平均を求める、というものがある。Esperantは、この問題に対しても素晴らしい能力を発揮したが、ほかの環境でもクエリを動かすために大量のビューを作ってしまっていた。

さて、あなたはSQL-92の機能を使って（あるいは昔ながらのSQL-89の機能を駆使して）、うまいクエリを作れるだろうか？

その1

答え　伝統的な解答としては、ネスト（入れ子構造）の深いクエリを使うものが挙げられる。このクエリを日本語に訳すなら、「顧客が注文していない製品が製品テーブルに存在するような、顧客集合の売掛金残高の平均を求めよ」である。

```
SELECT AVG(acct_balance)
  FROM Customers AS C1
 WHERE EXISTS
        (SELECT *
           FROM Products AS P1
          WHERE P1.item_id
            NOT IN (SELECT D1.item_id
                      FROM Orders AS O1, OrderDetails AS D1
                     WHERE O1.customer_id = C1.customer_id
                       AND O1.order_id = D1.order_id));
```

このクエリで、いくつかの製品を購入した顧客の平均残高が求められる[訳注2]。すべての製品を購入した顧客の平均残高を求めたければ、EXISTSの代わりにNOT EXISTSを使えばよい。

その2

答え　英国ウスターシャー州に住むギリアン・ロバートソンは、相関サブクエリのネストを少し省くことのできるきれいな"トリック"を見つけた。

```
SELECT AVG(acct_balance)
  FROM Customers AS C1
 WHERE (SELECT COUNT(DISTINCT item_id) -- 販売中の全製品
          FROM Products)               --        対
    <> (SELECT COUNT(DISTINCT item_id) -- 顧客が買った製品
          FROM Orders, OrderDetails
          WHERE Orders.customer_id = C1.customer_id
            AND Orders.order_id = OrderDetails.order_id);
```

このクエリでは、注文明細に存在する製品をDISTINCT付きでカウントした数と、製品リストをDISTINCT付きでカウントした数が一致しなければ、その発注をした顧客はすべての製品を購入したのではない、という事実を利用している。また、「<>」を「=」に変えることで、反対にすべての製品を購入した顧客が分かる。

その3

答え　アレックス・クズネツォーフは、すべての製品を購入した顧客と、一部の製品だけを購入した顧客の両方に対する答えが必要な点に注目して、2つの答えを1つのクエリで求められるようにした。これは2つのクエリを別々に実行するよりも簡単な上、パフォーマンスも良い[訳注3]。

訳注2：ただし、何も注文していない顧客がCustomersテーブルに存在する場合、その顧客も含まれるので注意。これは答えその2も同様である。

訳注3：前の2つの解とは違い、この解は注文したことのない顧客がCustomersテーブルに存在する場合に、その顧客を除外する。

```
SELECT AVG(acct_balance), ordered_all_desc
  FROM (SELECT Customers.customer_id, acct_balance,
               CASE WHEN num_ordered_products = all_product_cnt
                    THEN 'ordered all'
                    ELSE 'not ordered all' END AS ordered_all_desc
          FROM Customers
         INNER JOIN
           (SELECT customer_id,
                   COUNT(DISTINCT item_id) num_ordered_products
              FROM Orders INNER JOIN OrderDetails
                ON Orders.order_id = OrderDetails.order_id
             GROUP BY customer_id ) AS ordered_products
          ON Customers.customer_id = ordered_products.customer_id
         CROSS JOIN
           (SELECT COUNT(DISTINCT item_id)
              FROM Products) AS AllProducts (all_product_cnt)
 ) AS T
GROUP BY ordered_all_desc;
```

パ|ズ|ル|
32 税金の計算
木構造／階層構造を扱う

リチャード・レムレーはある問題をCompuServe経由で送ってきた。それは、税金の計算に関する問題を簡略化したものだった。

まず、用語をいくつか定義しよう。複数の税務当局（tax_authority）が集まって、課税地域（tax_area）を形成している。例えば、課税地域が市（city）である場合、その税務当局は市や市を含む郡（county）や州（state）であったりする。市内での買い物には、市税、郡税、州税の3つから構成される税金がかかる。税率は各当局が独自に変更できる。

さて、次のようなテーブルがあるとしよう。

```
CREATE TABLE TaxAuthorities
(tax_authority CHAR(10) NOT NULL,
 tax_area CHAR(10) NOT NULL,
 PRIMARY KEY (tax_authority, tax_area));
```

TaxAuthorities：税務当局テーブル　　tax_authority：税務当局　　tax_area：課税地域

各課税地域での納税先となる複数の税務当局は、次のように階層を成している。

```
TaxAuthorities

tax_authority   tax_area
=======================
'city1'         'city1'
'city2'         'city2'
'city3'         'city3'
'county1'       'city1'
'county1'       'city2'
'county2'       'city3'
'state1'        'city1'
'state1'        'city2'
'state1'        'city3'
```

このデータが意味するところは、city1とcity2はcounty1に属し、county1はstate1に属している。またcity3はcounty2に属し、county2はstate1に属する、ということである。税金の計算にもう1つ必要なのが、次に示す税率テーブルである。ここでは、税率は単純に足し算で求められることとしよう。

```
CREATE TABLE TaxRates
(tax_authority CHAR(10) NOT NULL,
 effect_date DATE NOT NULL,
 tax_rate DECIMAL (8,2) NOT NULL,
 PRIMARY KEY (tax_authority, effect_date));
```

TaxRates：税率テーブル　　tax_authority：税務当局　　effect_date：実効日
tax_rate：税率

このテーブルには次のようなデータを入れる。

```
TaxRates

tax_authority   effect_date    tax_rate
=======================================
'city1'         '1993-01-01'        1.0
'city1'         '1994-01-01'        1.5
'city2'         '1993-09-01'        1.5
'city2'         '1994-01-01'        2.0
'city2'         '1995-01-01'        2.0
'city3'         '1993-01-01'        1.7
'city3'         '1993-07-01'        1.9
'county1'       '1993-01-01'        2.3
'county1'       '1994-10-01'        2.5
'county1'       '1995-01-01'        2.7
'county2'       '1993-01-01'        2.4
'county2'       '1994-01-01'        2.7
'county2'       '1995-01-01'        2.8
'state1'        '1993-01-01'        0.5
'state1'        '1994-01-01'        0.8
'state1'        '1994-07-01'        0.9
'state1'        '1994-10-01'        1.1
```

このテーブルは、徴税官からの「1994年11月1日のcity2における合計税率はいくらか？」といった問いに答えるために使われる。答えは、次のように求める。

city2の税率　　= 2.0
county1の税率 = 2.5
state1の税率　 = 1.1

税率合計　　= 5.6

この徴税官からの質問に答えるクエリを、1つの文で書くことができるだろうか？

その1

答え　この問題を解く一番よい方法は、問題を分解することである。まずは、cityに対する税務当局がどこかを見つけたい。これは次のサブクエリで取得できる。

```
(SELECT tax_authority
   FROM TaxAuthorities AS A1
  WHERE A1.tax_area = 'city2')
```

このクエリの結果は {'city2', 'county1', 'state1'} という集合になる。

次に、1994年11月1日における各当局の税率が知りたい。これはもう1つのサブクエリを書くことで分かる。

```
(SELECT tax_authority, tax_rate
   FROM TaxRates AS R1
  WHERE R1.effect_date = (SELECT MAX(R2.effect_date)
                            FROM TaxRates AS R2
                           WHERE R2.effect_date <= '1994-11-01'))
```

ここまで来たら、2つのサブクエリを結合し、税率を足し、定数をSELECT句のリストに入れれば最終的な解答ができあがる。実際には、この定数をパラメータ入れて汎用的に使えるようにするのが好ましいが、ここは問題の要求に忠実に従おう。

```
SELECT 'city2' AS city, '1994-11-01' AS effect_date,
       SUM(tax_rate) AS total_taxes
  FROM TaxRates AS R1
 WHERE R1.effect_date =
       (SELECT MAX(R2.effect_date)
          FROM TaxRates AS R2
         WHERE R2.effect_date <= '1994-11-01'
           AND R1.tax_authority = R2.tax_authority)
   AND R1.tax_authority IN
       (SELECT tax_authority
          FROM TaxAuthorities AS A1
         WHERE A1.tax_area = 'city2')
GROUP BY tax_authority, effect_date;
```

これで一応完成だが、安心するのはまだ早い！ このクエリはもっと改善できる。2番目のAND述語は、もう一段深いレベルのネストへ移せる。

```
SELECT 'city2' AS city, '1994-11-01' AS effective_date,
       SUM(tax_rate) AS total_taxes
  FROM TaxRates AS R1
 WHERE R1.effect_date =
       (SELECT MAX(R2.effect_date)
          FROM TaxRates AS R2
         WHERE R2.effect_date <= '1994-11-01'
           AND R1.tax_authority = R2.tax_authority
           AND R2.tax_authority IN
               (SELECT tax_authority
                  FROM TaxAuthorities AS A1
                 WHERE A1.tax_area = 'city2'))
 GROUP BY tax_authority, effect_date;
```

このサブクエリは非相関なので、定数のリストになる。そのため、パフォーマンスはかなり高いはずだ。思ったとおり、私がWATCOM SQLで実行計画を見たところ、R1およびR2テーブルはシーケンシャルにスキャンされていたが、A1テーブルは主キーを使って読み出していた。TaxRatesテーブルにインデックスを作成すれば、もっと速くなるだろう。

その2

答え 米国ワシントン州のディオスダージョ・ネブレは、別解を送ってきた。

```
SELECT SUM(T2.tax_rate)
  FROM TaxAuthorities AS T1, TaxRates AS T2
 WHERE T1.tax_area = 'city2'
   AND T2.tax_authority = T1.tax_authority
   AND T2.effect_date =
       (SELECT MAX(effect_date)
          FROM TaxRates
         WHERE tax_authority = T2.tax_authority
           AND effect_date <= '1994-11-01');
```

彼はGROUP BY句を取り除いたが、これはよい修正である。なぜなら、GROUP BYがなくても結果的には同じだからである[訳注1]。さらに彼は、一番深いレベルのネストを

TaxAuthorityテーブルとTaxRatesテーブルの結合で置き換えた。これによって、最初のサブクエリの実行時間が大幅に削減される。

その3

答え 最近のSQLプログラマなら、税務当局が階層構造を成しており、それゆえ入れ子集合モデル[訳注2]が使えることに気づくだろう。ここでは、入れ子集合とその(lft, rgt)の組み合わせの使い方についての説明は割愛する(詳細は拙著『Joe Celko's Trees and Hierarchies in SQL for Smarties』Morgan Kaufmann刊, ISBN978-1-55860-920-4を参照してほしい)。とにかく、2つのテーブルは次の1つのテーブルで置き換えられる。

```
CREATE TABLE TaxRates
(tax_authority CHAR(10) NOT NULL,
 lft INTEGER NOT NULL CHECK (lft > 0),
 rgt INTEGER NOT NULL,
   CHECK (lft < rgt),
 start_date DATE NOT NULL,
 end_date DATE, -- NULLならば現在の税率を意味する
 tax_rate DECIMAL(8,2) NOT NULL,
 PRIMARY KEY (tax_authority, start_date));
```

start_dateとend_dateは、税率の有効期間を示す。この期間の重複は許されない。税率の有効期間を格納する理由は、過去の税率の計算を簡単にするためである。

```
SELECT SUM(DISTINCT T2.tax_rate)
  FROM TaxRates AS T1, TaxRates AS T2
 WHERE T1.tax_authority = :my_location
   AND :my_date BETWEEN T2.start_date
                    AND COALESCE (T2.end_date, CURRENT_DATE)
   AND T1.lft BETWEEN T2.lft AND T2.rgt;
```

COALESCE関数は、税率の未来の失効日が不明の場合でも、現行の税率を扱えるようにするために必要である。

訳注1：実際には、GROUP BYを削除したことで結果が微妙に変わる。答えその1は税務当局ごとの税率を3行戻すので、その後に合計する必要があったのに対し、答えその2では、それらを合計した税率(5.6)の1行だけが戻される。

訳注2：通常1次元の点と見なされる木構造のノードを、面積を持つ2次元の「円」(集合)と見なし、その左端と右端の座標を(lft, rgt)で表現するモデル。書籍『プログラマのためのSQL 第2版』の第29章が参考になる。

パズル 33

機械の平均使用コスト

複雑な条件での集約計算（平均値）

このパズルは、ゲルハルト・ジロヴェックがCompuServeに投稿した問題がもとになっている。彼は製造会社のデータベースを管理しており、そこから機械の時間あたりの平均使用コストを計算したいと考えた。そこで、彼は次のような機械を管理するためのテーブルを作成した。

```
CREATE TABLE Machines
(machine_name CHAR(20) NOT NULL PRIMARY KEY,
 purchase_date DATE NOT NULL,
 initial_cost DECIMAL(10,2) NOT NULL,
 lifespan INTEGER NOT NULL);
```

Machines：機械テーブル	machine_name：機械名	purchase_date：購入日
initial_cost：初期費用	lifespan：耐用期間	

purchase_date列には購入日が、initial_cost列には初期費用が格納される。lifespan列の値はその機械の期待される寿命であり、日数で表す。

もう1つ、ある機械をあるバッチ（一連の作業）で使用したときにかかる費用を格納するテーブルも作成した。

```
CREATE TABLE ManufactCosts
(machine_name CHAR(20) NOT NULL
   REFERENCES Machines(machine_name),
 manu_date DATE NOT NULL,
 batch_nbr INTEGER NOT NULL,
 manu_cost DECIMAL(6,2) NOT NULL,
 PRIMARY KEY (machine_name, manu_date, batch_nbr));
```

ManufactCosts：製造費用テーブル	machine_name：機械名	manu_date：稼働日
batch_nbr：バッチ番号	manu_cost：稼働費用	

manu_date列には実際に機械をバッチに使用した日付が、manu_cost列にはそのバッチに必要な費用が格納される。これとよく似たテーブルで、各バッチにかかった時間を格納するテーブルもある。構造は次のとおりだ。

```
CREATE TABLE ManufactHrs
(machine_name CHAR(20) NOT NULL REFERENCES Machines,
 manu_date DATE NOT NULL,
 batch_nbr INTEGER NOT NULL,
 manu_hrs DECIMAL(4,2) NOT NULL,
 PRIMARY KEY (machine_name, manu_date, batch_nbr));
```

ManufactHrs：稼働時間テーブル	machine_name：機械名	manu_date：稼働日
batch_nbr：バッチ番号	manu_hrs：稼働時間	

今回の問題は、このデータベース設計の改善策を提案することと、その設計をもとに、任意の日付の各機械の1時間あたりの平均費用を求めるクエリを書くことである。

その1

答え

オリジナルの設計では、時間と金額が別々のテーブルに格納されていた。機械の使用日が、タイムカードと会計部署という異なる情報源から別々に集められていたから、というのが理由だ。

まずは、稼働費用（manu_cost）と稼働時間（manu_hrs）を、（machine_name, manu_date, batch_nbr）を主キーとする1つのテーブルにまとめるべきだ。もし、費用は分からないが時間だけは分かっている、あるいは時間は分からないが費用だけは分かっているというケースがあり得るなら、この2つの列にNULLを許す設計になるかもしれない。だが、ここはよくよく考えたほうがよい。私なら、2つのテーブルを次の1つのテーブルで置き換える。

```
CREATE TABLE ManufactHrsCosts
(machine_name CHAR(20) NOT NULL
   REFERENCES Machines(machine_name),
 manu_date DATE NOT NULL,
 batch_nbr INTEGER NOT NULL,
 manu_hrs DECIMAL(4,2) NOT NULL,
 manu_cost DECIMAL(6,2) NOT NULL,
 PRIMARY KEY (machine_name, manu_date, batch_nbr));
```

では、いくつかサンプルデータを入れてみよう。5日前にFrammisカッターを1万ドルで買い、今までに7つのバッチに使った。Frammisカッターの寿命は1000日である。

```
ManufactHrsCosts

machine_name  manu_date   batch_nbr manu_hrs  manu_cost
=================================================
'Frammis'     '1995-07-24'       101       2.5     123.00
'Frammis'     '1995-07-25'       102       2.5     120.00
'Frammis'     '1995-07-25'       103       2.0     100.00
'Frammis'     '1995-07-26'       104       2.5     118.00
'Frammis'     '1995-07-27'       105       2.5     116.00
'Frammis'     '1995-07-27'       106       2.5     113.00
'Frammis'     '1995-07-28'       107       2.5     110.00
```

最初に使ったのは7月24日。この日の時間あたりの平均費用は(123.00 + 10.00)ドル÷2.5時間 = 53.20ドル／時。しかし、2日目の7月25日では、(123.00 + 120.00 + 100.00 + (2 × 10.00))ドル÷(2.5 + 2.5 + 2.0)時間 = 51.86ドル／時となり、5日間の最終日には1時間あたりの費用は50ドル／時にまで下がる。

いくつかの方法が考えられるが、私は毎日の総費用と総時間のビューを作るのがよいと思う。このビューは、ほかの通常業務でレポートを作成する際にも使える。

```
CREATE VIEW TotHrsCosts
(machine_name, manu_date, day_cost, day_hrs)
AS SELECT machine_name, manu_date, SUM(manu_cost), SUM(manu_hrs)
     FROM ManufactHrsCosts
    GROUP BY machine_name, manu_date;
```

2つのDATE型の変数を引き算することで、間隔日数を計算できると仮定しよう。このビューができれば、クエリはシンプルになる。

```
SELECT :mydate, M1.machine_name,
        (SELECT ((initial_cost / lifespan)
                  * (:mydate - M1.purchase_date + 1)
                  + SUM(THC.day_cost)) / SUM(THC.day_hrs)
           FROM TotHrsCosts AS THC
          WHERE M1.machine_name = THC.machine_name
            AND :mydate >= M1.purchase_date
            AND :mydate >= THC.manu_date) AS hourly_cost
  FROM Machines AS M1;
```

WHERE句の述語に注目してほしい。これは、1時間あたりの費用計算において、最初の部分に負数が入るのを避けるうまい“トリック”だ。

その2

答え フランシスコ・モレノは、彼がコロンビアで学生だったころにこの別解を見つけた。この方法では、ビューとスカラサブクエリを使わない。

```
SELECT :mydate AS my_date, F.machine_name,
       (MAX((initial_cost / lifespan)
            * (:mydate - purchase_date + 1)) + SUM(manu_cost))
        / SUM(manu_hrs) AS average_hour
  FROM ManufactHrsCosts AS F, Machines AS M
 WHERE M.machine_name = F.machine_name
   AND purchase_date <= :mydate
   AND manu_date <= :mydate
 GROUP BY F.machine_name;
```

パズル 34

コンサルタントの請求書

複雑な条件での集約計算（期間データ）

1994年11月、ブライアン・K・バックリーは次のような類の問題を投稿して、助けを求めてきた。彼は次のような3つのテーブルを持っていた。

```
CREATE TABLE Consultants
(emp_id INTEGER NOT NULL,
 emp_name CHAR(10) NOT NULL);
```

Consultants：コンサルタントテーブル　　emp_id：社員ID　　emp_name：社員名

```
INSERT INTO Consultants
VALUES (1, 'Larry'),
       (2, 'Moe'),
       (3, 'Curly');
```

```
CREATE TABLE Billings
(emp_id INTEGER NOT NULL,
 bill_date DATE NOT NULL,
 bill_rate DECIMAL (5,2));
```

Billings：請求書テーブル　　emp_id：社員ID　　bill_date：請求日　　bill_rate：時給

```
INSERT INTO Billings
VALUES (1, '1990-01-01', 25.00),
       (2, '1989-01-01', 15.00),
       (3, '1989-01-01', 20.00),
       (1, '1991-01-01', 30.00);
```

```
CREATE TABLE HoursWorked
(job_id INTEGER NOT NULL,
 emp_id INTEGER NOT NULL,
 work_date DATE NOT NULL,
 bill_hrs DECIMAL(5, 2));
```

HoursWorked：実働時間テーブル　　job_id：仕事ID　　emp_id：社員ID
work_date：実働日　　bill_hrs：実働時間

```
INSERT INTO HoursWorked
VALUES (4, 1, '1990-07-01', 3),
       (4, 1, '1990-08-01', 5),
       (4, 2, '1990-07-01', 2),
       (4, 1, '1991-07-01', 4);
```

バックリー氏が欲しかったのは、各仕事に対するコンサルタントの名前と総請求額を出力する単一のクエリだった。コンサルタントの総請求額は、各労働時間と時給とを掛け算し、それらを合計することで得られる。それだけなら話は簡単だが、コンサルタントの時給が期間によって変動する点にこの問題の難しさがある。時給には実働日以前の直近の請求日の金額を使う。サンプルデータからは、次のような答えが導かれる。

```
name      totalcharges
======================
'Larry'          320.00
'Moe'             30.00
```

Larryは、((3 + 5) 時間×時給25ドル + 4時間×時給30ドル) で320.00ドル、Moeは、(2時間×時給15ドル) = 30.00ドルという計算だ。

その1

答え　私が考える最善の方法は、ビューを作り、それから総計を求めることだ。このビューはほかのレポートでも役立つだろう。

```
CREATE VIEW HourRateRpt
(emp_id, emp_name, work_date, bill_hrs, bill_rate)
AS SELECT H1.emp_id, emp_name, work_date, bill_hrs,
          (SELECT bill_rate
            FROM Billings AS B1
           WHERE bill_date = (SELECT MAX(bill_date)
                                FROM Billings AS B2
                               WHERE B2.bill_date <= H1.work_date
                                 AND B1.emp_id = B2.emp_id
                                 AND B1.emp_id = H1.emp_id))
     FROM HoursWorked AS H1, Consultants AS C1
    WHERE C1.emp_id = H1.emp_id;
```

レポートの作成は簡単だ。

```
SELECT emp_id, emp_name, SUM(bill_hrs * bill_rate) AS bill_tot
  FROM HourRateRpt
 GROUP BY emp_id, emp_name;
```

しかし、バックリー氏はすべてを1つのクエリで処理したいということだった。彼の要求に応えるならこうなる。

```
SELECT C1.emp_id, C1.emp_name, SUM(bill_hrs *
        (SELECT bill_rate
           FROM Billings AS B1
          WHERE bill_date = (SELECT MAX(bill_date)
                               FROM Billings AS B2
                              WHERE B2.bill_date <= H1.work_date
                                AND B1.emp_id = B2.emp_id
                                AND B1.emp_id = H1.emp_id)))
 FROM HoursWorked AS H1, Consultants AS C1
WHERE H1.emp_id = C1.emp_id
GROUP BY C1.emp_id, C1.emp_name;
```

これは新米のSQLプログラマにとって、自明とは言いがたい答えだろうから、少し説明しておこう。まず一番内側のSELECT文からだ。ここでは、各社員について、請求日に直近の実効日を取得する。次のレベルのサブクエリは、その時点で社員に対して有効な時給を見つける。外側の相関名B1を使っているのはこのためだ。そして、戻された時給をSUM関数で合計した時間数と掛ける。最後に一番外側のクエリが、社員ごとの請求額をグループ化して、総額を作成する。

その2

答え リン・グエンは次のような別解を送ってきた。

```
SELECT emp_name, SUM(H1.bill_hrs * B1.bill_rate)
  FROM Consultants AS C1, Billings AS B1, Hoursworked AS H1
 WHERE C1.emp_id = B1.emp_id
   AND C1.emp_id = H1.emp_id
   AND bill_date = (SELECT MAX(bill_date)
                      FROM Billings AS B2
                     WHERE B2.emp_id = C1.emp_id
```

```
                        AND B2.bill_date <= H1.work_date)
                        AND H1.work_date >= bill_date
GROUP BY emp_name;
```

このクエリは、しばしばパフォーマンスを悪化させるサブクエリ式に頼らないという点で、最初の解より優れている。この話の教訓は、「新機能[訳注1]に頼りすぎると、かえってSQL文をややこしくすることがある」ということだ。

訳注1：答えその1のSELECT句で使っているスカラサブクエリを指している。

パ|ズ|ル|
35

在庫調整

再帰集合で累計を求める

このパズルは、SQL-92では簡単に解けるが、SQL-89で解くのは非常に難しい。

さて、あなたは会社の在庫管理部門の責任者だと思ってほしい。あなたは、いつ何個の部品が倉庫に入庫、または出庫されたかを記録した用紙を受け取った。記録されている数は、正数（入庫）のこともあれば負数（出庫）のこともある。

```
CREATE TABLE InventoryAdjustments
(req_date DATE NOT NULL,
 req_qty INTEGER NOT NULL
   CHECK (req_qty <> 0),
 PRIMARY KEY (req_date, req_qty));
```

InventoryAdjustments：在庫調整テーブル　　req_date：要求日　　req_qty：要求数量

あなたの仕事は、SQLで在庫の部品残数を累計的に求めることである。結果は、例えば次のような形になる。

```
Warehouse

req_date        req_qty    req_onhand_qty
=========================================
'1994-07-01'        100               100
'1994-07-02'        120               220
'1994-07-03'       -150                70
'1994-07-04'         50               120
'1994-07-05'        -35                85
```

Warehouse：倉庫　　req_onhand_qty：在庫の部品残数（累計）

答え

その1

SQL-92ではSELECT句でサブクエリが使える。そのサブクエリは相関サブクエリでもかまわない。ルールは、サブクエリの返す値が単一値であること（それゆえ「スカラサブクエリ」と呼ばれる）。もし、クエリの結果が空集合の場合、結果はNULLになる。

SQL-92のこの面白い機能を使うと、外部結合をSELECT句で書くことができる。例えば、次のクエリは顧客が注文を1つだけしているか、もしくは全く注文していない場合に限ってうまく動作する（全く注文していない場合、サブクエリはNULLを返す）。

```
SELECT cust_nbr, cust_name,
       (SELECT order_amt
          FROM Orders
         WHERE Customers.cust_nbr = Orders.cust_nbr)
  FROM Customers;
```

次のように書いても同じ結果が得られる。

```
SELECT cust_nbr, cust_name, order_amt
  FROM Customers
       LEFT OUTER JOIN Orders
       ON Customers.cust_nbr = Orders.cust_nbr;
```

このパズルでは、発行された注文用紙のすべてを足し上げて、問題となるデータをすべて含む必要がある。そのため、クエリでは次のように入れ子の自己結合を使わなければならない[訳注1]。

```
SELECT req_date, req_qty,
       (SELECT SUM(req_qty)
          FROM InventoryAdjustments AS A2
         WHERE A2.req_date <= A1.req_date) AS req_onhand_qty
  FROM InventoryAdjustments AS A1
 ORDER BY req_date;
```

訳注1：このクエリでは、カレント行の日付（A1.req_date）を起点として、それよりも前の日付（A2.req_date）の集合を作ることで累計を得ている。その集合は、次のような入れ子の再帰集合になる。

```
S0：{'1994-07-01'}
S1：{'1994-07-01', '1994-07-02'}
S2：{'1994-07-01', '1994-07-02', '1994-07-03'}
...
```

このタイプの再帰集合の作り方のアイデアの源泉は、数学者フォン・ノイマンによる集合を使った自然数の定義までさかのぼる。なお、このクエリのSUMをCOUNTに変えればランキングの算出にも利用できる（ランキングのほうがよりノイマンの考えに近い）。実は、パズル31で直近の日付を求めているクエリもSUMをMAXを変えただけである。

正直に言うと、レコードがソートされて記録されているファイルを使い、現在の在庫残数を直前の在庫残数から求められる手続き的な方法に比べれば、この解法は非常に遅いだろう。

その2

答え トライデントデータシステムに勤務するジム・アームスは、最初の解法より少し簡単な方法を思いついた。

```
SELECT A1.req_date, A1.req_qty,
       SUM(A2.req_qty) AS req_onhand_qty
  FROM InventoryAdjustments AS A1, InventoryAdjustments AS A2
 WHERE A2.req_date <= A1.req_date
 GROUP BY A1.req_date, A1.req_qty
 ORDER BY A1.req_date;
```

確かにこのクエリでもうまくいくが、反面、処理はとても重たい。例えば、テーブルにn行あるとしよう。大抵のRDBMSでは、GROUP BY句の処理でソートが発生する。このクエリのGROUP BY句は処理日ごとに実行されるため、1日目のグループについては1行、2日目のグループについては2行……最終日に属するグループはn行のソートを行うことになる。パフォーマンスが出ないのは明らかだ。

これに対し、先の「SELECT句の中にSELECT文を書く」方法では、GROUP BY句がないのでソートは発生しない。req_date列にインデックスが作成されていなければ、サブクエリを使ったクエリは、各処理日についてGROUP BYを使うクエリと同じだけのテーブルスキャンを行うが、その際に累計も保持できるだろう。したがって、「SELECT句の中にSELECT文」の方法では、テーブル検索がある程度減ることを期待できる。

その3

答え SQL:2003規格の標準SQLには、累計を求めるOLAP関数が導入されている。SQL-92ではスカラサブクエリで処理していたことを、関数で書けるようになったわけだ。規格では移動累計を求めるMOVING_SUMオプションまで提案されているが、まだ広く利用できる機能ではない。

```
SELECT req_date, req_qty,
       SUM(req_qty) OVER (ORDER BY req_date ASC
                          ROWS UNBOUNDED PRECEDING)
           AS req_onhand_qty
  FROM InventoryAdjustments
 ORDER BY req_date;
```

このSQL文はかなりコンパクトだし、しかも、何をしているのかを明確に表している。カレント行の処理要求日を1つ取り、それについて日付の降順で並べた以前の処理数量の合計を求めている。これは、先ほどのスカラサブクエリと同じ働きをする。あなたはどちらが読みやすいと思うだろうか。また、どちらならメンテナンスしてもよいと思うだろうか。

一言補足すると、OVERというウィンドウ句はそのままにして、SUMの代わりにAVGなどの集約関数を使うこともできる。本書の執筆時点（2006年）では、これらはまだ比較的新しい機能のため、実際のRDBMSでどの程度処理が最適化されるのかはよく分からない。

パ|ズ|ル|
36 1人2役
CASE式の中に集約関数を組み込む（その2）

CompuServeの初期、ナイジェル・ブルーメンタールがあるアプリケーションで立ち往生していると言ってこの問題を投稿してきた。

ここに、社員が会社の中で担当している役職を格納するテーブルがあるとしよう。例えば、'D'は重役（Director）、'O'は役員（Officer）といった具合だ。とりあえず、コードはこの2つだけ知っていればよい。

さて、やりたいことは重役と役員、ならびに2役を兼務する人物（コード'B'で表す）をレポートに出力することである。次に示すのが、必要最小限にまで絞り込んだ役職に関するデータである。

```
Roles

person    role
==============
'Smith'   'O'
'Smith'   'D'
'Jones'   'O'
'White'   'D'
'Brown'   'X'
```

Roles：役職テーブル　　person：社員　　role：役職

そして、そこから求めたい結果はこうだ。

```
person    combined_role
======================
'Smith'   'B'
'Jones'   'O'
'White'   'D'
```

combined_role：兼務する役職

ナイジェルは最初、一時テーブルを作る方法を試したのだが、これは処理時間が非常にかかってしまったという。

その1

答え

この質問に対して、ロイ・ハーヴェイは反射的に「集約クエリを使えばいいぞ」と――あまり深く考えずに――提案してきた。だが、2役をこなす人物以外に、'D'だけや'O'だけの人物も出力したいのだ。そこで、ハーヴェイの基本的な考えを発展させたのが次の解だ。

```
SELECT R1.person, R1.role
  FROM Roles AS R1
 WHERE R1.role IN ('D', 'O')
 GROUP BY R1.person
HAVING COUNT(DISTINCT R1.role) = 1
UNION
SELECT R2.person, 'B'
  FROM Roles AS R2
 WHERE R2.role IN ('D', 'O')
 GROUP BY R2.person
HAVING COUNT(DISTINCT R2.role) = 2;
```

R1.roleはGROUP BY句に含まれていないため、実装によってはMAX(R1.role)と書く必要があるかもしれない。また、この解でも欲しい結果を得ることはできるが、2つの集約クエリを実行するのは無駄なオーバーヘッドを発生させる。ほかによい考えはないだろうか。

その2

答え

レオナルド・C・メダルはビューを使うことで、集約した際の一時テーブルの問題を回避できるクエリを送ってきた。

```
SELECT DISTINCT R1.person,
       CASE WHEN EXISTS (SELECT COUNT(*)
                           FROM Roles AS R2
                          WHERE R2.person = R1.person
                            AND R2.role IN ('D', 'O')
                         HAVING COUNT(*) = 2)
            THEN 'B'
```

```
            ELSE (SELECT DISTINCT R3.role
                    FROM Roles AS R3
                   WHERE R3.person = R1.person
                     AND R3.role IN ('D', 'O'))
            END AS combined_role
FROM Roles AS R1
WHERE R1.role IN ('D', 'O');
```

ほかに、もっと優れた解を思いつくかな？

その3

答え　私はここまで、皆さんをわざと自己結合を使う方向へ誘導してきた。だが本当は、UNIONを利用すれば自己結合は必要ない。2役を兼務する人物は2度現れる。つまり、Rolesテーブルに2行登録されている人物の行を探すだけでよいのだ。

```
SELECT R1.person, MAX(R1.role)
  FROM Roles AS R1
 WHERE R1.role IN ('D','O')
 GROUP BY R1.person
HAVING COUNT(*) = 1
UNION
SELECT R2.person, 'B'
  FROM Roles AS R2
 WHERE R2.role IN ('D','O')
 GROUP BY R2.person
HAVING COUNT(*) = 2;
```

SQL-92では、UNIONをビューの中に入れるのは何の問題もないが、古いRDBMSでは通らないないかもしれない。

その4

答え　SQL-92ならCASE式が使える。CASE式は、置換をしたいときに有用なことが多い。多分、これが一番簡単な解答だろう。

```
SELECT person,
       CASE WHEN COUNT(*) = 1
            THEN role
            ELSE 'B' END
  FROM Roles
 WHERE role IN ('D','O')
 GROUP BY person;
```

「THEN role」句は正しく動く。なぜなら、COUNT(*)が1のときには、その社員の役職が1つであることが分かっているからだ。しかし、roleはGROUP BY句には含まれていないため、中にはTHEN MAX(role)と書く必要のあるRDBMS製品があるかもしれない。そうした製品では、ただroleとだけ書いた場合、SELECT句とGROUP BY句との間の不一致による構文エラーと見なされるだろう。

その5

答え

CASE式とGROUP BYを使った“トリック”はまだある。

```
SELECT person,
       CASE WHEN MIN(role) <> MAX(role)
            THEN 'B'
            ELSE MIN(role) END AS combined_role
  FROM Roles
 WHERE role IN ('D','O')
 GROUP BY person;
```

その6

答え

マーク・ウィータラは、全く異なるアプローチでこの問題を解いた。これは提案された時点で最速の解だった。

```
SELECT person,
       SUBSTRING ('DOB' FROM SUM (POSITION (role IN 'DO')) FOR 1)
  FROM Roles
 WHERE role IN ('D','O')
 GROUP BY person;
```

このSQL文はちょっと分かりにくいし、関数呼び出しをネストしているのが紛らわしい。社員名によって作られる1つ1つのグループについて、POSITION関数が'D'には1を、'O'には2を返す。その結果をSUM関数で合計して、SUBSTRING関数で再び1を'D'に、2を'O'に、そして3を'B'に変換している[訳注2]。

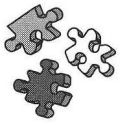
この方法は 共役性（conjugacy）の興味深い応用である。共役性とは、問題を簡単にするために変換とその逆変換を行うときの用語で、一番有名な例は指数関数と対数関数である。

訳注2：SUM関数で合計して3になる人物とは、要するに1（D）＋2（O）の計算が行われた人物、ということである。

パズル 37

移動平均

行同士を比較するSELECT文



あなたは、15分ごとに処理負荷に関する統計情報を収集しているとする。すると顧客がやってきて、時間ごとの統計情報を知りたいと言ってきた――「ある時刻の」ではなく「時間単位の」である。つまり、知りたいのは、1時00分の負荷量、2時00分の負荷量、3時00分の負荷量……ではない。最初の1時間を例にとるなら、(0時00分, 0時15分, 0時30分, 0時45分) の平均負荷量が知りたいのである。そして次には、(0時15分, 0時30分, 0時45分, 1時00分) の平均負荷量が知りたい。後は同様である。こういう平均を移動平均 (moving average) と言う。サンプルテーブルには次のものを使おう。

```
CREATE TABLE Samples
(sample_time TIMESTAMP NOT NULL PRIMARY KEY,
 load REAL NOT NULL);
```

Samples：サンプルテーブル	sample_time：測定時間	load：負荷量

その1

答え

ともあれ、まずはテーブルに移動平均 (moving_avg) を示す列を追加する。

```
CREATE TABLE Samples
(sample_time TIMESTAMP NOT NULL PRIMARY KEY,
 moving_avg REAL DEFAULT 0 NOT NULL ,
 load REAL DEFAULT 0 NOT NULL);
```

そして、テーブルを次のようなUPDATE文で更新する。

```
UPDATE Samples
SET moving_avg
    = (SELECT AVG(S1.load)
         FROM Samples AS S1
        WHERE S1.sample_time
              IN (Samples.sample_time,
                  (Samples.sample_time - INTERVAL '15' MINUTE),
                  (Samples.sample_time - INTERVAL '30' MINUTE),
                  (Samples.sample_time - INTERVAL '45' MINUTE)));
```

その2

答え　しかし、UPDATEの方法はこの1通りだけではない。というのも、おそらく15分ごとに正確に負荷のサンプリングが行われるという仮定は成立しないからだ。実際にはサンプリングエラーが起きることもあるだろう。その場合、タイムスタンプは何分かずれることになる。次のUPDATE文[訳注1]は、正確な一致条件の代わりに、ある程度の時間幅を許容するよう考慮したものである。

```
UPDATE Samples
   SET moving_avg
       = (SELECT AVG(S1.load)
            FROM Samples AS S1
           WHERE S1.sample_time
                 BETWEEN (Samples.sample_time - INTERVAL '1' HOUR)
                     AND Samples.sample_time);
```

その3

答え　答えその2のUPDATE文の条件を利用すれば、移動平均を求めるクエリ[訳注2]を作ることもできる。

```
SELECT S1.sample_time, AVG(S2.load) AS avg_prev_hour_load
  FROM Samples AS S1, Samples AS S2
 WHERE S2.sample_time BETWEEN (S1.sample_time - INTERVAL '1' HOUR)
                          AND S1.sample_time
 GROUP BY S1.sample_time;
```

さて、列を追加する方法とクエリで求める方法の2つを紹介したが、どちらがより適切だろうか。純粋に技術的な観点から言えば、列の追加はテーブルを非正規化することになるので、クエリのほうが好ましいだろう。もっとも、記録されている過去のデータに今後変更が発生せず、かつ移動平均の計算コストが高い場合は、列の追加を考慮して

訳注1：条件にBETWEEN演算子を使っているため、4レコードではなく5レコードを対象に移動平均を求めることなっている。4レコードにするならBETWEENに代えて、不等号で条件を記述する必要がある。

訳注2：このSELECT文も答えその2のUPDATE文と同様、移動平均を求める対象を4レコードにするなら、BETWEENに代えて不等号で条件を記述する必要がある。

もよいかもしれない。

その4

答え 移動平均は、SQL-99のOLAP関数を使って求めることもできる。計測したい時間帯のデータをテーブルに入れれば、次のクエリでそれが可能だ。

```
SELECT *
  FROM (SELECT sample_time,
        AVG(load) OVER (ORDER BY sample_time ASC ROWS 4 PRECEDING)
          FROM Samples)
 WHERE EXTRACT(MINUTE FROM sample_time) = 00;
```

このSELECT文は、先行する時間帯から現在までの累積を計算する。そして、求める観測点（分が00の時刻）だけを表示するために、WHERE句で全体の4分の3の行を除外している。

さらにもう1つ別の“トリック”として、15分刻みで24時間の周期を保持するClock Ticksテーブルを作る方法が挙げられる。そのテーブルから、毎日自身を更新するビューを作ることで、巨大なテーブルを保持する必要がなくなる。

```
CREATE VIEW DailyTimeSlots (slot_timestamp)
AS SELECT CURRENT_DATE + CAST(tick AS MINUTES)
     FROM ClockTicks;
```

パ|ズ|ル|
38 記録の更新
行同士を比較してUPDATEする

今度は会計に関する簡単なパズルである。ここに取引の明細書と、その処理日、処理金額を記録する仕訳テーブルがあるとしよう。1つの明細書からは、複数回の処理が発生する。このテーブルから各処理の間隔日数を割り出し、それを前回の処理が記録されている行に書き込みたい。こうすることで、「その明細書に対する次の処理が何日後に発生するか」を効率的に知ることができる。

テーブルは、次のようなごくシンプルなものだ。

```
CREATE TABLE Journal
(acct_nbr INTEGER NOT NULL,
 trx_date DATE NOT NULL,
 trx_amt DECIMAL(10, 2) NOT NULL,
 duration INTEGER NOT NULL);
```

Journal：仕訳テーブル	acct_nbr：明細書番号	trx_date：処理日
trx_amt：処理金額	duration：前の処理からの日数	

その1

答え

最初の解答は、期間の計算と、カレント行の日付から見て直近の処理日の検索をサブクエリ式で行うものだ。少し考えれば、次のようなUPDATE文ができあがるだろう。

```
UPDATE Journal
   SET duration
     = (SELECT CAST((Journal.trx_date - J1.trx_date) AS INTEGER)
          FROM Journal AS J1
         WHERE J1.acct_nbr = Journal.acct_nbr
           AND J1.trx_date =
                 (SELECT MIN(trx_date)
                    FROM Journal AS J2
                   WHERE J2.acct_nbr = Journal.acct_nbr
                     AND J2.trx_date > Journal.trx_date))
 WHERE EXISTS (SELECT *
                 FROM Journal AS J3
                WHERE J3.acct_nbr = Journal.acct_nbr
                  AND J3.trx_date > Journal.trx_date);
```

各明細書における最も新しい取引を記録している行については、どう扱うかを決めていなかったので、WHERE句の条件により更新対象外としている。

その2

答え

答えその1のUPDATE文を、もう少し詳しく調べてみよう。すると、J1テーブルは何の役にも立っていないことが分かる。ちょっとした“トリック”を使えば、結果に影響を及ぼすことなくこれを取り除くことができる。

```
UPDATE Journal
   SET duration
          = CAST((Journal.trx_date -
                    (SELECT MIN(J1.trx_date)
                       FROM Journal AS J1
                      WHERE J1.acct_nbr = Journal.acct_nbr
                        AND J1.trx_date > Journal.trx_date)
                  ) AS INTEGER)
 WHERE EXISTS (SELECT *
                 FROM Journal AS J2
                WHERE J2.acct_nbr = Journal.acct_nbr
                  AND J2.trx_date > Journal.trx_date);
```

このコードが動くかどうかは、関数の中にスカラサブクエリ式を入れられるかどうかにかかっている。不要なサブクエリを除去することで、I/O回数はSybase Ver.11では50%近くも減少する！ ネストした相関サブクエリは線形ではなく指数関数的に処理量を増やすのだから、この結果は驚くに当たらない。この解答には1つのサブクエリが存在するが、どちらもネストしていない。

ただ、プログラマとしての立場からこのコードの欠点を挙げるなら、同じロジックを2箇所に書かねばならない点だ。これはうまくないし、とりわけ将来コードを変更する際に間違いの原因となる。最初にコーディングしたときにはテキストエディタでカット&ペーストするだろうが、コードを保守するときには往々にしてそれを忘れがちだ。

その3

答え

この難点を回避する1つの方法は、WHERE句を全く使わないことである。次のようにCOALESCE関数を使うことで、最も新しい取引を記録している行を更新対象から外

すことができる。

```
UPDATE Journal
   SET duration
     = COALESCE(CAST((Journal.trx_date -
                  (SELECT MIN(trx_date)
                     FROM Journal AS J1
                    WHERE J1.acct_nbr = Journal.acct_nbr
                      AND J1.trx_date > Journal.trx_date)
                   ) AS INTEGER),
                Journal.duration);
```

また、このUPDATE文はJournalテーブルを1度しかスキャンしないはずだ。だからと言って、このUPDATE文が答えその2のUPDATE文より速いかどうかは定かでない。あなたの使用するRDBMSが、更新された領域をどうやって解放するかに左右されるだろう。

その4

答え

だが、この問題に対する最適解は、ここまでに挙げてきたものではない。先行する行を得るには、新機能であるOLAP関数を使えばよい。その結果をビューに入れれば、次のような解答が得られる。

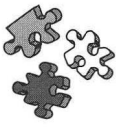
```
SELECT acct_nbr, trx_date,
       (trx_date - MAX(trx_date)
          OVER (PARTITION BY acct_nbr
                ORDER BY trx_date DESC
                ROWS 1 PRECEDING)) AS duration
  FROM Journal;
```

ただし、RDBMSによって持っている時間関数には多少違いがあり、この解を使うときには少しコードを修正する必要があるかもしれないので、注意してほしい。

パズル 39

保険損失

列持ちから行持ちへ



このパズルはマイク・ゴーラから電子メールで送られてきた。オリジナルの問題から少し変えているが、基本的な考え方は同じである。

さて、保険外交員による「顧客が将来被るかもしれない損失」の査定結果を保存するテーブルがあるとしよう。コードを簡単にするために、顧客に降りかかる「危険」はアルファベットのaからoで表すことにする。顧客に降りかかるおそれのない危険については、その危険を表すa～o列にNULLを入れる。反対に危険があれば、その危険度を評価した数値を入れる。例えば、山頂にある花火工場には洪水の危険はないが、爆発事故の危険性は極めて高くなる。なお、a～o列のうち、実際に数値が入るのは概ね5～6列程度である。このテーブルの構成はこんな感じだ。

```
CREATE TABLE Losses
(cust_nbr INTEGER NOT NULL PRIMARY KEY,
 a INTEGER, b INTEGER, c INTEGER, d INTEGER, e INTEGER,
 f INTEGER, g INTEGER, h INTEGER, i INTEGER, j INTEGER,
 k INTEGER, l INTEGER, m INTEGER, n INTEGER, o INTEGER);
```

Losses：(顧客の) 損失テーブル　　cust_nbr：顧客番号　　a～o：顧客に降りかかる危険

試しに、1人の顧客についてデータを入れてみる。

```
INSERT INTO Losses
VALUES (  99,
             5,   10,   15, NULL, NULL,
          NULL, NULL, NULL, NULL, NULL,
          NULL, NULL, NULL, NULL, NULL);
```

さて、2つ目のテーブルを見てみよう。このテーブルは、顧客の潜在的な損失（降りかかる危険）に基づいて販売可能な保険プランを決定するために使われる。

```
CREATE TABLE Policy_Criteria
(criteria_id INTEGER NOT NULL,
 criteria CHAR(1) NOT NULL,
 crit_val INTEGER NOT NULL,
 PRIMARY KEY (criteria_id, criteria, crit_val));
```

Policy_Criteria：プラン基準テーブル	criteria_id：基準ID
criteria：基準	crit_val：基準値

```
INSERT INTO Policy_Criteria VALUES (1, 'A', 5);
INSERT INTO Policy_Criteria VALUES (1, 'A', 9);
INSERT INTO Policy_Criteria VALUES (1, 'A', 14);
INSERT INTO Policy_Criteria VALUES (1, 'B', 4);
INSERT INTO Policy_Criteria VALUES (1, 'B', 10);
INSERT INTO Policy_Criteria VALUES (1, 'B', 20);
INSERT INTO Policy_Criteria VALUES (2, 'B', 10);
INSERT INTO Policy_Criteria VALUES (2, 'B', 19);
INSERT INTO Policy_Criteria VALUES (3, 'A', 5);
INSERT INTO Policy_Criteria VALUES (3, 'B', 10);
INSERT INTO Policy_Criteria VALUES (3, 'B', 30);
INSERT INTO Policy_Criteria VALUES (3, 'C', 3);
INSERT INTO Policy_Criteria VALUES (3, 'C', 15);
INSERT INTO Policy_Criteria VALUES (4, 'A', 5);
INSERT INTO Policy_Criteria VALUES (4, 'B', 21);
INSERT INTO Policy_Criteria VALUES (4, 'B', 22);
```

このデータの意味を言葉にすると、次のとおりになる。なお、「基準A」は「aという危険（損失テーブルのa列）の受容できる危険度」を意味し、基準B以降も同様である。

- プラン1は、基準A＝（5, 9, 14）、基準B＝（4, 10, 20）を持つ
- プラン2は、基準B＝（10, 19）を持つ
- プラン3は、基準A＝5、基準B＝（10, 30）、基準C＝（3, 15）を持つ
- プラン4は、基準A＝5、基準B＝（21, 22）を持つ

99番の顧客に設定された損失データはa＝5、b＝10、c＝15であった。したがって、99番の顧客はプラン1、2、3の保険に加入できるが、基準B＝10の値を持たないプラン4の保険には加入できない。最も優先順位の高いのが、すべての要件を満たすプラン3であり、これが答えになる。次に高いのがプラン1、最後がプラン2である。だが今は、2番目以降の選択肢については考えなくてよい。

その1

答え この問題のトリッキーなところは、損失テーブルでは損失が「列名」として表現されているのに対し、プラン基準テーブルでは「値」として表現されている点である。おかげでデータモデルは台無しだ。問題を解くには、テーブルの形を変換して統一しなければならない。私なら、変換の対象には損失テーブルのほうを選ぶ。以下に示すように、「列持ち」から「行持ち」へ変換するのだ。ビューにすることもできるが、ここでは作業用のテーブルを使う。

```
CREATE TABLE LossDoneRight
(cust_nbr INTEGER NOT NULL,
 criteria CHAR(1) NOT NULL,
 crit_val INTEGER NOT NULL);
```

LossDoneRight：保険損失テーブル（正解版）	cust_nbr：顧客番号
criteria：基準	crit_val：基準値

属性から値への変換は、次のように行う。

```
INSERT INTO LossDoneRight (cust_nbr, criteria, crit_val)
SELECT cust_nbr, 'A', a FROM Losses WHERE a IS NOT NULL
UNION ALL
SELECT cust_nbr, 'B', b FROM Losses WHERE b IS NOT NULL
UNION ALL
SELECT cust_nbr, 'C', c FROM Losses WHERE c IS NOT NULL
UNION ALL
SELECT cust_nbr, 'D', d FROM Losses WHERE d IS NOT NULL
UNION ALL
SELECT cust_nbr, 'E', e FROM Losses WHERE e IS NOT NULL
UNION ALL
SELECT cust_nbr, 'F', f FROM Losses WHERE f IS NOT NULL
UNION ALL
SELECT cust_nbr, 'G', g FROM Losses WHERE g IS NOT NULL
UNION ALL
SELECT cust_nbr, 'H', h FROM Losses WHERE h IS NOT NULL
UNION ALL
SELECT cust_nbr, 'I', i FROM Losses WHERE i IS NOT NULL
UNION ALL
SELECT cust_nbr, 'J', j FROM Losses WHERE j IS NOT NULL
UNION ALL
```

```
SELECT cust_nbr, 'K', k FROM Losses WHERE k IS NOT NULL
UNION ALL
SELECT cust_nbr, 'L', l FROM Losses WHERE l IS NOT NULL
UNION ALL
SELECT cust_nbr, 'M', m FROM Losses WHERE m IS NOT NULL
UNION ALL
SELECT cust_nbr, 'N', n FROM Losses WHERE n IS NOT NULL
UNION ALL
SELECT cust_nbr, 'O', o FROM Losses WHERE o IS NOT NULL;
```

一度このテーブルへ入れてしまえば、後は関係除算を行うだけだ[訳注1]。

```
SELECT L1.cust_nbr, ' could use policy ', C1.criteria_id,
       COUNT(*) AS score
  FROM LossDoneRight AS L1, Policy_Criteria AS C1
 WHERE L1.criteria = C1.criteria
   AND L1.crit_val = C1.crit_val
 GROUP BY L1.cust_nbr, C1.criteria_id
HAVING COUNT(*) = (SELECT COUNT(*)
                     FROM LossDoneRight AS L2
                    WHERE L1.cust_nbr = L2.cust_nbr);
```

このクエリを言葉で表現すれば、「損失と基準を結合し、もし損失がプラン基準テーブルのすべての基準と一致したなら（つまり同数だったなら）、それを保持しておく」となる。これは、2つのテーブルを1対1で対応付けていることを意味する。ただし、片方のテーブルの行は余ってもよいが、もう片方のテーブルの行は使い切らなければならない。

その2

答え さて、この解答を見たゴーラ氏は、「非常にいい線を行っているが、まだ完全ではない」と言ってきた。このクエリは完全に一致するプランを探すものだが、人生いつもいつもそんなうまくいくわけではない。そこで、プラン基準が損失にどの程度うまく適合しているかのランク付けをする代わりに、次のルールを採用したい。

訳注1：関係除算についてはパズル21を参照。

1. プランは、損失テーブルにおいて与えられる基準の部分集合でなければならない
   ―― つまり、プランは余計な基準を含んでいてはならない
2. 損失と一致した基準の値に応じて、プランに得点を付ける

例えば、このルールのもとでは、プラン3は満点の3点、プラン1は2点、プラン2は1点ということになる。他方、プラン4は不適合と見なされる。プラン4にも基準Bが含まれてはいるが、要求された値とは一致しなかったからである。このルールをSQLで表現するのはお安い御用だ。HAVING句を次のように拡張するだけでよい。

```
SELECT L1.cust_nbr, 'matches to ', C1.criteria_id,
       ' with a score of ', COUNT(*) AS score
  FROM LossDoneRight AS L1, Policy_Criteria AS C1
 WHERE L1.criteria = C1.criteria
   AND L1.crit_val = C1.crit_val
 GROUP BY L1.cust_nbr, C1.criteria_id
HAVING COUNT(*) <= (SELECT COUNT(*)
                      FROM LossDoneRight AS L2
                     WHERE L1.cust_nbr = L2.cust_nbr)
   AND COUNT(*) = (SELECT COUNT(DISTINCT C2.criteria)
                     FROM Policy_Criteria AS C2
                    WHERE C1.criteria_id = C2.criteria_id)
 ORDER BY L1.cust_nbr, score;
```

COUNT(*)との最初の比較では、この損失において基準と値の両方について、プランの基準を満たすかどうかを調べている。2番目のCOUNT(*)との比較では、マッチした部分集合がプランの基準と同じだったかどうかを調べている。例えば、プラン4は基準Aを満たすが、基準Bに10番の値を持たないために除外される。

このクエリの実行速度がどうなるかは、私には分からない。だが、かなり厳しいものになる気がする。あるいは、集約とスカラサブクエリ式で使われているcust_nbrとcriteria_idにインデックスを付加したくなるかもしれない[訳注2]。

訳注2：インデックスよりむしろ、LossDoneRightテーブルの(cust_nbr, criteria_id)に主キーを付加したほうがよいだろう。そうすれば、主キーのインデックスを利用することもでき、パフォーマンス上も有利に働く。

パ|ズ|ル|
40 順列
自己結合で順列を作る

2つの集合に含まれる要素の全組み合わせ(x, y)を得る演算を「クロス結合」と呼ぶ。SQLでは、これをとても簡単なクエリで実現できる。例えば次のように書く。

```
SELECT x, y
  FROM BigX CROSS JOIN BigY;
```

しかし、時には組み合わせを垂直方向にソートするのではなく、水平方向にソートしたいと思うこともあるだろう。

ところで、並び順を意識した集合要素の組み合わせを「順列」と言う。例えば{1, 2, 3}という集合があるとすれば、その順列は(1, 2, 3)、(1, 3, 2)、(2, 1, 3)、(2, 3, 1)、(3, 1, 2)、(3, 2, 1)である。n個の要素を組み合わせる順列はその階乗、すなわちn!個存在する。ここで求めたいのは、1から7までの整数の集合から、1行につき1つの順列を作るようなクエリである(合計5040行作られる)。できれば、もっと多くの整数に対しても簡単に適用できるよう、一般化されたクエリにしてほしい。

サンプルテーブルには次のものを使う。

```
CREATE TABLE Elements
(i INTEGER NOT NULL PRIMARY KEY);
```

Elements:集合の要素テーブル　　i:要素

```
INSERT INTO Elements
VALUES (1), (2), (3), (4), (5), (6), (7);
```

その1

答え

まず、明快ではあるが見るだに恐ろしい解答から挙げようか。

```
SELECT E1.i, E2.i, E3.i, E4.i, E5.i, E6.i, E7.i
  FROM Elements AS E1, Elements AS E2, Elements AS E3,
       Elements AS E4, Elements AS E5, Elements AS E6,
       Elements AS E7
```

```
WHERE E1.i NOT IN (E2.i, E3.i, E4.i, E5.i, E6.i, E7.i)
  AND E2.i NOT IN (E1.i, E3.i, E4.i, E5.i, E6.i, E7.i)
  AND E3.i NOT IN (E1.i, E2.i, E4.i, E5.i, E6.i, E7.i)
  AND E4.i NOT IN (E1.i, E2.i, E3.i, E5.i, E6.i, E7.i)
  AND E5.i NOT IN (E1.i, E2.i, E3.i, E4.i, E6.i, E7.i)
  AND E6.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i, E7.i)
  AND E7.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i, E6.i);
```

この巨大な条件部は、ある行内のすべての列値が一意であることを保証するものだ。しかし、パフォーマンスはかなり悪い。

その2

答え

実は、WHERE句に条件を1つ追加するだけでパフォーマンスは改善される。

```
SELECT E1.i, E2.i, E3.i, E4.i, E5.i, E6.i, E7.i
  FROM Elements AS E1, Elements AS E2, Elements AS E3,
       Elements AS E4, Elements AS E5, Elements AS E6,
       Elements AS E7
 WHERE (E1.i + E2.i + E3.i + E4.i + E5.i + E6.i + E7.i) = 28
   AND E1.i NOT IN (E2.i, E3.i, E4.i, E5.i, E6.i, E7.i)
   AND E2.i NOT IN (E1.i, E3.i, E4.i, E5.i, E6.i, E7.i)
   AND E3.i NOT IN (E1.i, E2.i, E4.i, E5.i, E6.i, E7.i)
   AND E4.i NOT IN (E1.i, E2.i, E3.i, E5.i, E6.i, E7.i)
   AND E5.i NOT IN (E1.i, E2.i, E3.i, E4.i, E6.i, E7.i)
   AND E6.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i, E7.i)
   AND E7.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i, E6.i);
```

なぜこれでパフォーマンスが良くなるのかというと、大抵のオプティマイザはANDで結ばれたIN述語の条件より先に「式 = 定数」という形の条件を見て、それを実行するからだ。列値の合計が28になるすべての行が1～7の順列というわけではないが、1～7の順列はすべて合計が28になる（つまり十分条件ではなく、必要条件である）。階乗計算を効率化するには、あらゆる手段を使って事前に行を絞っておかねばならない！

その3

答え

この合計値を使う“トリック”を、もう一歩進めてみよう。まずは、Elementsテーブルを、各要素に重み付けを持たせるように変更する。

```
CREATE TABLE Elements
(i INTEGER NOT NULL,
 wgt INTEGER NOT NULL);
```

Elements：集合の要素テーブル　　i：要素　　wgt：その要素の重み

```
INSERT INTO Elements
VALUES (1, 1), (2, 2), (3, 4), (4, 8), (5, 16), (6, 32), (7, 64);
```

この重みは2の累乗である。これを利用することで、ビットのベクトルをSQL文の中で書くことができる。すると、WHERE句は次のようになる。

```
SELECT E1.i, E2.i, E3.i, E4.i, E5.i, E6.i, E7.i
  FROM Elements AS E1, Elements AS E2, Elements AS E3,
       Elements AS E4, Elements AS E5, Elements AS E6,
       Elements AS E7
 WHERE (E1.wgt + E2.wgt + E3.wgt + E4.wgt + E5.wgt +
        E6.wgt + E7.wgt) = 127;
```

この条件だけで完全に行をフィルタリングできるので、あのたくさんのIN述語は不要になる。この解答にはもう1つ利点がある。それは、要素はもう整数である必要はなく、任意のデータ型でかまわないことだ。

答え

その4

イアン・ヤングは、ここまでに示した解答をSQL Server (7.0および2000) 上でいじって遊んでいるうちに、SQL Serverでは期待したように動かないことが分かった。

というのも、まず答えその1の場合、SQL Serverのオプティマイザは「各条件を別々に見て、結合のたびに関連する部分を適用する」という動作をする。その結果、i番目の結合に対し、結果は7!÷(7−i)!個の項目を持つことになる。

答えその2で追加した全体条件も、そのアプローチを変えることはなかった。それどころか、パフォーマンスは少し (10～15%程度) 悪くなってしまった。

ビットのベクトルを使う答えその3も、条件をいっさい局所化できなかった。この条件は結局最後のクロス結合を制限できただけで、n個の項目に対して$n^{n-1} \times n$項目が結合されることになる。その結果、7項目の場合は最初の単純な解答のほうが5倍から10

倍速く、ビットのベクトルを使う方法は、9項目の場合ではまず使いものにならない。

だが、この素朴な方法にも、まだ改善の余地はある。

手始めに、現状では同じ述語を2箇所でテストしているが、条件を上向き（または下向き）の“三角形状”にすることで、冗長なテストを省くことができる[訳注1]。だが、これもSQL Serverでは大して意味のある改善にならない。むしろ、7個の項目を生成するためにクロス結合を7回行っている点を重視すべきだ。7回のうち、最後の1回の結合はほかの結合によって一意に限定されている。そこで、この最後の結合を削除して、値は答えその2の全体条件と同じ方法で求めることにする。

```
SELECT E1.i, E2.i, E3.i, E4.i, E5.i, E6.i,
       (28 - E1.i - E2.i - E3.i - E4.i - E5.i - E6.i) AS i
  FROM Elements AS E1, Elements AS E2, Elements AS E3,
       Elements AS E4, Elements AS E5, Elements AS E6
 WHERE E2.i NOT IN (E1.i)
   AND E3.i NOT IN (E1.i, E2.i)
   AND E4.i NOT IN (E1.i, E2.i, E3.i)
   AND E5.i NOT IN (E1.i, E2.i, E3.i, E4.i)
   AND E6.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i);
```

もう1つの可能性は、n!回のループを試みることである。理論上、結合においては最小値が必要になる。これは、文字列を文字のリストと見なせば可能だ。再帰関数を展開したクエリをイメージしてほしい。この解では、文字列cが選択された文字を保持し、それを文字列aの中に詰めている。

```
SELECT a || c
  FROM (SELECT a || SUBSTRING(c FROM i FOR 1),
               STUFF(c, i, 1, '')
          FROM Elements,
           (SELECT a || SUBSTRING(c FROM i FOR 1),
                   STUFF(c, i, 1, '')
              FROM Elements,
               (SELECT a || SUBSTRING(c FROM i FOR 1),
                       STUFF(c, i, 1, '')
                  FROM Elements,
                   (SELECT a || SUBSTRING(c FROM i FOR 1),
                           STUFF(c, i, 1, '')
```

訳注1：ちょうど、順列について学校で習うとき必ず出てくる「袋の中から数字の書いてある玉を取り出す」イメージを持つと理解しやすい。最初に取り出すE1.iにはどんな玉を選んでもよいが、次にE2.iを取り出すときには袋の中にE1.iはないので、それ以外の玉が候補になる。E3.i以降も同様である。

```
                    FROM Elements,
                     (SELECT a || SUBSTRING(c FROM i FOR 1),
                             STUFF(c, i, 1, '')
                        FROM Elements,
                         (SELECT SUBSTRING('1234567' FROM i
                                           FOR 1),
                                 STUFF('1234567', i, 1, '')
                            FROM Elements
                           WHERE i <= 7) AS T1 (a,c)
                       WHERE i <= 6) AS T2 (a,c)
                  WHERE i <= 5) AS T3 (a,c)
             WHERE i <= 4) AS T4 (a,c)
        WHERE i <= 3) AS T5 (a,c)
   WHERE i <= 2) AS T6 (a,c);
```

STUFF関数は引数の文字列を分解し、別の文字列を指定された位置に挿入する。これは実装依存の関数だが、多くのRDBMSが持っているし、持っていない環境でもユーザ定義関数で簡単に書くことができる。

ただし、この方法にも問題はある。答えその3までは自己結合のために膨大なループが発生したが、この解でも文字列操作のためにそれと同じくらい多量の処理が必要なのだ。

その5

答え クエリの改善はこれで終わりではない。先ほどのSELECT文の実行計画を見てみると、文字列aの中に積み上がった文字列を解きほぐそうとして、次のような演算上は同値だが巨大なクエリを生成しているようだった。

```
SELECT
  SUBSTRING('1234567',
    a, 1) ||
  SUBSTRING(STUFF('1234567',
    a, 1, ''), b, 1) ||
  SUBSTRING(STUFF(STUFF('1234567',
    a, 1, ''), b, 1, ''), c, 1) ||
  SUBSTRING(STUFF(STUFF(STUFF('1234567',
    a, 1, ''), b, 1, ''), c, 1, ''), d, 1) ||
  SUBSTRING(STUFF(STUFF(STUFF(STUFF('1234567',
    a, 1, ''), b, 1, ''), c, 1, ''), d, 1, ''), e, 1) ||
```

```
SUBSTRING(STUFF(STUFF(STUFF(STUFF(STUFF('1234567',
  a, 1, ''), b, 1, ''), c, 1, ''), d, 1, ''), e, 1,''),f, 1) ||
STUFF(STUFF(STUFF(STUFF(STUFF(STUFF('1234567',
  a, 1, ''), b, 1, ''), c, 1, ''), d, 1, ''), e, 1, ''), f, 1,
  '')
FROM (SELECT i FROM Elements
       WHERE i <= 7) AS T1 (a),
     (SELECT i FROM Elements
       WHERE i <= 6) AS T2 (b),
     (SELECT i FROM Elements
       WHERE i <= 5) AS T3 (c),
     (SELECT i FROM Elements
       WHERE i <= 4) AS T4 (d),
     (SELECT i FROM Elements
       WHERE i <= 3) AS T5 (e),
     (SELECT i FROM Elements
       WHERE i <= 2) AS T6 (f);
```

もし、この種の問題に興味があるなら、ロバート・シジウィックによるアルゴリズムについての調査（http://www.princeton.edu/~rblee/ELE572Papers/p137-sedgewick.pdf）を参照してほしい。そこで紹介されているアルゴリズムは、ループや再帰を使った手続き的なものだが、おそらく再帰的な共通表式に翻訳できるだろう。

パ|ズ|ル|
41

予算

複雑な外部結合（その2）

米国フロリダ州マイアミにあるランソフト社のマーク・フロンテラは、1995年9月にこの問題を送ってきた。

彼は、3つのテーブルを使って予算情報を作っていた。その3つのテーブルとは、「購入する商品（Items）」「購入する商品の見積もり（Estimates）」「実際の購入費（Actuals）」である。テーブル構造は簡単なので、定義文は省略していきなりデータを示そう。

注意してほしいのは、1枚の請求書に複数の商品が記載されることもあれば、1つの商品の代金請求が複数の請求書に分割されていることもある、という点だ。

```
Items

item_nbr   item_descr
=====================
      10   'Item 10'
      20   'Item 20'
      30   'Item 30'
      40   'Item 40'
      50   'item 50'
```

Items：商品テーブル　　item_nbr：商品番号　　item_descr：商品の説明

```
Estimates

item_nbr   estimated_amt
========================
      10          300.00
      10           50.00
      20          325.00
      20          110.00
      40           25.00
```

Estimates：購入予算テーブル　　item_nbr：商品番号　　estimated_amt：見積もり金額

```
Actuals

item_nbr    actual_amt    check_nbr
================================
      10        300.00    '1111'
      20        325.00    '2222'
      20        100.00    '3333'
      30        525.00    '1111'
```

Actuals：購入費用テーブル　item_nbr：商品番号　actual_amt：購入金額
check_nbr：請求書番号

さて、これら3つのテーブルから次のようなレポートを作成したい。

```
item_nbr    item_descr    actual_tot    estimate_tot    check_nbr
=============================================================
      10    'item 10'         300.00          350.00    '1111'
      20    'item 20'         425.00          435.00    'Mixed'
      30    'item 30'         525.00            NULL    '1111'
      40    'item 40'           NULL            25.00    NULL
```

50番の商品はActualsテーブルにもEstimatesテーブルにもレコードがないので、レポートには表示されない。actual_tot列はある商品の購入で実際に支払った代金の合計、estimate_tot列はその商品の見積もり金額の合計を示している。

その1

答え

そもそも、このスキーマ自体を多少直したほうがよいとは思うが、スカラサブクエリを使ってややトリッキーなコードを書いてやれば、1つのクエリで欲しいアウトプットを得られる。

```
SELECT * FROM
  (SELECT I1.item_nbr, I1.item_descr,
          (SELECT SUM(A1.actual_amt)
             FROM Actuals AS A1
            WHERE I1.item_nbr = A1.item_nbr) AS tot_act,
          (SELECT SUM(E1.estimated_amt)
             FROM Estimates AS E1
            WHERE I1.item_nbr = E1.item_nbr) AS estimate_tot,
```

```
        (SELECT CASE WHEN COUNT(*) = 1
                     THEN MAX(check_nbr)
                     ELSE 'Mixed' END
           FROM Actuals AS A2
          WHERE I1.item_nbr = A2.item_nbr
          GROUP BY item_nbr) AS check_nbr
   FROM Items I1) AS TMP
  WHERE tot_act IS NOT NULL
     OR estimate_tot IS NOT NULL;
```

ここでの"トリック"は、3つのスカラサブクエリにある。最初の2つのスカラサブクエリは、まるでGROUP BYと外部結合を使っているかのように、実際に支払った代金の合計と見積もり金額の合計を計算する。

3番目のスカラサブクエリはもっとトリッキーだ。これは、結果テーブルに含まれる商品と結びつくActualsテーブルの行をすべて見つけ出し、それらの行から商品ごとのグループを作る。もしグループが空なら（つまり、その商品に対する請求書が1枚も発行されていなければ）、サブクエリは1つのNULLを返すので、そのNULLを表示すればよい。もしグループに請求書が1枚だけであれば、CASE式はその請求書番号を返す。MAX関数は、サブクエリが常に単一のスカラ値を返すことを保証するための安全策だ。SQL-92に準拠するRDBMSなら、MAX関数は不要かもしれない。最後に、請求書が2枚以上存在した場合は、COUNT(*)が1より大きくなり、伝票番号を表す文字列の代わりに'Mixed'が返ることになる。

その2

答え 答えその1のクエリをLEFT OUTER JOIN（左外部結合）を使って書き換えると、次のようになる。

```
SELECT I1.item_nbr, I1.item_descr,
       SUM(A1.actual_amt) AS tot_act,
       SUM(E1.estimated_amt) AS estimate_tot,
       (SELECT CASE WHEN COUNT(check_nbr) = 0
                    THEN NULL
                    WHEN COUNT(check_nbr) = 1
                    THEN MAX(check_nbr)
                    ELSE 'Mixed' END
          FROM Actuals A2
         WHERE A2.item_nbr = I1.item_nbr) AS check_nbr
```

```
 FROM (Items AS I1
       LEFT OUTER JOIN
      (SELECT item_nbr,
              SUM(actual_amt) AS actual_amt
         FROM Actuals
        GROUP BY item_nbr) AS A1
      ON I1.item_nbr = A1.item_nbr)
         LEFT OUTER JOIN
         (SELECT item_nbr,
                 SUM(estimated_amt) AS estimated_amt
            FROM Estimates
           GROUP BY item_nbr) AS E1
         ON I1.item_nbr = E1.item_nbr
WHERE actual_amt IS NOT NULL
   OR estimated_amt IS NOT NULL
GROUP BY I1.item_nbr, I1.item_descr;
```

パズル 42

魚のサンプリング調査

存在しないデータの集計

魚釣りへ行こう！ 私たちが釣り場へ到着すると、そこの管理人が釣り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い、魚に関するサンプル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_tally INTEGER NOT NULL,
 PRIMARY KEY (sample_id, fish_name));
```

Samples：サンプルテーブル　　sample_id：サンプルID　　fish_name：魚名
found_tally：発見総数

```
INSERT INTO Samples
VALUES (1, 'minnow', 18),
       (1, 'pike', 7),
       (2, 'pike', 4),
       (2, 'carp', 3),
       (3, 'carp', 9),
        ... ;
```

'minnow'：小魚　　'pike'：カワカマス　　'carp'：コイ

```
CREATE TABLE SampleGroups
(group_id INTEGER NOT NULL,
 group_descr CHAR(20) 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
group_descr：グループの説明（生息域など）　　sample_id：サンプルID

```
INSERT INTO SampleGroups
VALUES (1, 'muddy water', 1),
       (1, 'muddy water', 2),
       (2, 'fresh water', 1),
       (2, 'fresh water', 2),
       (2, 'fresh water', 3),
       ...;
```

'muddy water'：泥水　　'fresh water'：清水

1つのサンプルは、複数のグループに分類され得ることに注意してほしい。例えば、サンプル1には、muddy waterにすむ（グループ1に分類される）魚と、fresh waterにすむ（グループ2に分類される）魚が含まれている。

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いた。例えば、グループ1（muddy water）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_tally)
  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？

その1

答え 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);
```

その2

答え 実は困ったことに、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文で平均を求めるには、いくつかの“トリック”を使う必要がある。

```
SELECT fish_name, SUM(found_tally) /
            (SELECT COUNT(sample_id)
               FROM SampleGroups
              WHERE group_id = :my_group) AS X
  FROM Samples SA
       INNER JOIN
       SampleGroups SG
       ON SA.sample_id = SG.sample_id
 WHERE fish_name = :my_fish_name
   AND group_id = :my_group
 GROUP BY fish_name;
```

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。とはいえ、このSQL文は少々トリッキーだ。

まず、被除数を求めるSUM関数は、空集合を引数にとった場合にNULLを返す。そ

の場合、この分数（商）はNULLになる。一方、除数を導くサブクエリ式も結果が空集合の場合にはNULLを返すのだが、サブクエリの中のCOUNT(<式>)は、引数が空集合だった場合にはゼロを返す。

COUNT(<式>)がNULLを返すケースはただ1つ、NULLしか含まないテーブルに適用した場合である。だがこの問題では、Samples、SamplesGroupsテーブルともにNULLを排除するよう宣言してあるので、この危険をすでに回避できている。

その3

答え

米国カンサス市のアニルバブ・ジャイスワールは、Oracle版の少し違った解答を送ってきた。それをSQL-92へ変換したコードを示そう。

```
SELECT COALESCE(fish_name, :my_fish_name),
       AVG(COALESCE(found_tally, 0))
  FROM Samples AS SA
       RIGHT OUTER JOIN
       SampleGroups AS SG
       ON SA.sample_id = SG.sample_id
       AND SA.fish_name = :my_fish_name
 WHERE SG.group_id = :my_group
 GROUP BY COALESCE(fish_name, :my_fish_name);
```

COALESCE関数は、引数のリストを走査して、最初のNULLでない値を返す。これによって、AVG関数の引数はNULLからゼロへ変換される。AVG関数が単一の列ではなく式を引数として取れることに、ちょっと違和感を感じるかもしれない。また、この解答のもう1つのうまいトリックは、1つではなく2つの列（sample_id、fish_name）に対して右外部結合をしていることだ。主キーが常に1列とは限らないことを考えれば、これは非常に優れた方法と言える。

実は、答えその1とその2、その3が異なる結果を返すことがある。SampleGroupsテーブルにSamplesテーブルにないsample_idが登録されていた場合、答えその1はそのsample_idを分母にカウントしないが、答えその2とその3はカウントする。

もっとも、このような「調査対象外」のIDが含まれることは、本来あり得ないことである。したがって、SampleGroupsテーブルに外部キーを付してあらかじめ不当なデータを除外しておくのもよい手だろう。

パ|ズ|ル|
43 卒業
CASE式の高度な応用

このパズルは、リチャード・レムレーがもっと複雑な問題のロジックをもとに作ったものである。ANSI/ISO SQL-92規格の範疇(はんちゅう)でSQL文を書くときには、以前とは全く違う考え方が必要なことを示す格好の例題になっている。SQL-92の新機能を使うことで、これまで書けなかったきれいな解法がいくつも生まれた。同じことは、SQL-99規格の機能についても言える。このパズルでは、導出テーブルとCASE式、および等結合以外の条件を使った外部結合を利用する——たった1つのクエリの中でだ。

それではパズルを説明しよう。まず、学生名を表すstudent_names列がある。学生たちは、単位を取得するための講義を受講しており、講義はいずれかの科目に属している。Categories（科目）テーブルには、科目を表すcredit_cat列と、その科目で卒業に必要な単位数を表すrqd_credits列がある。CreditsEarned（取得済み単位）テーブルの各行には、学生がどの科目の単位をどれだけ取得したかが格納される（より論理的に設計するなら、このテーブルには学生名、講義、取得単位数を持たせ、科目〔credit_cat〕は別途Coursesテーブルを参照するようにするべきだが、ここでは定義を少し簡単にしておく）。

なお、学生が卒業するためには、Categoriesテーブルにある全科目で必要単位数を取得しなければならない。

```
CREATE TABLE Categories
(credit_cat CHAR(1) NOT NULL,
 rqd_credits INTEGER NOT NULL);
```

Categories：科目テーブル　　credit_cat：科目
rqd_credits：卒業に必要なその科目の単位数

```
CREATE TABLE CreditsEarned  --主キーなし
(student_name CHAR(10) NOT NULL,
 credit_cat CHAR(1) NOT NULL,
 credits INTEGER NOT NULL);
```

CreditsEarned：取得済み単位テーブル　　student_name：学生名
credit_cat：科目　　credits：取得単位数

```
INSERT INTO Categories
VALUES ('A', 10),
       ('B', 3),
       ('C', 5);
```

```
INSERT INTO CreditsEarned
VALUES ('Joe', 'A', 3), ('Joe', 'A', 2), ('Joe', 'A', 3),
       ('Joe', 'A', 3), ('Joe', 'B', 3), ('Joe', 'C', 3),
       ('Joe', 'C', 2), ('Joe', 'C', 3),
       ('Bob', 'A', 2), ('Bob', 'C', 2), ('Bob', 'A', 12),
       ('Bob', 'C', 4),
       ('John', 'A', 1), ('John', 'B', 1),
       ('Mary', 'A', 1), ('Mary', 'A', 1), ('Mary', 'A', 1),
       ('Mary', 'A', 1), ('Mary', 'A', 1), ('Mary', 'A', 1),
       ('Mary', 'A', 1), ('Mary', 'A', 1), ('Mary', 'A', 1),
       ('Mary', 'A', 1), ('Mary', 'A', 1), ('Mary', 'B', 1),
       ('Mary', 'B', 1), ('Mary', 'B', 1), ('Mary', 'B', 1),
       ('Mary', 'B', 1), ('Mary', 'B', 1), ('Mary', 'B', 1),
       ('Mary', 'C', 1), ('Mary', 'C', 1), ('Mary', 'C', 1),
       ('Mary', 'C', 1), ('Mary', 'C', 1), ('Mary', 'C', 1),
       ('Mary', 'C', 1), ('Mary', 'C', 1), ('Mary', 'C', 1);
```

さて、まず初めに卒業資格のある学生――つまり、すべての科目で卒業に必要な数の単位を取得した学生をリストを作成してほしい。それから、卒業資格のない学生のリストもお願いしたい。ただ、リストを見る側としては、これら2つのリストが1つにまとめられ、全学生の卒業可／不可を一覧できるのが一番ありがたい。そこで、各行に学生名と、卒業可もしくは卒業不可の列に'X'印を記した次のようなリストの作成を、ここでの出題とする。

```
EligibleReport

student_name   grad   nograd
============================
'Bob'                 'X'
'Joe'          'X'
'John'                'X'
'Mary'         'X'
```

EligibleReport：卒業資格レポート	student_name：学生名
grad：卒業可	nograd：卒業不可

その1

答え 私が思いついた最適解は、次のようなものだ。

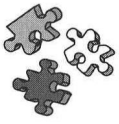
```
SELECT X.student_name,
       CASE WHEN COUNT(C1.credit_cat)
                 >= (SELECT COUNT(*) FROM Categories)
            THEN 'X'
            ELSE ' ' END AS grad,
       CASE WHEN COUNT(C1.credit_cat)
                 < (SELECT COUNT(*) FROM Categories)
            THEN 'X'
            ELSE ' ' END AS nograd
  FROM (SELECT student_name, credit_cat,
               SUM(credits) AS cat_credits
          FROM CreditsEarned
         GROUP BY student_name, credit_cat) AS X
       LEFT OUTER JOIN
       Categories AS C1
         ON X.credit_cat = C1.credit_cat
        AND X.cat_credits >= C1.rqd_credits
 GROUP BY X.student_name;
```

導出テーブルXは、学生名と科目、それにその学生が取得した科目ごとの合計単位数(cat_credits)を含んでいる。この解法のカギは次のステップにある——そう、credit_cat列同士を等結合し、cat_credits >= rqd_creditsという条件を使う左外部結合である。それから学生名(student_name)でグループ化すれば、各学生が卒業に必要な単位数を取得できた科目の数をCOUNT(C1.credit_cat)で求められる。その数と科目の総数を比較すれば、学生に卒業資格があるか否かが判明するというわけだ。

学生がどちらに振り分けられるかは、'X'マークで示している。この方法なら、学生に講義を1つも受けなかった科目がある場合も自動的に扱える。なぜなら、COUNT(C1.credit_cat)は、少なくとも1単位を取得した科目しか数えないからだ[訳注1]。

訳注1：COUNT(*)と違って、COUNT(<列名>)は集計の前にNULLを排除するため。同じトリックを使う類題として、パズル20も参照。

パ|ズ|ル|
44 商品のペア
順列から組み合わせに変換する



アボット・デ・ラームは、1996年11月にこの問題をAccessフォーラムに投稿してきた。彼の手元には、売上伝票から集めたデータがあった。各伝票には、売れた商品が売れた順序でペアになって印字されていた。データを格納しているSalesSlipsテーブルは、次のような形をしている。

```
CREATE TABLE SalesSlips
(item_a INTEGER NOT NULL,
 item_b INTEGER NOT NULL,
 PRIMARY KEY(item_a, item_b),
 pair_tally INTEGER NOT NULL);
```

SalesSlips：売上伝票テーブル　　item_a：商品a
item_b：商品b　　pair_tally：このペアの数

item_aは先に売れた商品、item_bはitem_aの次に売れた商品で、pair_tallyはitem_aとitem_bのペアの合計数である。売れた順序は必ず「item_a→item_b」と決まっているため、売れた順序が逆のペアは別に計算（別の行に記録）されている。

```
SalesSlips

item_a    item_b    pair_tally
============================
 12345     12345            12
 12345     67890             9
 67890     12345             5
```

しかしラームは、いくつかのレポートを作成するために、売れた順序に関係なくペアの数を合計した結果が欲しいと考えていた。要するに「順列」を「組み合わせ」に変換したい、ということである。上のサンプルを使うなら、求める結果は次のようになる。

```
item_a   item_b   pair_tally
============================
 12345    12345           12
 12345    67890           14
```

ラームが試してみたところ、順序だけが違うペア同士を足し上げることは、自己結合で簡単にできた。しかし、重複行を排除できないのが悩みの種だった。

```
SELECT S0.item_a, S0.item_b,
       SUM(S0.pair_tally + S1.pair_tally) AS pair_tally
  FROM SalesSlips AS S0, SalesSlips AS S1
 WHERE S0.item_b = S1.item_a
   AND S0.item_a = S1.item_b
 GROUP BY S0.item_a, S0.item_b, S1.item_a, S1.item_b;
```

このクエリは、次のような間違った結果を返す。

```
item_a   item_b   pair_tally
============================
 12345    12345           24
 12345    67890           14
 67890    12345           14
```

ラームは、順序だけが違うペアを探してその値を合計し、片方を削除するプログラムを書くことも考えた。だが、できればSQLで解決したいというのが彼の要望である。

その1

答え まず、ラームの考えたクエリは簡単な修正でちゃんと動くようになる。

```
SELECT S0.item_a, S0.item_b,
       SUM(S0.pair_tally + S1.pair_tally) AS pair_tally
  FROM SalesSlips AS S0, SalesSlips AS S1
 WHERE S0.item_a <= S0.item_b
   AND S0.item_a = S1.item_b
   AND S0.item_b = S1.item_a
 GROUP BY S0.item_a, S0.item_b, S1.item_a, S1.item_b;
```

しかし、自己結合は高コストだし、本当は使う必要もない。代わりにこう書けばよい。

```
SELECT CASE WHEN item_a <= item_b
            THEN item_a
            ELSE item_b END AS s1,
       CASE WHEN item_a <= item_b
            THEN item_b
            ELSE item_a END AS s2,
       SUM (pair_tally)
  FROM SalesSlips
 GROUP BY s1, s2;
```

正直に言うと、このクエリは動かないかもしれない。列s1とs2はGROUP BYが実行された後に作られるので、GROUP BYのキーに使うことはできないからだ。とはいえこの構文は、実際には多くのRDBMSで通用する。不正ではあるが、SELECT句のリストを最初に作り、それを埋めていくという実行順序がとられているためだ。SQL-92準拠の正しいクエリでは、次のようにテーブルサブクエリを使うことになるだろう。

```
SELECT s1, s2, SUM(pair_tally)
  FROM (SELECT CASE WHEN item_a <= item_b
                    THEN item_a
                    ELSE item_b END,
               CASE WHEN item_a <= item_b
                    THEN item_b
                    ELSE item_a END,
               pair_tally
          FROM SalesSlips) AS Report (s1, s2, pair_tally)
 GROUP BY s1, s2;
```

その2

答え

SQL-89では、テーブルサブクエリ式をビューにして、そのビューを別のクエリの中で使う、という方法を使わなければならなかった。したがって、次に示すクエリも上と基本的に同じなのだが、ステップを2つに分けている。次に示すReportビューは、ほかのレポートを作成する際にも利用できるだろう。

```
CREATE VIEW Report (s1, s2, pair_tally)
AS SELECT CASE WHEN item_a <= item_b
               THEN item_a
```

```
                ELSE item_b END,
           CASE WHEN item_a <= item_b
                THEN item_b
                ELSE item_a END,
           pair_tally
      FROM SalesSlips;

SELECT s1, s2, SUM(pair_tally)
  FROM Report
GROUP BY s1, s2;
```

その3

答え しかし、一番の方法は、クエリを実行する前にデータベース自身を更新し、item_aに2つのコード番号の最小値を持たせることである。これはとても簡単にできる。

```
UPDATE SalesSlips
   SET item_a = item_b,
       item_b = item_a
 WHERE item_a > item_b;
```

同じことを、INSERT時にトリガーを使って行うこともできるが、そうすると実装依存の手続き型のコードを書くことになる。やはり真の解答は、モップで床をふき（＝上のUPDATEを実行し）、その後に次のCHECK制約を付加して“水漏れ箇所を修復する”ことである[訳注1]。

```
CREATE TABLE SalesSlips
(item_a INTEGER NOT NULL,
 item_b INTEGER NOT NULL,
   CHECK (item_a <= item_b)
 pair_tally INTEGER NOT NULL);
```

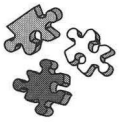
訳注1：ただし、答えその3のUPDATE文は次のようにデータを更新するため、もとのSalesSlipsテーブルから事前に主キーを外しておく必要がある。

item_a	item_b	pair_tally
12345	12345	12
12345	67890	9
12345	67890	5

パズル 45

ペパロニピザ

期間別合計を求める



古典的な会計問題に、過去の請求書から時系列のレポートを作るという良問がある。ここでは「フレンズ・オブ・ペパロニ」という、ピザショップで使えるクレジットカードを発行している会社を題材に取り上げよう。もし、あなたの友人に会員がいたら、その人のカードでピザを購入できるか確認しておくとよかったのだが。

請求データを記録しているFriendsOfPepperoniテーブルには、会員の識別番号（cust_id）、発行日（bill_date）、金額（pizza_amt）が含まれている。だが、これらはどれもキーにはならない。そのため、1人の会員に日付や金額が重複した行が発生し得る。要するに、これは昔ながらのジャーナルファイルを、SQLのテーブルを使って実現しているだけなのだ。

ここで考えるのは、0～30日前、31日～60日前、61日～90日前、そして90日以上前という4つの期間における各会員の合計請求額を求める方法である。「請求金額の時系列レポート」と呼ばれるこのレポートからは、フレンズ・オブ・ペパロニのクレジットカードサービスがどう使われているかが見えてくる。

その1

答え

各期間に対するクエリは、UNION ALL演算子を使って書くことができる。次のクエリを見てほしい。

```
SELECT cust_id, '0-30 days = ' AS age, SUM(pizza_amt)
  FROM FriendsOfPepperoni
 WHERE bill_date BETWEEN (CURRENT_DATE - INTERVAL '30' DAY)
   AND CURRENT_DATE
 GROUP BY cust_id
UNION ALL
SELECT cust_id, '31-60 days = ' AS age, SUM(pizza_amt)
  FROM FriendsOfPepperoni
 WHERE bill_date BETWEEN (CURRENT_DATE - INTERVAL '60' DAY)
   AND (CURRENT_DATE - INTERVAL '31' DAY)
 GROUP BY cust_id
UNION ALL
SELECT cust_id, '61-90 days = ' AS age, SUM(pizza_amt)
  FROM FriendsOfPepperoni
 WHERE bill_date BETWEEN (CURRENT_DATE - INTERVAL '90' DAY)
```

```
    AND (CURRENT_DATE - INTERVAL '61' DAY)
 GROUP BY cust_id
UNION ALL
SELECT cust_id, '90+ days = ' AS age, SUM(pizza_amt)
  FROM FriendsOfPepperoni
 WHERE bill_date < CURRENT_DATE - INTERVAL '90' DAY
 GROUP BY cust_id
 ORDER BY cust_id, age;
```

2番目の列は、期間をテキストで表現している。文字列は時間と同じ順序を持つため、こうすることで各会員のデータを期間でソートするのが簡単になる。ただ、このクエリはうまく動くが、少し時間がかかる。SQL-92ならばもっとよい解法があるに違いない。

その2

答え

CASE式を使えば、UNIONを使う必要はない。UNIONを使うとテーブルを何度も検索することになるが、CASE式ならば検索は1回で済む。

```
SELECT cust_id,
       SUM(CASE WHEN bill_date
                     BETWEEN CURRENT_DATE - INTERVAL '30' DAY
                         AND CURRENT_DATE
                THEN pizza_amt ELSE 0.00 END) AS age1,
       SUM(CASE WHEN bill_date
                     BETWEEN CURRENT_DATE - INTERVAL '60' DAY
                         AND CURRENT_DATE - INTERVAL '31' DAY
                THEN pizza_amt ELSE 0.00 END) AS age2,
       SUM(CASE WHEN bill_date
                     BETWEEN CURRENT_DATE - INTERVAL '90' DAY
                         AND CURRENT_DATE - INTERVAL '61' DAY
                THEN pizza_amt ELSE 0.00 END) AS age3,
       SUM(CASE WHEN bill_date < CURRENT_DATE - INTERVAL '91' DAY
                THEN pizza_amt ELSE 0.00 END) AS age4
  FROM FriendsOfPepperoni
 GROUP BY cust_id
 ORDER BY cust_id;
```

UNIONをCASE式で置き換えるのは、便利な“トリック”である。ただし、答えその1が期間を行持ちで返すのに対し、答えその2は列持ちで返すという違いはある。

その3

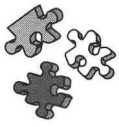
答え レポートに必要な期間を持つ共通表式か導出テーブルを作ると、UNIONもCASE式も不要になる。

```
WITH ReportRanges(day_count, start_cnt, end_cnt)
AS (VALUES ('under Thirty days', 00, 30),
   ('Sixty days', 31, 60),
   ('Ninty days', 61, 90))
SELECT F1.cust_id, COALESCE(R1.day_count, 'Over Ninety days'),
       SUM(pizza_amt)
  FROM FriendsOfPepperoni AS F1
       LEFT OUTER JOIN
       ReportRanges AS R1
       ON F1.bill_date
          BETWEEN CURRENT_DATE - end_cnt
              AND CURRENT_DATE - start_cnt
 GROUP BY F1.cust_id, R1.day_count
 ORDER BY F1.cust_id, R1.day_count;
```

このSQL文のほうがCASE式よりも保守しやすいし、拡張性にも優れている。インデックスを利用すれば、スピードもさらに速くなる。忘れないでもらいたいが、SQLは計算のためではなく結合のために設計された言語なのだ。

パ|ズ|ル|
46 販売促進

期間内での最大値



たった今、あなたはデパートの営業部長に着任したとしよう。データベースには2つのテーブルがある。1つは、その店舗で過去に開かれた販促イベントのカレンダー。もう1つは、その販促期間中の売上高のリストである。

あなたはこれら2つのテーブルから、各期間中に最高の売上をあげた販売員を調べるクエリを書かなければならない。その販売員には臨時ボーナスが支給されるのだ。

```
CREATE TABLE Promotions
(promo_name CHAR(25) NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 CHECK (start_date <= end_date));
```

Promotions：販売促進テーブル	promo_name：販促イベントの名称
start_date：開始日	end_date：終了日

```
Promotions

promo_name                  start_date     end_date
=========================================================
'Feast of St. Fred'         '1995-02-01'   '1995-02-07'
'National Pickle Pageant'   '1995-11-01'   '1995-11-07'
'Christmas Week'            '1995-12-18'   '1995-12-25'
```

```
CREATE TABLE Sales
(ticket_nbr INTEGER NOT NULL PRIMARY KEY,
 clerk_name CHAR (15) NOT NULL,
 sale_date DATE NOT NULL,
 sale_amt DECIMAL (8,2) NOT NULL);
```

Sales：売上テーブル	ticket_nbr：チケット番号	clerk_name：店員名
sale_date：売上日	sale_amt：売上額	

その1

答え

今から示すクエリの“トリック”は、1人1人の販売員が販促期間中にいくら売り上げ

たかを調べ、そのグループの中から最も高い合計額を抜き出す、というものだ。

クエリの第1段階は、比較的簡単な結合とGROUP BYの文で書ける。トリッキーなのは、各グループの中から最大の売上高を見つけ出す第2段階のHAVING句である。まずは答えのSQL文を見てもらいたい。解説はそれからだ。

```
SELECT S1.clerk_name, P1.promo_name,
       SUM(S1.sale_amt) AS sales_tot
  FROM Sales AS S1, Promotions AS P1
 WHERE S1.sale_date BETWEEN P1.start_date AND P1.end_date
 GROUP BY S1.clerk_name, P1.promo_name
HAVING SUM(sale_amt) >= ALL(
         SELECT SUM(sale_amt)
           FROM Sales AS S2
          WHERE S2.clerk_name <> S1.clerk_name
            AND S2.sale_date
                BETWEEN (SELECT start_date
                           FROM Promotions AS P2
                          WHERE P2.promo_name = P1.promo_name)
                    AND (SELECT end_date
                           FROM Promotions AS P3
                          WHERE P3.promo_name = P1.promo_name)
          GROUP BY S2.clerk_name);
```

私たちが求めたいのは、同一期間内での売上がほかのどの販売員があげた売上よりも多いか等しい販売員と、その期間である。「S2.clerk_name <> S1.clerk_name」という述語は、サブクエリの合計からほかの販売員を除外している。BETWEEN述語の中の2つのサブクエリ式は、販促期間について正しい日付を使っていることを保証するものだ。

このクエリを改善しようとしたときに最初に浮かぶ考えは、次のようにBETWEEN述語の中にあったサブクエリ式を1つ外側のクエリで直接参照することだ。

```
SELECT S1.clerk_name, P1.promo_name,
       SUM(S1.sale_amt) AS sales_tot
  FROM Sales AS S1, Promotions AS P1
 WHERE S1.sale_date BETWEEN P1.start_date AND P1.end_date
 GROUP BY S1.clerk_name, P1.promo_name
HAVING SUM(sale_amt) >= ALL (
         SELECT SUM(sale_amt)
           FROM Sales AS S2
          WHERE S2.clerk_name <> S1.clerk_name
            AND S2.sale_date
                BETWEEN P1.start_date
```

```
                AND P1.end_date -- エラー!
          GROUP BY S2.clerk_name);
```

ところが、このコードは動かない——なぜ動かないか、その理由が分かるようなら、このSQL文を本当に理解できている。先に進む前に、各自で理由を考えてみてほしい。

その2

答え

「GROUP BY S1.clerk_name, P1.promo_name」句は、集約されたテーブルを作る。そこには、集約関数と2つの集約キーの列しか含まれない。集約された時点で、FROM句で作られたオリジナルの作業テーブルは消去され、この集約された作業テーブルにとって代わられる。この時点でstart_dateもend_dateもすでに存在しないのだ。

サブクエリ式が正しく動作するには、外側のP1テーブルの存続中にこれを参照しなければならない。なぜなら、クエリは一番内側のサブクエリから外側へ向かって展開されるのであって、集約テーブルから展開されるのではないからだ。

ただし、売上を探す期間がはっきりしている場合なら、P1.start_dateとP1.end_dateに代えてその起点と終点の日付を直接書けば、このクエリは正しく動作する。

さて、私が書いたコラムを読んだ2人の読者が、答えその1の改良版を送ってきてくれた。リチャード・レムレーとJ.D.マクドナルドは、もし期間に重複がなければ、Promotionsテーブルはキー列しか持たないことに気づいた。その場合、GROUP BY句でpromo_name、start_date、end_dateを使っても、グループは一切変化しない。だがそれによって、HAVING句でstart_dateとend_dateが使えるようになるのだ。

```
SELECT S1.clerk_name, P1.promo_name,
       SUM(S1.sale_amt) AS sales_tot
  FROM Sales AS S1, Promotions AS P1
 WHERE S1.sale_date BETWEEN P1.start_date AND P1.end_date
 GROUP BY P1.promo_name, P1.start_date, P1.end_date,
          S1.clerk_name
HAVING SUM(S1.sale_amt) >= ALL (
         SELECT SUM(S2.sale_amt)
           FROM Sales AS S2
          WHERE S2.sale_date
                BETWEEN P1.start_date AND P1.end_date
                    AND S2.clerk_name <> S1.clerk_name
 GROUP BY S2.clerk_name);
```

あるいは、HAVING句のサブクエリの中を少し変えることで、述語の数を減らすことができる。次のように書くのだ。

```
...
HAVING SUM(S1.sale_amt) >= ALL (
         SELECT SUM(S2.sale_amt)
           FROM Sales AS S2
          WHERE S2.sale_date
                BETWEEN P1.start_date
                    AND P1.end_date
          GROUP BY S2.clerk_name);
```

この2パターンのコードの間にパフォーマンス上の大きな差があるかどうかについては、はっきりしたことは言えない。ただし、2番目のほうがきれいではある。

その3

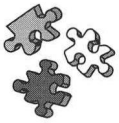
答え

SQL-99の新機能である共通表式を使うと、データを複数のレベルで簡単に集約することができる。

```
WITH ClerksTotals (clerk_name, promo_name, sales_tot)
AS
(SELECT S1.clerk_name, P1.promo_name, SUM(S1.sale_amt)
   FROM Sales AS S1, Promotions AS P1
  WHERE S1.sale_date BETWEEN P1.start_date AND P1.end_date
  GROUP BY S1.clerk_name, P1.promo_name)
SELECT C1.clerk_name, C1.promo_name, C1.sales_tot
  FROM ClerksTotals AS C1
 WHERE C1.sales_tot
       = (SELECT MAX(C2.sales_tot)
            FROM ClerksTotals AS C2
           WHERE C1.promo_name = C2.promo_name);
```

このコードはかなり簡潔だから、メンテナンスも楽になるはずだ。

パ|ズ|ル|
47 座席のブロック
CHECK制約の中でサブクエリを使う



このパズルのオリジナルバージョンは、ジョージア大学のボブ・スターンズから送られてきた。もともとは、インターネットサーバ上のページ割り当てを扱う問題だったが、私が劇場の最前列の席をまとめて予約する問題として書き換えた。

予約は、予約者の名前および予約の開始席番（start_seat）と終了席番（finish_seat）から構成されている。予約に際しては「予約された座席ブロック同士で重複を許さない（座席のダブルブッキングを防ぐ）」という規則がある。予約テーブルは次のような形だ。

```
CREATE TABLE Reservations
(reserver CHAR(10) NOT NULL PRIMARY KEY,
 start_seat INTEGER NOT NULL,
 finish_seat INTEGER NOT NULL);
```

Reservations：予約テーブル　　reserver：予約者　　start_seat：開始席番
finish_seat：終了席番

```
Reservations

reserver   start_seat   finish_seat
===================================
'Eenie'             1             4
'Meanie'            6             7
'Mynie'            10            15
'Melvin'           16            18
```

ここでやりたいことは、重複を許さないという予約規則を破るような挿入が行われないことを保証する制約を、Reservationsテーブルに付加することだ。これはいくつかのステップに分けて考えないと、見た目よりも難しい問題である。

その1

まず思いつく解法は、CHECK制約を付加することだろう。座席ブロックが重複するケースにはどういうパターンがあり得るか、絵を描いて調べる人もいるかもしれない。そうすると、次のような制約が考えられるだろう。

```
CREATE TABLE Reservations
(reserver CHAR(10) NOT NULL PRIMARY KEY,
 start_seat INTEGER NOT NULL,
 finish_seat INTEGER NOT NULL,
   CHECK (start_seat <= finish_seat),
   CONSTRAINT No_Overlaps
   CHECK (NOT EXISTS
             (SELECT R1.reserver
                FROM Reservations AS R1
               WHERE Reservations.start_seat
                     BETWEEN R1.start_seat AND R1.finish_seat
                  OR Reservations.finish_seat
                     BETWEEN R1.start_seat AND R1.finish_seat));
```

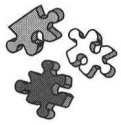
このCREATE TABLE文には座席ブロックの重複を防ぐだけでなく、別の人が同じ開始席番と終了席番を持つ座席ブロックを予約できないようにもする、うまい“トリック”が使われている。

ただ、この解法には問題が2つある。まず、CHECK制約内でのサブクエリはSQL-92のフルレベルでは許されているが、中間レベルでは許されていない。それゆえ、このCREATE TABLE文は現在リリースされているRDBMS上ではうまく動かないだろう。

また、この問題を回避できた場合でも、テーブルに最初の1行をINSERTするときに別の問題が起きる可能性がある。主キー制約とNOT NULL制約には問題はない。厄介なのは、CHECK制約が実行されるときに、SQLエンジンがサブクエリ内で空のReservationsテーブルをR1という名前でコピーする点だ。R1.start_seatとR1.finish_seatの値は、CREATE TABLE文の制約によってNULLであってはならない。しかし、R1は空だから、BETWEEN述語においてはNULLにならざるを得ない。そのため、この自己参照が制約チェッカーを混乱させて、絶対に1行目がINSERTできないという可能性が大いにあるのだ。

一番安全な方法は、「テーブルを宣言し、数行分データを入力して、その後にNo_Overlaps制約を付ける」というものだ。あるいは、「遅延制約で宣言しておいて、セッションが終わるときに戻す」という方法も選択できる。

パ|ズ|ル|
48 非グループ化
GROUP BYの逆演算



シシー・カブは、CompuServeに奇妙な質問を投げてきた。彼女は、こんなテーブルを持っていた。

```
CREATE TABLE Inventory
(goods CHAR(10) NOT NULL PRIMARY KEY,
 pieces INTEGER NOT NULL CHECK (pieces >= 0));
```

Inventory：在庫テーブル	goods：商品	pieces：個数

シシーは、このテーブルを分解しようとしていた。つまり、商品1個につき1行とするビュー、あるいは非テーブルを得ることである。例えば、('CD-ROM', 3) という行があるとすると、3行の ('CD-ROM', 1) に分解するのだ。聞かれる前に言っておくが、なぜ彼女がこんなことをしたいのかは知らない。純粋に練習問題だと考えてほしい。

SQLは「UN-COUNT(*) ... DE-GROUP BY ...」演算子を持っていないので、彼女の希望を実現するにはカーソルか、RDBMSベンダが提供する4GL[訳注1]を使う必要がある。正直に言うと、私ならSQLではなくレポート作成プログラム側で処理するだろう。なぜなら、結果がキーを持たないため、テーブルにならないからだ。だが、とりあえず練習問題だと思って、奇抜な解答を探してほしい。

その1

答え

まさに手続き的だが、あなたが使っているSQLに対応した4GLで、Inventoryテーブルから1行読み出し、その商品の個数に応じてループを回し、結果テーブルに書き込んでいくルーチンを作るという方法がある。

この方法では結果テーブルに対し、(SELECT SUM(pieces) FROM Inventory) 回の単一行挿入が必要になる。そのため、処理にはかなり時間がかかるだろう。

もっとよい解法はないだろうか？

訳注1：4th Generation Language（第4世代言語）の略。高度なプログラミングスキルがなくてもアプリケーションを開発できるように、言語自身や環境、ツールなどが実装を大きく支援してくれる点に特徴がある。SQLや手続き型SQL（OracleのPL/SQL、SQL ServerのT-SQLなど）も4GLの1つと言われる。

その2

答え

私は常々、SQLでは集合の観点から考える必要があると強調してきた。もっとよい解法を作る方法として、「ロシア農民のアルゴリズム」に基づいたテクニックを使い、自己挿入を繰り返すというものがある。これは、初期のコンピュータで乗除算を実行するために使われていたアルゴリズムで、数学史の教科書やコンピュータサイエンスの本に載っている。2進数の算術を基礎としており、アセンブラ言語の左右シフト演算で実装できる。

これでもまだ4GLを使う必要はあるが、それほど悪くない方法だろう。まずは、2つの作業テーブル（WorkingTable1、WorkingTable2）と最終的な結果を入れるテーブル（Answer）を作ろう。

```
CREATE TABLE WorkingTable1 -- 主キーを設定できない！
(goods CHAR(10) NOT NULL,
 pieces INTEGER NOT NULL);

CREATE TABLE WorkingTable2
(goods CHAR(10) NOT NULL,
 pieces INTEGER NOT NULL);

CREATE TABLE Answer
(goods CHAR(10) NOT NULL,
 pieces INTEGER NOT NULL);
```

手始めに、Inventoryテーブル内に1つしかない商品をAnswerテーブルに入れてみる。

```
INSERT INTO Answer
SELECT * FROM Inventory WHERE pieces = 1;
```

次に、残りのデータをWorkingTable1に入れる。

```
INSERT INTO WorkingTable1
SELECT * FROM Inventory WHERE pieces > 1;
```

次のコードは、WorkingTable1テーブルの各行から個数を半分（または半分＋1）にしたペア（2行）を作り、それをWorkingTable2テーブルへ挿入している。

```
INSERT INTO WorkingTable2
SELECT goods, FLOOR(pieces / 2.0)
  FROM WorkingTable1
 WHERE pieces > 1
UNION ALL
SELECT goods, CEILING(pieces / 2.0)
  FROM WorkingTable1
 WHERE pieces > 1;
```

FLOOR(x)はxより小さい最大の整数を、CEILING(x)はxより大きい最小の整数を返す。もし、あなたが使っているRDBMSがこれらの関数を持っていなければ、丸めと切り捨てを行う関数でも書ける。2ではなく2.0で割るのも重要なポイントだ。こうすることで、計算結果が小数まで出る。

今度は、WorkingTable2テーブルで個数が1にまで減った商品の行をAnswerテーブルに挿入し、WorkingTable1テーブルをクリアする。

```
INSERT INTO Answer
SELECT *
  FROM WorkingTable2
 WHERE pieces = 1;

DELETE FROM WorkingTable1;
```

次に、WorkingTable1とWorkingTable2の役割を入れ替え、両方のテーブルが空になるまでこのプロセスを繰り返す。これは直接的で簡単な手続き的コーディングだ。結果がテーブルからテーブルへ移動するプロセスを追ってみると、なかなか興味深い。アニメーションのコマ送りのように図示してみよう。

ステップ1：WorkingTable1へロードする

```
WorkingTable1          WorkingTable2

goods      pieces      goods      pieces
=================      =================
'Alpha'         4
'Beta'          5
'Delta'        16
'Gamma'        50
```

なお、('Epsilon', 1) は即座にAnswerテーブルへ移動してよい。

ステップ2：個数を半分にし、WorkingTable2の行を倍にする。そしてWorking Table1を空にする

```
WorkingTable1          WorkingTable2

goods      pieces      goods      pieces
=================      =================
                       'Alpha'          2
                       'Alpha'          2
                       'Beta'           2
                       'Beta'           3
                       'Delta'          8
                       'Delta'          8
                       'Gamma'         25
                       'Gamma'         25
```

ステップ3：両方のテーブルが空になるまで、このプロセスを繰り返す

```
WorkingTable1          WorkingTable2

goods      pieces      goods      pieces
=================      =================
'Alpha'          1
'Alpha'          1
'Alpha'          1
'Alpha'          1
'Beta'           1 -- 'Alpha'と'Beta'は
'Beta'           1 -- Answerテーブルへ入れてよい
'Beta'           1
'Beta'           2
'Delta'          4
'Delta'          4
'Delta'          4
'Delta'          4
'Gamma'         12
'Gamma'         12
'Gamma'         13
'Gamma'         13
```

1つのテーブルを空にするコストは、大抵の場合、とても安上がりだ。あるテーブルから別のテーブルへデータをコピーする場合も同様で、いちいち1行ずつ挿入していくより、該当データが記録されているディスク上の物理ブロックを丸ごとメモリ上のバッファへ移すほうが、コストはずっと低くて済む。

このコードは、片方の作業テーブルに結果を残すようにもできるが、今の方法だと作業テーブルが徐々に小さくなっていくので、バッファをより効率的に使える。また、このアルゴリズムでは「SELECT SUM(pieces) FROM Inventory」行分のメモリと、「log2((SELECT MAX(pieces) FROM Inventory))+1」回の移動が必要になるが、そのどちらにも無駄がない。

その3

答え この「合計ほぐし」問題に対して、ピーター・ロレンスはCompuServeに別解を送ってきた。まず、少なくとも商品個数（pieces）の最大値nまでの整数をすべて含む連番補助テーブルを作る。

```
CREATE TABLE Sequence (seq INTEGER PRIMARY KEY);
INSERT INTO Sequence VALUES (1), (2), ..., (n);
```

Sequence：連番補助テーブル	seq：連番

あるいは、次のコードを使ってもよい。

```
INSERT INTO Sequence(seq)
WITH Digits (digit)
AS (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9))
SELECT D1.digit + 10*D2.digit + 100*D3.digit + ...
  FROM Digits AS D1, Digits AS D2, ...
 WHERE D1.digit + 10*D2.digit + 100*D3.digit + ... > 0;
```

テーブルを作ったら、今度はSELECT文で合計（pieces）を1ずつにほぐす。

```
SELECT goods, 1 AS tally, seq
  FROM Inventory AS I1, Sequence AS S1
 WHERE I1.pieces >= S1.seq;
```

結果は、例えば次のようになるはずだ。

```
goods       tally   seq
=======================
'CD-ROM'        1     1
'CD-ROM'        1     2
'CD-ROM'        1     3
'Printer'       1     1
'Printer'       1     2
...
```

ロレンス氏は、このような連番テーブルの便利さを熟知している。彼はまた、ある日付や時間の範囲に含まれるすべての時刻を保持している「時間テーブル」も頻繁に使うという。時間テーブルを使うと、答えその3と同じ考え方で、「誰が、何時から何時までオフィスにいたか」を記録したデータベースから9時ちょうど、10時ちょうど……に誰がオフィスにいたかどうかを求められる。

この答えは、私のお気に入りだ。私の作業テーブル間で複雑なシャッフルを行う解法よりも、単純な結合のほうが高速に違いない。なお、『DBMS』誌に掲載された私のコラムを読んでこの方法を使った解答を思いついたのは、ロレンス氏だけではなかった。

その4

答え

メアリ・アッテンボローも同じ解答を思いついたが、彼女はさらに連番テーブルを作るのに、「ロシア農民のアルゴリズム」の別バージョンと言える新しい方法を考え出した。その後、ヴィニシウス・メロが作業テーブルの作り方を改善してくれた。解答となるのプロシージャは次のようなものだ。

```
BEGIN
DECLARE maxnum INTEGER NOT NULL;
DECLARE increment INTEGER NOT NULL;

INSERT INTO Sequence VALUES (1), (2);

-- Sequenceテーブルの行数はループのたびに倍になる
SET maxnum = (SELECT MAX(pieces) FROM Inventory);
SET increment = 2;
```

```
WHILE increment < maxnum
  DO INSERT INTO Sequence
     SELECT seq + increment FROM Sequence;
        SET increment = increment + increment;
  END WHILE;
```

もしSequenceテーブルのデータを、毎度プロシージャでロードするのではなく永続的に使いたいのなら、seqの最大値を超える個数の商品には手を付けないようにする必要がある。これは次のクエリで実現できる。

```
SELECT goods, 1 AS tally, seq
  FROM Inventory AS I1, Sequence AS S1
 WHERE I1.pieces >= S1.seq
   AND S1.seq BETWEEN 1
                  AND (SELECT MAX(pieces) FROM Inventory);
```

もう1つの対処法は、seqの最大値を超える個数の商品が存在した場合に、クエリ全体を拒否することである。

```
SELECT goods, 1 AS tally, seq
  FROM Inventory AS I1, Sequence AS S1
 WHERE I1.pieces >= S1.seq
   AND (SELECT MAX(pieces) FROM Inventory)
       <= (SELECT MAX(seq) FROM Sequence);
```

クエリ全体が実行される間、WHERE句のANDに続くサブクエリ式がずっと定数であることは明白だから、オプティマイザはこのサブクエリを一度だけ評価すればよい。その際、Sequenceテーブルにはインデックススキャンが行われるが、Inventoryテーブルのpieces列にはおそらくインデックスはないから、テーブルスキャンが発生するだろう。

ともあれ、まずはスカラサブクエリで個数の最大値を見つけ、その後にFROM句でその最大値をちょうどカバーできるだけの連番を作るわけだ。

その5

答え 別解として、重複行を持つテーブルと結合するという方法がある。

```
CREATE TABLE Repetitions --主キーなし
(pieces INTEGER NOT NULL,
 one INTEGER DEFAULT 1 NOT NULL
     CHECK (one = 1));
```

```
INSERT INTO Repetitions
VALUES (2,1), (2,1), (3,1), (3,1), (3,1)..;
```

```
INSERT INTO WorkingTable
SELECT goods, one
  FROM Repetitions AS R1
       CROSS JOIN
       Inventory AS I1
 WHERE I1.pieces = R1.pieces
   AND I1.pieces > 1;
```

もし、商品の個数がRepetitionsテーブルの上限より大きければ、十分な行がそろうまで行を挿入してほしい。

■計算の複雑さについての補足

Inventoryテーブルが全体でm行、商品の最大個数がn個だと仮定して、「ロシア農民のアルゴリズム」を使う方法と「シーケンス（連番）結合」を使う方法とを比較してみよう。結果テーブルの行数をrとすると、

```
r = (SELECT SUM(pieces) FROM Inventory)
```

で求められる。このことから、r <= (m × n) であることが分かる。

ロシア農民のアルゴリズムによれば、この問題を解くのに$O(\log_2(n))$回のループを必要とする。この方法は、繰り返すたびに問題を半分に切り詰めていく。結合や検索コストは一切かからない。ただし、1つのテーブルからもう1つのテーブルへ何度も何度も書き込むコストがかかる。1行が書き込まれる回数は$\log_2(pieces)$であり、テーブル全体としては$O(\log_2(r))$回になる。したがって、この方法の合計コストは$O(\log_2(n) + \log_2(r))$である。

他方、シーケンス結合では、Sequenceテーブルを作るための時間に加え、結合にかかる時間も考えねばならない。反復的なSequenceテーブルの作成にかかるコストは$O(n)$、InventoryテーブルとSequenceテーブルの結合にかかるコストは、純粋なクロス結合だから$O(m \times n)$である。適切なインデックスがあれば、Sequenceテーブル内の不要な値を参照せずに済むので、コストは$O(r)$まで減らせるかもしれない。したがって、合計コストは$O(n + r)$となる。これはロシア農民のアルゴリズムのコストよりも高い。

そうは言うものの、おそらく現実には、ロシア農民のアルゴリズムがシーケンス結合より速いということはないだろう。実際のところ、前者ではテーブル間での行の読み出しと書き込みに高いコストがかかるからだ。

パ|ズ|ル|
49 部品の数
データを等分割する

あなたは、複数の部品生産拠点から生産報告を受け取ったとしよう。その報告書には日付、生産拠点コード、およびその日に拠点に送られた原材料から生産された部品数が記録されている。これを管理するテーブルは次のとおりである。

```
CREATE TABLE Production
(production_center INTEGER NOT NULL,
 wk_date DATE NOT NULL,
 batch_nbr INTEGER NOT NULL,
 widget_cnt INTEGER NOT NULL,
 PRIMARY KEY (production_center, wk_date, batch_nbr));
```

Production：生産テーブル	production_center：生産拠点	wk_date：作業日
batch_nbr：バッチ番号	widget_cnt：部品数	

そこへ上司がやって来て、日付・拠点ごとに、すべてのバッチ工程（batch）から生産された部品数の平均が知りたいと言った。あなたは、「お安い御用です」と言ってそれを算出した。次の日、またその上司がやって来て、今度はそのデータを3つの同じサイズのバッチグループに分割してほしいと言った。生産関係の業務では、こういう種類の統計分析が重要な意味を持つのだ。

2つ目の上司の依頼を例を挙げて説明すると、もし2月24日に42番の拠点で21個のバッチが処理されたら、報告書には最初の7つのバッチから作られた部品の平均数、次の7つのバッチから作られた平均数、そして最後の7つのバッチから作られた平均数をそれぞれ表示する、ということである。そこで、日付・拠点ごとに、バッチグループと各グループの平均部品数を表示するようなクエリを書いてほしい。

その1

答え

出発点となるクエリは、ごく単純なものだ。

```
SELECT production_center, wk_date, COUNT(batch_nbr),
       AVG(widget_cnt)
  FROM Production
 GROUP BY production_center, wk_date;
```

2番目のクエリには、いくつかの仮定が必要になる。まず、バッチ番号は、毎日振り直される1からnまでの番号とする。バッチの数が3で割り切れない場合には、3つのグループでなるべく均等になるようにバッチを振り分ける。SQL-92のCASE式を使うと、あるbatch_nbrが3つのグループのどれに含まれるかが分かる。これらをビューとしてまとめると、次のように書ける[訳注1]。

```
CREATE VIEW Prod3 (production_center, wk_date, widget_cnt, third)
AS SELECT production_center, wk_date, widget_cnt,
          CASE WHEN batch_nbr <= (SELECT MAX(batch_nbr) / 3
                                    FROM Production AS P2
                                   WHERE P1.production_center
                                         = P2.production_center
                                     AND P1.wk_date = P2.wk_date)
               THEN 1
               WHEN batch_nbr > (SELECT MAX(batch_nbr * 2) / 3
                                    FROM Production AS P2
                                   WHERE P1.production_center
                                         = P2.production_center
                                     AND P1.wk_date = P2.wk_date)
               THEN 3
               ELSE 2
           END
     FROM Production AS P1;
```

もし、使用しているRDBMSでCASE式が使えないなら、同じことを次のように書く必要がある。

```
CREATE VIEW Prod3
(production_center, wk_date, third, batch_nbr, widget_cnt)
AS SELECT production_center, wk_date, 1, batch_nbr, widget_cnt
     FROM Production AS P1
    WHERE batch_nbr <= (SELECT MAX(batch_nbr) / 3
                          FROM Production AS P2
                         WHERE P1.production_center =
                               P2.production_center
                           AND P1.wk_date = P2.wk_date)
    UNION
    SELECT production_center, wk_date, 2, batch_nbr, widget_cnt
```

訳注1：これ以降でビューを作成しているDDL文はどちらもMAX(batch_nbr)を使用しているが、これにはbatch_nbrが連番を成していることが前提となる。batch_nbrが連番を成していない場合には、ROW_NUMBER関数などを使ってコード内で連番を生成する必要がある。

```
  FROM Production AS P1
 WHERE batch_nbr > (SELECT MAX(batch_nbr) / 3
                      FROM Production AS P2
                     WHERE P1.production_center =
                           P2.production_center
                       AND P1.wk_date = P2.wk_date)
   AND batch_nbr <= (SELECT 2 * MAX(batch_nbr) / 3
                      FROM Production AS P2
                     WHERE P1.production_center =
                           P2.production_center
                       AND P1.wk_date = P2.wk_date)
UNION
SELECT production_center, wk_date, 3, batch_nbr, widget_cnt
  FROM Production AS P1
 WHERE batch_nbr > (SELECT 2 * MAX(batch_nbr) / 3
                      FROM Production AS P2
                     WHERE P1.production_center =
                           P2.production_center
                       AND P1.wk_date = P2.wk_date);
```

その2

答え どちらの書き方をするにせよ、ビューを問い合わせるクエリは次のようになる。

```
SELECT production_center, wk_date, third, COUNT(*),
       AVG(widget_cnt)
  FROM Prod3
 GROUP BY production_center, wk_date, third;
```

パ|ズ|ル|
50 3分の2
GROUP BYの効果とありがたみ

これから、たくさんの書籍から論文を抜粋して論文集を編さんするところである（抜粋元の本は、国際標準図書番号〔ISBN〕で一意に識別される）。編さんにあたり、特定の3分野のうち2分野で論文を書いている著者を全員リストアップする必要が出てきた。そこで、3つの分野をパラメータとして取り、該当する著者のリストを出力するクエリを考えてほしい。

```
CREATE TABLE AnthologyContributors
(isbn CHAR(10) NOT NULL,
 contributor CHAR(20) NOT NULL,
 category INTEGER NOT NULL,
 ...,
 PRIMARY KEY (isbn, contributor));
```

AnthologyContributors：寄稿者テーブル	isbn：国際標準図書番号（ISBN）
contributor：寄稿者	category：論文の分野

その1

答え 最初に考えつくのは、次のようなGROUP BYを使った単純なクエリだろう。

```
SELECT isbn, contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE A1.category IN (:cat_1, :cat_2, :cat_3)
 GROUP BY isbn, contributor
HAVING COUNT(*) = 2;
```

だが、このクエリは2つの理由からうまく動かない。第1の理由は、テーブルの主キーにGROUP BYを適用すると、1行ごとにグループが作られてしまうこと。第2の理由は、1つの分野に2本の論文を書いている著者がいるかもしれないことだ。その場合、2つの論文がともにカウントされてしまう。こういうときには、COUNT(DISTINCT <式>)を使う必要がある。この第2の問題については、COUNT(DISTINCT A1.category)＝2とすることで、簡単に解決できる。

```
SELECT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE A1.category IN (:cat_1, :cat_2, :cat_3)
 GROUP BY contributor
HAVING COUNT(DISTINCT A1.category) = 2;
```

さて、これで解答はできたが、GROUP BYを使わずに同じ結果を得る方法も考えられないだろうか？ といっても、私は以下に紹介する解法を、どれ1つとして薦めるつもりはない。この練習問題のポイントは、GROUP BY句のありがたみを実感してもらうことにある。

その2

答え

問題の仕様からは、欲しい結果が3つのうち**任意の**2つの分野なのか、それとも結果に何らかの組み合わせの制限（例えば、分野1と分野2に書いているが分野3には書いていない、など）があるのかが分からない。もし後者であれば、解答は実に簡単に作れる。

```
SELECT A1.isbn, A1.contributor, :cat_1, :cat_2
  FROM AnthologyContributors AS A1,
       AnthologyContributors AS A2
 WHERE A1.contributor = A2.contributor -- 自己結合
   AND A1.category = :cat_1 -- 1：分野1
   AND A2.category = :cat_2 -- 2：分野2
   AND NOT EXISTS (SELECT * -- 3：しかし、分野3には執筆していない
                     FROM AnthologyContributors AS A3
                    WHERE A1.contributor = A3.contributor
                      AND A3.category = :cat_3);
```

その3

答え

3つの中から任意の2つを見つけるクエリは、少しトリッキーなコーディングにならざるを得ない。

```
SELECT isbn, contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE A1.category IN (:cat_1, :cat_2, :cat_3)
   AND EXISTS
       (SELECT *
          FROM AnthologyContributors AS A2
         WHERE A2.category IN (:cat_1, :cat_2, :cat_3)
           AND A1.category < A2.category
           AND A1.contributor = A2.contributor
           AND NOT EXISTS
               (SELECT *
                  FROM AnthologyContributors AS A3
                 WHERE A3.category IN (:cat_1, :cat_2, :cat_3)
                   AND A1.contributor = A3.contributor
                   AND A1.category <> A3.category
                   AND A2.category <> A3.category));
```

ちなみに3分野すべてに寄稿している著者を見つけたいなら、NOT EXISTSをEXISTSに変えるだけでよい。

1分野にだけ寄稿している著者を見つけるなら、次のようになる。

```
SELECT isbn, contributor, :cat_1
  FROM AnthologyContributors AS A1
 WHERE A1.category = :cat_1
   AND NOT EXISTS
       (SELECT *
          FROM AnthologyContributors AS A2
         WHERE A2.category IN (:cat_2, :cat_3)
           AND A1.isbn = A2.isbn
           AND A1.category <> A2.category);
```

これはいわば、先に示した「3分の2」を求めるクエリの“崩しバージョン”である。

その4

答え

ここまでには、何も問題はない。だが、GROUP BYを使わないという制限を思い出して、答えその3をちょっと見てほしい。この答えは少し間違っている。著者を結合する条件を忘れているのだ。

```
SELECT DISTINCT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE 2 = (SELECT COUNT(DISTINCT A2.category)
              FROM AnthologyContributors AS A2
             WHERE A1.contributor = A2.contributor
               AND A2.category IN (:cat_1, :cat_2, :cat_3));
```

その5

答え 次のSQL文は、いろいろと紹介してきた中で最良の解だ。最初の2つの分野にマッチして、3つ目にマッチしないという条件をうまく処理している。

```
SELECT DISTINCT contributor, :cat_1, :cat_2
  FROM AnthologyContributors AS A1
 WHERE (SELECT SUM(DISTINCT
                   CASE WHEN category = :cat_1
                        THEN 1
                        WHEN category = :cat_2
                        THEN 2
                        WHEN category = :cat_3
                        THEN -3 ELSE NULL END)
          FROM AnthologyContributors AS A2
         WHERE A1.contributor = A2.contributor
           AND A2.category IN (:cat_1, :cat_2, :cat_3)) = 3;
```

もちろん、再びGROUP BYを使うことも可能だ。

```
SELECT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE A1.category IN (:cat_1, :cat_2, :cat_3)
 GROUP BY contributor
HAVING (SELECT SUM(DISTINCT
                   CASE WHEN category = :cat_1
                        THEN 1
                        WHEN category = :cat_2
                        THEN 2
                        WHEN category = :cat_3
                        THEN -3 ELSE NULL END)) = 3;
```

その6

答え **3分野すべて**に書いている著者を見つけてみよう。そのためのSQL文は、答えその1やその4で示したクエリの基本的パターンを使えば、直接的に表現できる。

```
SELECT DISTINCT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE (SELECT COUNT(DISTINCT A2.category)
          FROM AnthologyContributors AS A2
         WHERE A1.contributor = A2.contributor
           AND A2.category IN (:cat_1, :cat_2, :cat_3)) = 3;
```

もちろん、GROUP BYを使うともっと直接的に書ける。

また、問題が「3分野のうち任意の1分野に寄稿している著者を探す」というものだったなら、答えは次のようになる。

```
SELECT DISTINCT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE category IN (:cat_1, :cat_2, :cat_3);
```

パ|ズ|ル|
51 予算と実支出の比較

集約と外部結合の合わせ技

C.コンラッド・キャディは、CompuServe上のGuptaフォーラムに簡単なSQLの問題を投稿した。彼は、予算と実支出を表す2つのテーブルを持っていた。どちらのテーブルにも、あるプロジェクトでどのように費用が必要とされているかが記録されている。また、予算テーブルと実支出テーブルとは1対多の関係にある。1つの予算に対する支払いが複数回に分割されることがあるからだ。テーブル定義は次のとおり。

```
CREATE TABLE Budgeted
(task INTEGER NOT NULL PRIMARY KEY,
 category INTEGER NOT NULL,
 est_cost DECIMAL(8,2) NOT NULL);
```

Budgeted：予算テーブル　　task：業務　　category：カテゴリ
est_cost：予算額

```
CREATE TABLE Actual
(voucher INTEGER NOT NULL PRIMARY KEY,
 task INTEGER NOT NULL REFERENCES Budgeted(task),
 act_cost DECIMAL(8,2) NOT NULL);
```

Actual：実支出テーブル　　voucher：領収書　　task：業務
act_cost：実支出額

キャディは、カテゴリ別に予算と実支出を比較したいと考えた。これは、例で見たほうが理解しやすいだろう。

```
Budgeted

task    category    est_cost
==========================
   1        9100      100.00
   2        9100       15.00
   3        9100        6.00
   4        9200        8.00
   5        9200       11.00
```

```
Actual

voucher   task   act_cost
=========================
      1      1      10.00
      2      1      20.00
      3      1      15.00
      4      2      32.00
      5      4       8.00
      6      5       3.00
      7      5       4.00
```

この2つのテーブルから彼が求めたかった結果は、次のとおりだ。

```
category   estimated   spent
============================
    9100      121.00   77.00
    9200       19.00   15.00
```

estimated列の121.00ドルは、カテゴリ9100に含まれる3つの業務に関する予算額(est_cost)の合計額である。spent列の77.00ドルは、それら3つの業務に関連する4つの支出(act_cost)の合計額だ(3つの支出が業務〔task〕1に、1つが業務2に紐付いている。業務3に支出はない)。

彼は、次のようなクエリを試した。

```
SELECT category, SUM(est_cost) AS estimated,
       SUM(act_cost) AS spent
  FROM (Budgeted LEFT OUTER JOIN Actual
        ON Budgeted.task = Actual.task)
 GROUP BY category;
```

だが、思うような結果は得られなかった。

```
category   estimated   spent
============================
    9100      321.00   77.00
    9200       30.00   15.00
```

問題は、100.00ドルが結合の中で3回カウントされ、121.00ドルではなく321.00ドルとなることと、11.00ドルが2回カウントされ、19.00ドルではなく30.00ドルとなること

だ。前出の予算および実支出テーブルを使って、彼が本当に求めたかった結果を1つのSQL文で簡単に得る方法はあるだろうか？

その1

答え　ボブ・バドゥアールは「SQL-89でもビューを作ればできるぞ」と報告してきた。まず、

```
CREATE VIEW cat_costs (category, est_cost, act_cost)
AS SELECT category, est_cost, 0.00
     FROM Budgeted
   UNION
   SELECT category, 0.00, act_cost
     FROM Budgeted, Actual
    WHERE Budgeted.task = Actual.task;
```

というビューを作り、それから、

```
SELECT category, SUM(est_cost), SUM(act_cost)
  FROM cat_costs
 GROUP BY category;
```

というクエリを実行すればOKだ。

また、SQL-92では、各業務の支出合計額と予算テーブルにあるカテゴリを、次のようなクエリで結び付けることができる。

```
SELECT B1.category, SUM(est_cost), SUM(spent)
  FROM Budgeted AS B1
       LEFT OUTER JOIN
       (SELECT task, SUM(act_cost) AS spent
          FROM Actual
         GROUP BY task) AS A1
       ON A1.task = B1.task
 GROUP BY B1.category;
```

左外部結合は、支出がゼロだった場合に対処している。

その2

答え 次に紹介するのは、コロンビアのフランシスコ・モレノから届いた解答だ。GROUP BY句とスカラサブクエリを併用している。サブクエリ内のMIN関数とMAX関数の値に注目してほしい[訳注1]。

```
SELECT category, SUM(B1.est_cost) AS estimated,
       (SELECT SUM(T1.act_cost)
          FROM Actual AS T1
         WHERE T1.task BETWEEN MIN(B1.task)
                           AND MAX(B1.task)) AS spent
  FROM Budgeted AS B1
 GROUP BY category;
```

訳注1：サブクエリ内のWHERE句でMAX/MIN関数を使用していることに驚くかもしれないが、これは適法な構文である。なぜなら、外側のB1テーブルを集約しているため、SELECT句で直接参照できるB1テーブルの列は集約キーであるcategoryだけであり、残りの列は集約関数の形でしか参照できなくなるからだ。

パ|ズ|ル|
52

部署の平均人数

2段階の集約

ダレン・レイスはGupta SQLBaseを使って、集約した結果をさらに集約できないものかと試していた。しかし、どうしても一時テーブルかビューを使わないことにはできなかった。例えば、

```
Personnel

emp_name   dept_id
==================
'Daren'    'Acct'
'Joe'      'Acct'
'Lisa'     'DP'
'Helen'    'DP'
'Fonda'    'DP'
```

Personnel：社員テーブル　emp_name：社員名　dept_id：部署ID
'Acct'：経理部　'DP'：情報システム部

というサンプルデータがあった場合、まずこれをdept_idで集約する。

```
SELECT dept_id, COUNT(*)
  FROM Personnel
 GROUP BY dept_id;
```

すると、次のような結果が得られる。

```
dept_id   COUNT(*)
==================
'Acct'            2
'DP'              3
```

だが、彼が求めたかったのは部署の規模（明確に言えば人数）の平均だった。彼がビューを使って考えたクエリは以下のとおりだ。

まず、ビューを作る。

```
CREATE VIEW DeptView (dept_id, tally)
AS SELECT dept_id, COUNT(*)
     FROM Personnel
    GROUP BY dept_id;
```

そして次のSQLを実行する。結果は、(2 + 3) ÷ 2 = 2.5人だ。

```
SELECT AVG(tally) FROM DeptView;
```

レイス氏は、CompuServe上のCentura (Guptaの旧名) フォーラムで、「これと同じことを一時テーブル (またはビュー) を使わずにできないか」と質問したのである。すると、2つの返答があった。1つは、

```
SELECT AVG(DISTINCT dept_id)
  FROM Personnel;
```

というもの。もう1つは、

```
SELECT COUNT(*) / COUNT(DISTINCT dept_id)
  FROM Personnel;
```

というものだった。

それでは、今回の問題だ。それぞれの返答について、どこがまずいのかを指摘してほしい。

その1

答え 最初の返答は全くお話にならない。部署コードは数字ではなくアルファベットだ。部署の人数とは何の関係もない。

2番目の返答はずっとマシだ。実際、先ほどのサンプルデータに対しては正しい結果を返してくれる。COUNT(*) = 5、COUNT(DISTINCT dept_id) = 2なので、答えは2.5となる。まさに欲しかった結果だ。

だが、次のようなケースで問題が起きる。新たに、Larry、Moe、Curlyという3人を社員として雇ったとしよう。彼らはまだ、どの部署にも配属されていない。すると、Personnelテーブルはこんな風になる。

```
Personnel

emp_name   dept_id
==================
'Daren'    'Acct'
'Joe'      'Acct'
'Lisa'     'DP'
'Helen'    'DP'
'Fonda'    'DP'
'Larry'    NULL
'Moe'      NULL
'Curly'    NULL
```

このテーブルに対しては、COUNT(*)＝8だが、COUNT(DISTINCT dept_id)＝2となってしまう。これは、dept_idを数える前にCOUNT関数がNULLを除外するからだ。そのため、今度は答えが4になる。つまり、こういう場合には新人3人をどう扱うかを、私たちの側で決める必要がある。選択肢は次のとおりだ。

1. 1人につき、1つの新しい部署を割り当てる（部署の合計数は5）
2. 3人ともNULLで表示される新部署の配属とする（部署の合計数は3）
3. 3人とも情報システム（'DP'）部の所属とする
4. 3人とも経理（'Acct'）部の所属とする
5. 1人が経理部、2人が情報システム部の所属とする
6. 1人が経理部、2人が新部署の所属とする
7. 1人が情報システム部、2人が経理部の所属とする
8. 1人が情報システム部、2人が新部署の所属とする
9. 1人が新部署、2人が経理部の所属とする
10. 1人が新部署、2人が情報システム部の所属とする
11. 1人が新部署、2人がさらに別の新部署の所属とする
12. 1人が経理部、1人が情報システム部、1人が新部署の所属とする

13. 1人が経理部、1人が新部署、1人がさらに別の新部署の所属とする

14. 1人が情報システム部、1人が新部署、1人がさらに別の新部署の所属とする

そうすると、部署の平均規模は、ある場合には8÷2＝4人となり、別の場合には8÷5＝1.6人となる。もし、レイス氏が自身の考えた方法にこだわるなら、次のようなビューが得られるだろう。

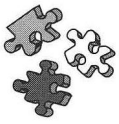
```
dept_id    COUNT(*)
==================
'Acct'            2
'DP'              3
NULL              3
```

NULLは、ビューの中ではそれ自身が1つのグループを成しているので、最終結果は(2＋3＋3)÷3≒2.7人となる。

パ|ズ|ル|
53

テーブルを列ごとに折りたたむ

自己結合とCASE式（その1）



ロバート・ブラウンは、2004年にある問題を送ってきた。次のようなテーブルがあるとしよう。

```
CREATE TABLE Foobar
(lvl INTEGER NOT NULL PRIMARY KEY,
 color VARCHAR(10),
 length INTEGER,
 width INTEGER,
 hgt INTEGER);
```

lvl：レベル　color：色　length：長さ　width：幅　hgt：高さ

```
INSERT INTO Foobar
VALUES (1, 'RED', 8, 10, 12),
       (2, NULL, NULL, NULL, 20),
       (3, NULL, 9, 82, 25),
       (4, 'BLUE', NULL, 67, NULL),
       (5, 'GRAY', NULL, NULL, NULL);
```

このテーブルに対し、lvl列の「底から頂点」（5が底、1が頂点）へ向かう順序に従い、各列を折りたたんだ結果を返すクエリを書きたい。このサンプルデータでは、次の結果が返されるクエリが答えとなる。

```
('GRAY', 9, 67, 25)
```

つまり、各列を底辺から上へ見ていき、最初に見つかった非NULL値をとる、というルールである。すると、color列についてはすぐに'GRAY'が見つかる。length列では、レベル3で9が見つかる。同様に、width列は67（レベル4）、hgt列は25（レベル3）となる。別の言い方をすれば、「より頂点に近いレベルでセットされた値が、より底に近いレベルのNULLでない値によって上書きされる」ということだ。

その1

答え ジョン・ギルソンは、2つの解答を思いついた。

【方法1】（スカラサブクエリを使う）

```
SELECT (SELECT color FROM Foobar WHERE lvl = M.lc) AS color,
       (SELECT length FROM Foobar WHERE lvl = M.ll) AS length,
       (SELECT width FROM Foobar WHERE lvl = M.lw) AS width,
       (SELECT hgt FROM Foobar WHERE lvl = M.lh) AS hgt
  FROM (SELECT MAX(CASE WHEN color IS NOT NULL
                        THEN lvl END) AS lc,
               MAX(CASE WHEN length IS NOT NULL
                        THEN lvl END) AS ll,
               MAX(CASE WHEN width IS NOT NULL
                        THEN lvl END) AS lw,
               MAX(CASE WHEN hgt IS NOT NULL
                        THEN lvl END) AS lh
          FROM Foobar) AS M;
```

【方法2】

```
SELECT MIN(CASE WHEN Foobar.lvl = M.lc
                THEN Foobar.color END) AS color,
       MIN(CASE WHEN Foobar.lvl = M.ll
                THEN Foobar.length END) AS length,
       MIN(CASE WHEN Foobar.lvl = M.lw
                THEN Foobar.width END) AS width,
       MIN(CASE WHEN Foobar.lvl = M.lh
                THEN Foobar.hgt END) AS hgt
  FROM (SELECT MAX(CASE WHEN color IS NOT NULL
                        THEN lvl END) AS lc,
               MAX(CASE WHEN length IS NOT NULL
                        THEN lvl END) AS ll,
               MAX(CASE WHEN width IS NOT NULL
                        THEN lvl END) AS lw,
               MAX(CASE WHEN hgt IS NOT NULL
                        THEN lvl END) AS lh
          FROM Foobar) AS M
       INNER JOIN
       Foobar
       ON Foobar.lvl IN (M.lc, M.ll, M.lw, M.lh);
```

その2

答え　次に見せるSQL文は、私が考えたものだ。COALESCE関数は、引数として与えられた列を与えられた順番で見ていくので、底（lvl = 5）から頂上（lvl = 1）へ向かってスキャンし、最初のNULLでない値を取り出すロジックを簡単に実装できる。

```
SELECT COALESCE(F5.color, F4.color, F3.color, F2.color, F1.color)
                AS color,
       COALESCE(F5.length, F4.length, F3.length, F2.length,
                F1.length) AS length,
       COALESCE(F5.width, F4.width, F3.width, F2.width, F1.width)
                AS width,
       COALESCE(F5.hgt, F4.hgt, F3.hgt, F2.hgt, F1.hgt)
                AS hgt
  FROM Foobar AS F1, Foobar AS F2, Foobar AS F3, Foobar AS F4,
       Foobar AS F5
 WHERE F1.lvl = 1
   AND F2.lvl = 2
   AND F3.lvl = 3
   AND F4.lvl = 4
   AND F5.lvl = 5;
```

パ|ズ|ル|
54 隠れた重複行
自己結合とCASE式（その2）

ロニー・ワイスは、彼がDB2のアプリケーションで抱えていた問題を送ってきた。

ある店が、約2万人の顧客を抱えているとしよう。月日が経つにつれ、従業員たちはタイプミスや入力間違い、同じ家族の人を異なる顧客として登録する（その店では顧客を家族単位で登録していた）、といったデータの不備があることに気づき始めた。そこで、彼らは「隠れた重複行」のレポートを作り、顧客テーブルのデータをメンテナンスすることにした。

隠れた重複行とは、同じ姓を持ち、住所関連の5列のうち2列のデータが一致する複数の行のことだ。住所関連の5列とは、「名前（first_name）」「番地（street_address）」「市町村名（city_name）」「州コード（state_code）」「電話番号（phone_nbr）」である。

彼らが最初に考えたSQL文は、こんな感じである。

```
CREATE VIEW Dups (custnbr, last_name, first_name,
street_address, city_name, state_code, phone_nbr, m)
AS
SELECT C0.custnbr, C0. last_name, C0.first_name,
       C0.street_address, C0.city_name,
       C0.state_code, C0.phone_nbr,
       (CASE WHEN C0.first_name = C1.first_name
             THEN 1 ELSE 0 END)
     + (CASE WHEN C0.street_address = C1.street_address
             THEN 1 ELSE 0 END)
     + (CASE WHEN C0.city_name = C1.city_name
             THEN 1
             ELSE 0 END)
     + (CASE WHEN C0.state_code = C1.state_code
             THEN 1
             ELSE 0 END)
     + (CASE WHEN C0.phone_nbr = C1.phone_nbr
             THEN 1
             ELSE 0 END) AS m
  FROM Customers AS C1, Customers AS C0
 WHERE C0.custnbr <> C1.custnbr
   AND C0.last_name = C1.last_name;
```

```
SELECT DISTINCT *
  FROM Dups
 WHERE m >= 2
 ORDER BY last_name;
```

顧客は2度以上現れる可能性があるので、DISTINCTが必要だ。例えば、A1がA3、A5、A6と一致したら、Dupsビューには3行のA1が現れる。

この解の欠点は、パフォーマンスが良くないことだ。何とかして改善してほしい。

その1

答え　独CODATA社のヨハネス・ベッヒャーは、次のような解を考えた。

```
SELECT C0.custnbr
  FROM Customers AS C0
 WHERE EXISTS (
         SELECT *
           FROM Customers AS C1
          WHERE C0.last_name = C1.last_name
            AND C0.custnbr <> C1.custnbr
            AND (CASE WHEN C0.first_name = C1.first_name
                      THEN 1 ELSE 0 END)
              + (CASE WHEN C0.street_address = C1.street_address
                      THEN 1 ELSE 0 END)
              + (CASE WHEN C0.city_name = C1.city_name
                      THEN 1 ELSE 0 END)
              + (CASE WHEN C0.state_code = C1.state_code
                      THEN 1 ELSE 0 END)
              + (CASE WHEN C0.phone_nbr = C1.phone_nbr
                      THEN 1 ELSE 0 END) >= 2);
```

「C0.custnbr <> C1.custnbr」という条件は、どの行も自分自身を重複行と見なすことを防ぐためのものだ。また、ベッヒャーはCASE式を、本来計算列を書く場所であるSELECT句からWHERE句に移している。こうすることで、オプティマイザが式を評価するときに“短絡評価”を利用できるかもしれない。短絡評価では、述語の真偽が判明したら即座に計算が止まり、答えが返される。SELECT句の計算列では“完全評価”を行う必要があるため、ビューの中で実体化されるだろう。

この解をもう一歩発展させてみよう。もし、オプティマイザがCASE式を評価する順序（左から右か／右から左か）と、最も一致しやすい列が分かれば、CASE式の並び順を調整することでパフォーマンスを高められる。例えば、city_nameとstate_codeについては一般的にほとんどスペルミスはないだろうから、一番最後に持ってきたほうがよい。それらに比べて、人名と番地は誤記が多い。

だが、この種の問題に対する最適解は、結局名寄せツールを使うことだろう。

パ|ズ|ル|
55

競走馬の入賞回数

外部結合で行と列を変換

突然だが、あなたはたった今、競馬のブックメーカー[訳注1]にDBマネージャーとして雇われたと想像してほしい。胴元は、統計を取るためにレースの記録を残している。記録先であるテーブルの基本的な形は次のとおりだ。

```
CREATE TABLE RacingResults
(track_id CHAR(3) NOT NULL,
 race_date DATE NOT NULL,
 race_nbr INTEGER NOT NULL,
 win_name CHAR(30) NOT NULL,
 place_name CHAR(30) NOT NULL,
 show_name CHAR(30) NOT NULL,
 PRIMARY KEY (track_id, race_date, race_nbr));
```

RacingResults：レース結果テーブル	track_id：トラックID
race_date：レース日	race_nbr：レース番号
win_name：1着の馬名	place_name：2着の馬名
show_name：3着の馬名	

track_id列はレースが開催されたトラックの名前、race_dataはレースの開催日、race_nbrは各レースに振られる番号である。また、win_name、place_name、show_nameは、それぞれ1着、2着、3着になった馬の名前である。

ある日、雇い主の胴元がやってきて、「各馬がこれまで何回入賞したかを知りたい」と言ってきた。この要求をかなえるためには、どんなSQL文を書けばよいだろうか？

その1

答え　ここで言う「入賞」とは、3着以内に入ることである。具体的に何着だったかは問わない。最初のステップは、集約情報を持った次のようなビューを作ることである。

訳注1：ブックメーカーとは、主に欧米での賭けの胴元のこと。禁止されている国も少なくないが、イギリスなどでは合法的に認められている。

```
CREATE VIEW InMoney (horse, tally, position)
AS SELECT win_name, COUNT(*), 'win_name'
     FROM RacingResults
    GROUP BY win_name
   UNION ALL
   SELECT place_name, COUNT(*), 'place_name'
     FROM RacingResults
    GROUP BY place_name
   UNION ALL
   SELECT show_name, COUNT(*), 'show_name'
     FROM RacingResults
    GROUP BY show_name;
```

次に、このビューから最終的な合計を取得する。

```
SELECT horse, SUM(tally)
  FROM InMoney
 GROUP BY horse;
```

SELECT句のリストに定数を置いているのには、2つの理由がある。第1の理由は、UNION ALLを使っても重複行が発生しないようにするためである。第2の理由は、胴元から各馬が何着に何回入ったのか詳しく知りたいと言われたときにも、次のような簡単な変更で済むからである。

```
SELECT horse, position, SUM(tally)
  FROM InMoney
 GROUP BY horse, position;
```

その2

答え　もし、すべての馬名（horse）を保持するHorseNamesテーブル[訳注2]があるなら、次のようにも書ける。

訳注2：HorseNames（馬名）テーブルの定義は、例えば次のようになるだろう。

```
CREATE TABLE HorseNames (horse VARCHAR(32) NOT NULL PRIMARY KEY);
```

```
SELECT H1.horse, COUNT(*)
  FROM HorseNames AS H1, RacingResults AS R1
 WHERE H1.horse IN (R1.win_name, R1.place_name, R1.show_name)
 GROUP BY H1.horse;
```

外部結合を使えば、RacingResultsテーブルに記録のない馬も結果に含めることができる[訳注3]。ここからは、重要な設計上の原理が見て取れる。すなわち、「あるものが実体なのか属性なのかを決めることは難しい」ということである。馬は実体だから、そのデータはテーブルでは行として保持されるべきだ。しかし、馬の名前は、RacingResultsテーブルの3つの列で値としても使われている。

その3

答え

HorseNamesテーブルを使ったもう1つの解法は、入賞回数のトータルをスカラサブクエリで求めるというものだ。

```
SELECT H1.horse,
       (SELECT COUNT(*)
          FROM RacingResults AS R1
         WHERE R1.win_name = H1.horse)
     + (SELECT COUNT(*)
          FROM RacingResults AS R1
         WHERE R1.place_name = H1.horse)
     + (SELECT COUNT(*)
          FROM RacingResults AS R1
         WHERE R1.show_name = H1.horse)
  FROM HorseNames AS H1;
```

これも正解だが、実行コストは高くなるだろう。

訳注3：ただしその場合、COUNT(*)を使うことはできないことに注意しよう。COUNT(*)は行数を数えるため、一度も入賞経験のない馬の入賞回数を「1」にしてしまう。正しくはCOUNT(R1.track_id)などに変える必要がある。

パ|ズ|ル|
56

ホテルの部屋番号

連番を入れていくUPDATE文

ロン・ハイナーは、CompuServeに以下のような質問を投稿してきた。

彼はデータ変換のプロジェクトに携わっていて、あるホテルの部屋番号を管理するHotelテーブルの主キーの一部の値を自動的に割り当てなければならなかった。

Hotelテーブルの定義は次のとおりだ。フロア番号は主キーの一部であり、ビル(Bldg)テーブルを参照する外部キー制約が定義されている。

```
--room_nbrにNULLを許可するため、現段階では主キーを定義していない
CREATE TABLE Hotel
(floor_nbr INTEGER NOT NULL,
 room_nbr INTEGER,
 FOREIGN KEY floor_nbr REFERENCES Bldg(floor_nbr));
```

Hotel：ホテルテーブル　　floor_nbr：フロア(階)番号　　room_nbr：部屋番号

これから割り当てる主キーの一部とは、部屋番号(room_nbr)である。部屋番号は、フロア番号をxとすると、x01から始まる連番になっている。また、このホテルはそれほど大きくないので、部屋番号は3桁あれば十分であることが分かっている。

目下作業中のHotelテーブルのデータは次のとおりだ。

```
floor_nbr   room_nbr
====================
        1       NULL
        1       NULL
        1       NULL
        2       NULL
        2       NULL
        3       NULL
```

WATCOM(および当時のRDBMS製品)は、実装依存のNUMBER(*)関数を持っていた。この古い関数は1から始まり、行が呼び出されるたびにカウントアップされた連番を返してくれる。一方、現在の標準SQLでも、ROW_NUMBER() OVER(<ウィンドウ式>)関数によって、やはり連番を簡単に得ることができる。

では、これらの連番関数(もしくはそれと類似の方法)を利用して、自動的にroom_nbr列を増加させる簡単な方法はあるだろうか？ 当時ハイナー氏は、フロアが変わるた

びに連番の下2ケタを1に戻すため、「GROUP BY floor_nbr」を使おうと考えていた。

その1

答え WATCOMは、次のような方法をサポートしている。まず、room_nbr列を埋めるために、データベース全体を通して更新するパスを作る。ただし、この"トリック"は、Hotelテーブルがソート順に更新されるという保証がなければ使えない。だが偶然にも、WATCOMではUPDATE文中のORDER BY句でそれを保証できる。

```
UPDATE Hotel
   SET room_nbr = (floor_nbr * 100) + NUMBER(*)
 ORDER BY floor_nbr;
```

このUPDATE文を実行すると、次のような結果になる。

```
floor_nbr   room_nbr
====================
        1        101
        1        102
        1        103
        2        204
        2        205
        3        306
```

次に、下の2つのUPDATE文を実行する。

```
UPDATE Hotel
   SET room_nbr = (room_nbr - 3)
 WHERE floor_nbr = 2;
```

```
UPDATE Hotel
   SET room_nbr = (room_nbr - 5)
 WHERE floor_nbr = 3;
```

これによって、正しい結果が得られる。

```
floor_nbr   room_nbr
====================
        1        101
        1        102
        1        103
        2        201
        2        202
        3        301
```

この方法の惜しいところは、このホテルの部屋番号について詳しく知っていなければ使えないことだ（何階建てなのか、ある階の部屋数はいくつなのか……）。さて、ORDER BY句を使わずにもっとうまくやる方法はないだろうか？

答え

その2

私なら、「SQL文を書くためにSQLを使う」という方法を使いたい。この見事なトリックは、まだあまり知られていない。次に示すクエリの、シングルクォーテーション（'）で囲まれた箇所をよく見てほしい。数値を文字列に変換することも忘れないように。

```
SELECT DISTINCT
'UPDATE Hotel
    SET room_nbr = ('
          || CAST(floor_nbr AS CHAR(1))
          || ' * 100) + NUMBER(*) WHERE floor_nbr = '
          || CAST(floor_nbr AS CHAR(1)) || ';'
   FROM Hotel;
```

このSQL文は、次のような1列の結果テーブルを書き出す。

```
UPDATE Hotel
   SET room_nbr = (1 * 100) + NUMBER(*) WHERE floor_nbr = 1;
UPDATE Hotel
   SET room_nbr = (2 * 100) + NUMBER(*) WHERE floor_nbr = 2;
UPDATE Hotel
   SET room_nbr = (3 * 100) + NUMBER(*) WHERE floor_nbr = 3;
```

あとはこの列をテキストとしてコピーして、あなたが使っている対話式のSQLツールやバッチファイルの中に貼り付けて実行すればよい。この方法は、テーブル内の行の順

序に依存しない。

あるいは、これをストアドプロシージャ本体の中に入れて、floor_nbrをパラメータとして渡すようにすることもできる。いったんプロシージャを書いてコンパイルしてしまえば、後は何もしなくて済む。

その3

答え 古いSQLでは難解だったこの問題も、SQL-99ならあっさり解ける。

```
UPDATE Hotel
SET room_nbr
    = (floor_nbr * 100)
      + ROW_NUMBER() OVER (PARTITION BY floor_nbr);
```

パズル 57

欠番探し バージョン1

集合指向言語で数列を扱う（その1）

今回取り上げるのは、ニュースグループで頻繁に質問される古典的なSQLプログラミングの問題である。できる限り簡単に言うと、「テーブルの1列に格納された一意な数が、歯抜けのない完全な連番を成しているか、それともギャップ（1つ以上連続する欠番）を含むかどうかを調べる」というものだ。サンプルデータには次のものを使おう。

```
CREATE TABLE Numbers (seq INTEGER NOT NULL PRIMARY KEY);
```

Numbers：自然数テーブル	seq：連番

```
INSERT INTO Numbers
VALUES (2), (3), (5), (7), (8), (14), (20);
```

その1

答え

1からnまで抜けのない連番を成しているかどうかを調べるのは、とても簡単だ。次のクエリによって、どこにギャップがあるかは分からないが、とにかくギャップがあるか否かは判明する。

```
SELECT CASE WHEN COUNT(*) = MAX(seq)
            THEN 'Sequence'
            ELSE 'Not Sequence' END
  FROM Numbers;
```

次のクエリも処理は明白だが、集合が1（または0）から始まるかどうかはチェックしていない。ただ、当該の範囲の中にギャップが存在するか調べるだけである。

```
SELECT CASE WHEN COUNT(*) + MIN(seq) - 1 = MAX(seq)
            THEN 'Sequence'
            ELSE 'Not Sequence' END
  FROM Numbers;
```

その2

答え 次のクエリは、ギャップの開始と終了の値を見つけ出す。ただし、以降のクエリではNumbersテーブルにゼロを「番兵」として追加しておく必要がある。

```
SELECT N1.seq + 1 AS gap_start, N2.seq - 1 AS gap_end
  FROM Numbers AS N1, Numbers AS N2
 WHERE N1.seq +1 < N2.seq
   AND (SELECT SUM(seq)
          FROM Numbers AS Num3
         WHERE Num3.seq BETWEEN N1.seq AND N2.seq)
       = (N1.seq + N2.seq);
```

この問題では、最初の値がネックになることが多い。例えば、次のクエリは「ギャップの開始の値」と「seq列の最大値より1つ大きい値」しか返さない。そして、Numbersの中に0がない場合、このクエリでは1は出てこない。

```
-- ギャップの最初の値のみを見つける：うまくいかない
SELECT N1.seq + 1
  FROM Numbers AS N1
       LEFT OUTER JOIN
       Numbers AS N2
       ON N1.seq = N2.seq - 1
 WHERE N2.seq IS NULL;
```

最初のギャップを見つけるには、もっと複雑だが精密な方法を使う必要がある。

```
--最初のギャップを見つける
SELECT CASE WHEN MAX(seq) = COUNT(*)
            THEN MAX(seq) + 1
            WHEN MIN(seq) > 1
            THEN 1
            WHEN MAX(seq) <> COUNT(*)
            THEN (SELECT MIN(seq) + 1
                    FROM Numbers
                   WHERE (seq + 1)
                         NOT IN (SELECT seq FROM Numbers))
            ELSE NULL END
  FROM Numbers;
```

CASE式の最初の分岐は、「もしNumbersにギャップがなければ、seq列の最大値＋1の値を結果とする」という意味だ。2番目の分岐は、1がギャップの場合には1を出力する。そして3番目の分岐は、ギャップの最小値を求める。

その3

答え お決まりの連番補助テーブルとSQL-99の集合演算子を使えば、次のように書ける。

```
SELECT seq FROM Sequence
 WHERE seq <= (SELECT MAX(seq) FROM Numbers)
EXCEPT ALL
SELECT seq FROM Numbers;
```

どちらのテーブルにも重複行はないので、EXCEPT ALLを使ったことに注目してほしい。もっとも、SQLの新機能に必ずしもオプティマイザが対応してくれるとは限らないが。

パ|ズ|ル|
58 欠番探し バージョン2

集合指向言語で数列を扱う（その2）

さて今度は、連番中のギャップ（1つ以上連続する欠番）を見つけるという古典的問題の第2バージョンを考えてみよう。あなたは解答をいくつ思いつくことができるだろうか？ また、SQL-92やSQL-99の機能を利用すると、どんなうまい方法が見つかるだろうか？

```
CREATE TABLE Tickets
(buyer_name CHAR(5) NOT NULL,
 ticket_nbr INTEGER DEFAULT 1 NOT NULL
            CHECK (ticket_nbr > 0),
 PRIMARY KEY (buyer_name, ticket_nbr));
```

Tickets：チケットテーブル　　buyer_name：バイヤー名　　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);
```

その1

答え

カナダ・トロント在住のトム・モロー[訳注1]は、有名なSQLの書き手である。彼は、UNION ALLを使わずに解く方法を考え出した。持っているチケットの番号にギャップがあるバイヤーを調べるものだが、「ギャップを埋める」ことまではしてくれない。例えば、バイヤーD氏は（1, 6, 7, 9）のチケットを持っているので、（2, 3, 4, 5, 8）がギャップとして返される。

しかし、トムの方法ではバイヤーA氏の持っているチケットにはギャップがないと見なされて、A氏は該当するバイヤーに含まれない。

訳注1：SQL Serverを主な専門とするシステムコンサルタント。

```
SELECT buyer_name
  FROM Tickets
 GROUP BY buyer_name
HAVING NOT (MAX(ticket_nbr) - MIN(ticket_nbr) <= COUNT(*));
```

もし、チケット番号が比較的小さい数に収まるなら、1からnまでの連番を持つテーブル（Sequence）を使ってもよい。

```
SELECT DISTINCT T1.buyer_name, S1.seq
  FROM Tickets AS T1, Sequence AS S1  -- Sequence：連番テーブル
 WHERE seq <= (SELECT MAX(ticket_nbr) -- 集合の範囲を設定する
                 FROM Tickets AS T2
                WHERE T1.buyer_name = T2.buyer_name)
   AND seq NOT IN (SELECT ticket_nbr  -- 欠番を求める
                     FROM Tickets AS T3
                    WHERE T1.buyer_name = T3.buyer_name);
```

この場合のもう1つの“トリック”は、連番中に1が欠けているケースのために、境界値として0をTicketsテーブルへ追加しておくことだ。

その2

答え

リバプール・ファンも、連番（Sequence）テーブルを利用するクエリを提案してきた。ただ、彼のクエリにはSequence.seq列の上限値についての条件が欠けていた。次に示すのはそれを付け加えたSQL文である。

```
SELECT DISTINCT T1.buyer_name, S1.seq
  FROM Tickets AS T1, Sequence AS S1
 WHERE NOT EXISTS
       (SELECT *
          FROM Tickets AS T2
         WHERE T2.buyer_name = T1.buyer_name
           AND T2.ticket_nbr = S1.seq)
   AND S1.seq <= (SELECT MAX(ticket_nbr)
                    FROM Tickets AS T3
                   WHERE T3.buyer_name = T1.buyer_name);
```

その3

答え　オムニバズは、Sequenceテーブルの上限値を設定するクエリを考えた。

```
SELECT DISTINCT T2.buyer_name, T2.ticket_nbr
  FROM (SELECT T1.buyer_name, S1.seq AS ticket_nbr
          FROM (SELECT buyer_name, MAX(ticket_nbr)
                  FROM Tickets
                 GROUP BY buyer_name) AS T1(buyer_name, max_nbr),
               Sequence AS S1
         WHERE S1.seq <= max_nbr) AS T2
       LEFT OUTER JOIN Tickets AS T3
       ON T2.buyer_name = T3.buyer_name
       AND T2.ticket_nbr = T3.ticket_nbr
 WHERE T3.ticket_nbr IS NULL;
```

その4

答え　ディーター・ノースが送ってきたのは、「前の行の値」を計算するために、SQL-99のOLAP関数を利用する解だ。もし「カレント行の値」と前の値の差が1よりも大きければ、歯抜けがあるということになる。ただし、連番は1から始まるので、1に対する「前の値（prev_nbr）」を表すダミー値を、COALESCE関数を使って追加する必要がある。

```
SELECT buyer_name, (prev_nbr + 1) AS gap_start,
       (ticket_nbr - 1) AS gap_end
  FROM (SELECT buyer_name, ticket_nbr,
               COALESCE(MIN(ticket_nbr) OVER (
                 PARTITION BY buyer_name
                 ORDER BY ticket_nbr
                 ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING), 0)
          FROM Tickets) AS DT(buyer_name, ticket_nbr, prev_nbr)
 WHERE (ticket_nbr - prev_nbr) > 1;
```

その5

答え　オムニバズは、共通表式を使う別解も考えた。これなら、連番テーブルもDISTINCTも必要ない。

```
WITH CTE(buyer_name, ticket_nbr)
AS
(SELECT buyer_name, MAX(ticket_nbr)
   FROM Tickets
  GROUP BY buyer_name
 UNION ALL
 SELECT buyer_name, ticket_nbr - 1
   FROM CTE
  WHERE ticket_nbr - 1 >= 0)
SELECT A.buyer_name, A.ticket_nbr
  FROM CTE AS A
       LEFT OUTER JOIN
       Tickets AS B
       ON A.buyer_name = B.buyer_name
      AND A.ticket_nbr = B.ticket_nbr
 WHERE B.buyer_name IS NULL;
```

このクエリで注目してほしいのは、チケット番号の範囲を完全にカバーする連番を生成するために、再帰的な共通表式を使っている点だ。メインのSELECT文では、外部結合で差集合演算を行っている[訳注2]。

訳注2：外部結合によって、マスターテーブルに存在しない値がNULLで現れる。これをWHERE句で除外することで、差集合演算を行うことができる。この方法は、EXCEPT演算子を持っていないRDBMSで差集合演算を代用するときにも便利である。

パ|ズ|ル|
59 期間を結合する

重複する期間をまとめる

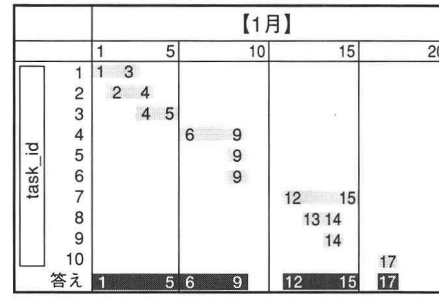
スケジュール表を扱っていると、連続していたり重なり合ったりしている期間同士をつなげる必要がしばしば生じる。ただ、これを簡単なクエリで行うのは難問である。以下に紹介するコードも、目で追って動作を理解するのは容易ではない。注意深く読んでほしい[訳注1]。

```
CREATE TABLE Timesheets
(task_id CHAR(10) NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
          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');
```

訳注1：この問題は、『プログラマのためのSQL 第2版』（ピアソン・エデュケーション刊）の「4.4.2 期間の連続」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現すれば、右のようになる。求める答えは最下行の期間である。答えすべてに共通する考え方は、「すべての開始日と終了日（ただし開始日 <= 終了日）を組み合わせて"期間"を表すペアを作り、その後にほかの期間に含まれる開始日と終了日を除外する」である。



その1

答え

```
SELECT T1.start_date, MAX(T2.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);

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'
```

その2

答え

次に示すのは長いクエリだが、実行時間をチェックしてもらいたい。

```
SELECT X.start_date, MIN(Y.end_date) AS end_date
  FROM (SELECT T1.start_date
          FROM Timesheets AS T1
               LEFT OUTER JOIN
               Timesheets AS T2
               ON T1.start_date > T2.start_date
              AND T1.start_date <= T2.end_date
```

```
          GROUP BY T1.start_date
         HAVING COUNT(T2.start_date) = 0) AS X(start_date)
         INNER JOIN
        (SELECT T3.end_date
           FROM Timesheets AS T3
                LEFT OUTER JOIN
                Timesheets AS T4
                ON T3.end_date >= T4.start_date
               AND T3.end_date < T4.end_date
          GROUP BY T3.end_date
         HAVING COUNT(T4.start_date) = 0) AS Y(end_date)
        ON X.start_date <= Y.end_date
 GROUP BY X.start_date;
```

その3

```
SELECT X.start_date, MIN(X.end_date) AS end_date
  FROM (SELECT T1.start_date, T2.end_date
          FROM Timesheets AS T1, Timesheets AS T2,
               Timesheets AS T3
         WHERE T1.end_date <= T2.end_date
         GROUP BY T1.start_date, T2.end_date
        HAVING MAX(CASE WHEN (T1.start_date > T3.start_date
                              AND T1.start_date <= T3.end_date)
                              OR
                              (T2.end_date >= T3.start_date
                              AND T2.end_date < T3.end_date)
                        THEN 1 ELSE 0 END) = 0) AS X
 GROUP BY X.start_date;
```

この小さなクエリの中にも、多くのロジックが組み込まれている。

パ|ズ|ル|
60 バーコード
手続き型から宣言型へ考え方を切り換える（その2）

最近、www.sswug.org[訳注1]の常連寄稿者が、13桁の標準的なバーコードのチェックサムを計算するT-SQL（Transact-SQL）関数を投稿してきた。使っていたアルゴリズムはシンプルな加重和である（加重和を知らない方は、拙著『Data & Databases』（Morgan Kaufmann刊, 1998）のSection 15.3.1を参照してほしい）。13桁の数字のうち最初の12桁を取り、それにある計算を施した結果が13桁目と一致するかどうかを確認する。計算の規則は単純で、

1. 偶数番目の数字を足した合計をS1
2. 奇数番目の数字を足した合計をS2

とすると、S1からS2を引き、その結果を10で割った余りを求め、最後にその絶対値をとる。公式化すれば、ABS(MOD(S1-S2), 10)となる。

以下に、投稿されたT-SQLの関数のコードを、標準のSQL/PSM[訳注2]に変換したものを示す。

```
CREATE FUNCTION Barcode_CheckSum(IN my_barcode CHAR(12))
RETURNS INTEGER
  BEGIN
  DECLARE barcode_checkres INTEGER;
  DECLARE idx INTEGER;
  DECLARE sgn INTEGER;

  SET barcode_checkres = 0;

  -- 与えられたバーコードが数値かチェックする
  IF IsNumeric(my_barcode) = 0
  THEN RETURN -1;
  END IF;

  -- バーコードの長さをチェックする
  IF CHAR_LENGTH(TRIM(BOTH ' ' FROM my_barcode)) <> 12
  THEN RETURN -2;
  END IF;
```

訳注1：SQL Serverユーザのコミュニティ「SQL Server Worldwide User Group」。

訳注2：PSMとは「Persistent Stored Modules」の略で、いわゆるストアドプロシージャの標準規格の名称。

```
  -- バーコードのチェックサムを計算する
  SET idx = 1;
  WHILE idx <= 12
  DO -- 数字の符号を求める

    IF MOD(idx, 2) = 0
       THEN SET sgn = -1;
       ELSE SET sgn = +1;
    END IF;

    SET barcode_checkres = barcode_checkres +
        CAST(SUBSTRING(my_barcode FROM idx FOR 1)
             AS INTEGER) * sgn;
    SET idx = idx + 1;
  END WHILE;

  -- チェックディジット
  RETURN ABS(MOD(barcode_checkres, 10));
END;
```

このプロシージャは、例えば次のように動作する。

```
barcode_checkSum('283723281122')
= ABS (MOD(2-8 + 3-7 + 2-3 + 2-8 + 1-1 + 2-2), 10))
= ABS (MOD(-6 -4 -1 -6 + 0 + 0), 10)
= ABS (MOD(-17, 10))
= ABS(-7) = 7
```

その1

答え

OK、話は分かった。さて、どこから手を付けよう？ まず、不必要なローカル変数を作っていることや、T-SQL固有の関数であるIsNumericを使っていることが目に付く。また、チェックディジットがバーコードから切り分けられた整数ではなく、文字として考えられていることも気になる。さらに、このコードには3つのIF文と1つのWHILEループがあり、考えられる限り最も手続き的な作法で書かれている。

念のために言っておくと、SQL/PSMは負の戻り値をエラー扱いとはしていない。だが今は、文法の説明には立ち入らないでおこう。SQL/PSMの文法はT-SQLとはかなり異なる、とだけ言っておく。

しかし、なぜこんな、どこからどう見ても手続き的なコードを書くのだろうか。このコードの大半は、宣言的な表現に置き換えることが可能だ。手始めに、ループとネスト

した関数呼び出しをお馴染みの連番補助テーブルで置き換え、IF文をなくすためにCASE式を使おう。

置き換えの大まかなガイドラインはこんな感じだ。

- 手続き的なループは連番集合になる。

```
FOR seq FROM 1 TO n DO f(seq);
➜ SELECT f(seq) FROM Sequence WHERE seq <= n;
```

- 手続き的な分岐はCASE式になる。

```
IF .. THEN .. ELSE
➜ CASE WHEN .. THEN .. ELSE .. END;
```

- 一連の割り当てと関数呼び出しは、関数呼び出しのネストになる。

```
DECLARE x <type>;
SET x = f(..);
SET y = g(x);
  ..;
➜f(g(x))
```

その2

答え

上記のガイドラインに従って置き換えた修正第1版がこれだ。

```
CREATE FUNCTION Barcode_CheckSum(IN barcode CHAR(12))
RETURNS INTEGER
BEGIN
  IF barcode SIMILAR TO '%[^0-9]%'
  THEN RETURN -1;
  ELSE RETURN
    (SELECT ABS(MOD(SUM(CAST(
            SUBSTRING(barcode FROM S.seq FOR 1) AS INTEGER)
            * CASE MOD(S.seq, 2) WHEN 0 THEN 1 ELSE -1 END),10))
       FROM Sequence AS S
      WHERE S.seq <= 12);
  END IF;
END;
```

「SIMILAR TO <正規表現>」という述語は、注目すべき“トリック”である。12文字すべてが数字であることを保証するために、「^」を付けて否定条件(“バーコードの文字列の中に0～9ではない文字が存在する”)を使っている。なお、サイズオーバーの文字列が渡された場合、パラメータに収まり切らないのでオーバーフローエラーになるが、逆に文字数が少なかった場合には空白で埋められる点には注意が必要だ。

その3

答え だが、もう少し考えてほしい。答えその2のコードはもっと改善できる。

```
CREATE FUNCTION Barcode_CheckSum(IN barcode CHAR(12))
RETURNS INTEGER
RETURN
  (SELECT ABS(MOD(SUM(CAST(
          SUBSTRING(barcode FROM S.seq FOR 1) AS INTEGER)
          * CASE MOD(S.seq, 2) WHEN 0 THEN 1 ELSE -1 END), 10))
     FROM Sequence AS S
    WHERE S.seq <= 12
      AND barcode NOT SIMILAR TO '%[^0-9]%');
```

この関数は、不正なバーコードが渡された場合にNULLを返す。この関数はこれで1つのSQL文であり、大変素晴らしい。さらにちょっとした修正を加えると、次のようになる。

```
CREATE FUNCTION Barcode_CheckSum(IN barcode CHAR(12))
RETURNS INTEGER
RETURN
  (SELECT ABS(MOD(SUM(CAST(
          SUBSTRING(barcode FROM Weights.seq FOR 1) AS INTEGER)
          * Weights.wgt), 10))
     FROM (VALUES (CAST(1 AS INTEGER), CAST(-1 AS INTEGER)),
                  (2, +1), (3, -1), (4, +1), (5, -1),
                  (6, +1), (7, -1), (8, +1), (9, -1),
                  (10, +1), (11, -1), (12, +1)
          ) AS weights(seq, wgt)
    WHERE barcode NOT SIMILAR TO '%[^0-9]%');
```

標準SQLを使ったもう1つの賢いトリックは、VALUES式でテーブル定数を作ることだ。テーブル定数の1行目では、明示的なキャストにより列のデータ型を指定している。

このSQL文では、整数型を2列指定したことになる。

その4

答え さて、最適解はどれだろう？ 実は、真の答えはここまでに挙げたいずれでもない。この練習問題のポイントは、**集合指向的**かつ**宣言的**な答えを考えてもらうことにある。

我々は、条件をチェックする関数を書いてきた。だが本当に求めるべきは、バーコードに制約を適用するCHECK句である。これまでのアプローチに代わり、次の解を試してほしい。

```
CREATE TABLE Products
(..
  barcode CHAR(13) NOT NULL
  CONSTRAINT all_numeric_checkdigit
    CHECK (barcode NOT SIMILAR TO '%[^0-9]%')
  CONSTRAINT valid_checkdigit
    CHECK ((SELECT ABS(MOD(SUM(CAST(
                   SUBSTRING(barcode FROM Weights.seq FOR 1)
                   AS INTEGER) * Weights.wgt), 10))
              FROM (VALUES (CAST(1 AS INTEGER),
                            CAST(-1 AS INTEGER)),
                            (2, +1), (3, -1), (4, +1), (5, -1),
                            (6, +1), (7, -1), (8, +1), (9, -1),
                            (10, +1), (11,-1), (12, +1)
                   ) AS weights(seq, wgt))
           = CAST(SUBSTRING(barcode FROM 13 FOR 1) AS INTEGER)),
 ...
);
```

こうすることで、スキーマから不正なデータを締め出せる。それは関数にはできないことだ。どうしても関数を使うというなら、INSERT時にトリガーを起動すれば近いことはできる。なお、上のコードで制約を2つに分けているのは、そのほうがエラーメッセージがより具体的になるからだ。これがSQLの考え方である[訳注3]。

訳注3：ただし、例によってCHECK制約の中でサブクエリを使うことは、多くの実装ではまだ不可能。

パ|ズ|ル|
61 文字列をソートする
ループを使わないでソート

トニー・ウィルトンは、2003年に面白い問題を送ってきた。今、ある決まった形式、例えば「CABBDBC」のような文字列を生成するストアドプロシージャを書いているとしよう。問題というのは、この文字列をアルファベット順にソートすることだ。つまり、「CABBDBC」という入力なら「ABBBCCD」を出力する。ただし、ライブラリ関数は使えないものとする。また、トニーの言う「決まった形式」とは、文字列は常にCHAR(7)で、{'A', 'B', 'C', 'D'}の4文字だけを使うことを意味している。

その1

答え

多くの人が最初に提案してきた解答は、RDBMS製品が独自に実装している4GLでWHILEループを使うプロシージャだった。要するに、文字列内のSUBSTRING(gen_str FROM i FOR 1)にバブルソートを適用するわけだ。そのため、1行に1度のプロシージャ呼び出しが必要になる。

もし文字列が固定長なら、ボース＝ネルソンの恐ろしく高速なソート法[訳注1]が利用できる。そのソート法では、並べ替えるべき文字列のスワップペアを生成する。ただ、そのアルゴリズムの詳細については、ここで検討したい数学的範囲を超えるので割愛する。詳しくは、私の著書『SQL for Smarties Third Edition』(Morgan Kaufmann刊, 2005)を参照してほしい。答えは、大量のUPDATE文から成るプロシージャになる。

その2

答え

もし生成する文字列が比較的小さい集合なら、参照テーブルを使う方法がある。

```
CREATE TABLE SortMeFast
(unsorted_string CHAR(7) NOT NULL PRIMARY KEY,
 sorted_string CHAR(7) NOT NULL);
```

訳注1：R.C.ボースとR.J.ネルソンが1962年に提示した、再帰手続きによって配列の高速なソートを行うアルゴリズムのこと。"A Sorting Problem"：Journal of the ACM, Vol9, pp282-96 (http://portal.acm.org/citation.cfm?id=321126&coll=portal&dl=ACM)を参照。また、著者(セルコ)はこのアルゴリズムをSQLで表現する方法を考案している。"Scrubbing Data with Non-1NF Tables - Part 1" DBAzine 2006年1月 (http://www.dbazine.com/ofinterest/oi-articles/celko19)

ファイクのアルゴリズム[訳注2]を使えば、テーブルに文字の順列をロードできる。こうすることで、行を参照するたびに関数を呼び出すのではなく、ソートされていない文字列の集合全体を一度に処理することが可能になる。

その1

答え もし、対象の文字集合が小さければ、こういうやり方もある。'A' の数を数え、その数だけ文字列を生成する。次に、'B' も数えて同様に文字列を作り、Aの文字列の後ろにつなげる。これを、残りのアルファベット（'C' と 'D'）についても繰り返す。

仮に、<文字列>をn個コピーするREPLICATE(<文字列>, n)関数と、<対象文字列>内に現れるすべての<置換前文字列>を<置換後文字列>に置き換えるREPLACE(<対象文字列>, <置換前文字列>, <置換後文字列>)関数、および<文字列>の長さを返すCHARACTER_LENGTH(<文字列>)が使えるとしよう。すると、次のような解答が考えられる。

```
BEGIN
DECLARE instring CHAR(7);
SET instring = 'DCCBABA';

REPLICATE('A', (CHARACTER_LENGTH(instring) -
  CHARACTER_LENGTH(REPLACE(instring, 'A', '')))) ||
REPLICATE('B', (CHARACTER_LENGTH(instring) -
  CHARACTER_LENGTH(REPLACE(instring, 'B', '')))) ||
REPLICATE('C', (CHARACTER_LENGTH(instring) -
  CHARACTER_LENGTH(REPLACE(instring, 'C', '')))) ||
REPLICATE('D', (CHARACTER_LENGTH(instring) -
  CHARACTER_LENGTH(REPLACE(instring, 'D', ''))))

END;
```

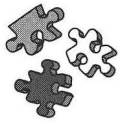
この式は、ビューに入れることもできる。そうすれば、スキーマ内に手続き的なコードは一切残らない。SQLプログラマの大部分は手続き的プログラミングで育ってきたので、こういう考え方ができないようだ。反対に、この解答をLISPプログラマに見せたときの反応は「もちろんそうするさ。ほかにどんな方法があるんだ？」というものだった。

訳注2：C.T.ファイクが1975年に提示した、順列を作成する高速なアルゴリズムのこと。"A permutation generation method"：The Computer Journal Vol.18(1), pp.21-22 (http://comjnl.oxfordjournals.org/cgi/content/abstract/18/1/21) を参照。

パズル 62

レポートの整形

結果を列数固定で表示する



SQLはデータ検索言語であって、レポートを作成するための道具ではない。残念なことに、人々はこの点に無自覚で、本来SQLのやるべき仕事ではないことまでSQLでやろうとするケースが散見される。その中でも最も多く見受けられるのが、表示用の決まった数の列数に値のリストを配置することである。

このパズルは、もともとスミス・バーニー社のリチャード・レムレーから送られてきた。彼はたびたび私のSQLパズルの解法をうまく料理してくれる人物だ。彼は、そんなことできっこないと言った友人と賭けをして、見事に勝利をおさめたという。

その問題だが、まずはnameという一意な列を1つだけ持つNamesテーブルを作り、次のようにデータを入れる。

```
CREATE TABLE Names
(name VARCHAR(15) NOT NULL PRIMARY KEY);
```

```
INSERT INTO Names
VALUES ('Al'), ('Ben'), ('Charlie'),
       ('David'), ('Ed'), ('Frank'),
       ('Greg'), ('Howard'), ('Ida'),
       ('Joe'), ('Ken'), ('Larry'),
       ('Mike'), ('Neal');
```

単純に「SELECT name FROM Names ORDER BY name;」を実行すれば、アルファベット順になった名前のリストが得られる。では、この結果を次のような3列1行の形式で表示するには、どうすればよいだろうか。

```
name1     name2      name3
============================
'Al'      'Ben'      'Charlie'
'David'   'Ed'       'Frank'
'Greg'    'Howard'   'Ida'
'Joe'     'Ken'      'Larry'
'Mike'    'Neal'     NULL
```

あるいは、4列1行ならこうなる。

```
name1     name2     name3       name4
=====================================
'Al'      'Ben'     'Charlie'   'David'
'Ed'      'Frank'   'Greg'      'Howard'
'Ida'     'Joe'     'Ken'       'Larry'
'Mike'    'Neal'    NULL        NULL
```

最終的には、任意の列数に対応できるところまで拡張したい。あなたはこうした結果を返すSQLを、たった1文で書けるだろうか？

その1

答え

ここは、簡単な2列1行の解法から考え始めるのが一番よい。

```
SELECT N1.name AS name1, MIN(N2.name) AS name2
  FROM Names AS N1
       LEFT OUTER JOIN
       Names AS N2
       ON N1.name < N2.name
 WHERE N1.name
          IN (SELECT A.name
                FROM Names AS A
                     INNER JOIN
                     Names AS B
                     ON A.name <= B.name
              GROUP BY A.name
              HAVING MOD(COUNT(B.name), 2) =
                     (SELECT MOD(COUNT(*), 2) FROM Names))
 GROUP BY N1.name
 ORDER BY N1.name;
```

自己外部結合（2～5行目）では、まず2列のうち、アルファベット順でより早い名前を第1列（name1）に入れ、その後MIN関数で残っている名前からN1.name（第1列に入れた名前）以降で最も順序の早い名前（いわば「第2位」の名前）を取り出している。

だが本当の“トリック”は、WHERE句にある。結果集合を定義するためには、求めるべき結果の各行のN1.nameの値を見つけ出し、その名前のリストを使いたい。このケースでは、最初の名前（'Al'）、3番目の名前（'Charlie'）などである。これを実現しているのがMOD関数である。この関数は公式にはSQL-92の一部ではないので、技術的には整数の算術で書くのが正しい。しかし、実際にはほとんどのRDBMSが持っている関数

であるし、SQL-99で標準に入ったので、使ってもかまわないだろう。

それでは、(WHERE句中にある)実験的な結果テーブルから見てみよう。

```
SELECT A.name
  FROM Names AS A
       INNER JOIN
       Names AS B
       ON A.name <= B.name
 GROUP BY A.name;
```

4つの名前について見ると、非グループ化した結果は次のようになる。

```
A.name      B.name
=====================
'Al'        'Al'
'Al'        'Ben'
'Al'        'Charlie'
'Al'        'David'
---------------------
'Ben'       'Ben'
'Ben'       'Charlie'
'Ben'       'David'
---------------------
'Charlie'   'Charlie'
'Charlie'   'David'
---------------------
'David'     'David'
```

これを、(MOD(COUNT(A.name), 2))= 0という述語によってふるいにかけると、欲しい行だけを取り出せる。この方法は、行数が偶数のときにはうまくいくのだが、奇数のとき(例えば'Oliver'をINSERTした場合)は、その「はぐれ者」の行も結果のテーブルに入れる必要が生じる。これは、元テーブルの行数を求めて、最終的な結果テーブルの第1列の選択を調節すれば可能である。少し詳細は省くが、簡単にできるはずだ。

さて、次は3列や4列の場合を飛ばし、一気に5列の場合を考えて、解法がどういう風に一般化されるかを見てみよう。

```
SELECT N1.name,
       MIN(N2.name) AS name2,
       MIN(N3.name) AS name3,
       MIN(N4.name) AS name4,
```

```
       MIN(N5.name) AS name5
  FROM (Names AS N1
        LEFT OUTER JOIN
        Names AS N2
        ON N1.name < N2.name)
          LEFT OUTER JOIN
          Names AS N3
          ON N1.name < N2.name
             AND N2.name < N3.name
            LEFT OUTER JOIN
            Names AS N4
            ON N1.name < N2.name
               AND N2.name < N3.name
               AND N3.name < N4.name
              LEFT OUTER JOIN
              Names AS N5
              ON N1.name < N2.name
                 AND N2.name < N3.name
                 AND N3.name < N4.name
                 AND N4.name < N5.name
 WHERE N1.name IN (SELECT A.name
                     FROM  Names AS A
                           INNER JOIN
                           Names AS B
                           ON A.name <= B.name
                    GROUP BY A.name
                   HAVING MOD(COUNT(B.name), 5) =
                          (SELECT MOD(COUNT(*), 5)
                             FROM Names))
 GROUP BY N1.name;
```

その2

答え 答えその1のクエリを短くしたバージョンとして、次のような解がある。

```
SELECT N3.name, MIN(N4.name), MIN(N5.name), MIN(N6.name),
       MIN(N7.name)
  FROM (SELECT N1.name
          FROM Names AS N1
               INNER JOIN
               Names AS N2
               ON N1.name >= N2.name
         GROUP BY N1.name
        HAVING MOD(COUNT(*), 5) = 1) AS N3(name)
```

```
        LEFT OUTER JOIN
        Names AS N4
        ON N3.name < N4.name
          LEFT OUTER JOIN
          Names AS N5
          ON N4.name < N5.name
            LEFT OUTER JOIN
            Names AS N6
            ON N5.name < N6.name
              LEFT OUTER JOIN
              Names AS N7
              ON N6.name < N7.name
GROUP BY N3.name;
```

その3

答え このパズルが『DBMS』誌1997年2月号に掲載された後、ナヤン・ラヴァルが私に電子メールを送ってきた。彼は、答えその1のLEFT OUTER JOIN句で「N1.name < N2.name」が繰り返し使われている点に着目し、この条件が実は最初の1度しか必要ないことに気づいた。次のクエリは、私のシステムでは同じ結果になる。

```
-- これより上は答えその1と同じ
FROM (Names AS N1
      LEFT OUTER JOIN
      Names AS N2
      ON N1.name < N2.name)
    LEFT OUTER JOIN
    Names AS N3
    ON N2.name < N3.name
      LEFT OUTER JOIN
      Names AS N4
      ON N3.name < N4.name
        LEFT OUTER JOIN
        Names AS N5
        ON N4.name < N5.name
-- これより下も答えその1と同じ
```

その4

答え 同じような方向性で、ボスニア・ヘルツェゴヴィナのジャヴィド・ダウベゴヴィッチは答えその1の、

```
...
 WHERE N1.name IN (SELECT A.name
                     FROM  Names AS A
                           INNER JOIN
                           Names AS B
                           ON A.name <= B.name
                    GROUP BY A.name
                   HAVING MOD(COUNT(B.name), 5) =
                          (SELECT MOD(COUNT(*), 5)
                             FROM Names))
```

という部分を、

```
...
 WHERE N1.name IN (SELECT A.name
                     FROM  Names AS A
                           INNER JOIN
                           Names AS B
                           ON A.name >= B.name
                    GROUP BY A.name
                   HAVING MOD(COUNT(B.name), 5) = 1)
```

と書き換えたほうがよいと意見を述べた。

その5

答え

ドミトリー・シツィンツェフは、また異なる解を考え出した。ここでは、N＝5の場合の解法を示す。これは、リチャード・レムレーの解法とは大きく異なり、5回の自己結合を使わずに済んでいる。

```
SELECT MAX(name1), MAX(name2), MAX(name3), MAX(name4), MAX(name5)
  FROM ( -- 巨大な副問い合わせの始まり
         SELECT (COUNT(*) - 1) / 5,
                (SELECT MAX(N1.name)
                   FROM Names AS N3
                  WHERE N1.name <= N3.name
                 HAVING MOD(COUNT(*), 5)
                        = (SELECT MOD(COUNT(*), 5)
                             FROM Names)),
                (SELECT MAX(N1.name)
                   FROM Names AS N3
```

```
                  WHERE N1.name <= N3.name
                 HAVING MOD(COUNT(*), 5)
                        = (SELECT MOD((COUNT(*) - 1), 5)
                             FROM Names)),
                (SELECT MAX(N1.name)
                   FROM Names AS N3
                  WHERE N1.name <= N3.name
                 HAVING MOD(COUNT(*), 5)
                        = (SELECT MOD((COUNT(*) - 2), 5)
                             FROM Names)),
                (SELECT MAX(N1.name)
                   FROM Names AS N3
                  WHERE N1.name <= N3.name
                 HAVING MOD(COUNT(*), 5)
                        = (SELECT MOD((COUNT(*) - 3), 5)
                             FROM Names)),
                (SELECT MAX(N1.name)
                   FROM Names AS N3
                  WHERE N1.name <= N3.name
                 HAVING MOD(COUNT(*), 5)
                        = (SELECT MOD((COUNT(*) - 4), 5)
                             FROM Names))
            FROM Names AS N1
                 INNER JOIN
                 Names AS N2
                 ON N1.name >= N2.name
           GROUP BY N1.name
       ) AS X0(cnt, name1, name2, name3, name4, name5)
GROUP BY cnt;
```

その6

答え 2000年3月12日に、私は次に示す解法をレムレーに電子メールで教えた。彼はこの解法をそのとき初めて知ったが、すぐにアレンジしてきた。

```
--3列の場合
SELECT FirstCol.name AS name1,
       MAX(CASE WHEN OtherCols.cnt = 2
                THEN OtherCols.final_name
                ELSE NULL END) AS name2,
       MAX(CASE WHEN OtherCols.cnt = 3
                THEN OtherCols.final_name
                ELSE NULL END) AS name3
```

```
  FROM (SELECT N1.name
          FROM Names AS N1, Names AS N2
         WHERE N1.name >= N2.name
         GROUP BY N1.name
        HAVING MOD(COUNT(*), 3) = 1) AS FirstCol(name)
        LEFT OUTER JOIN
        (SELECT N3.name, N5.name, COUNT(*)
           FROM Names AS N3, Names AS N4, Names AS N5
          WHERE N3.name < N5.name
            AND N4.name BETWEEN N3.name AND N5.name
          GROUP BY N3.name, N5.name)
          AS OtherCols(name,final_name, cnt)
        ON FirstCol.name = OtherCols.name
 GROUP BY FirstCol.name
 ORDER BY FirstCol.name;
```

```
--5列の場合
SELECT FirstCol.name AS name1,
       MAX(CASE WHEN OtherCols.cnt = 2
                THEN OtherCols.final_name
                ELSE NULL END) AS name2,
       MAX(CASE WHEN OtherCols.cnt = 3
                THEN OtherCols.final_name
                ELSE NULL END) AS name3,
       MAX(CASE WHEN OtherCols.cnt = 4
                THEN OtherCols.final_name
                ELSE NULL END) AS name4,
       MAX(CASE WHEN OtherCols.cnt = 5
                THEN OtherCols.final_name
                ELSE NULL END) AS name5
  FROM (SELECT N1.name
          FROM Names AS N1, Names AS N2
         WHERE N1.name >= N2.name
         GROUP BY N1.name
        HAVING MOD(COUNT(*), 5) = 1) AS FirstCol(name)
        LEFT OUTER JOIN
        (SELECT N3.name, N5.name, COUNT(*)
           FROM Names AS N3, Names AS N4, Names AS N5
          WHERE N3.name < N5.name
            AND N4.name BETWEEN N3.name AND N5.name
          GROUP BY N3.name, N5.name)
          AS OtherCols(name,final_name, cnt)
        ON FirstCol.name = OtherCols.name
GROUP BY FirstCol.name
ORDER BY FirstCol.name;
```

```
--6列の場合
SELECT FirstCol.name AS name1,
       MAX(CASE WHEN OtherCols.cnt = 2
                THEN OtherCols.final_name
                ELSE NULL END) AS name2,
       MAX(CASE WHEN OtherCols.cnt = 3
                THEN OtherCols.final_name
                ELSE NULL END) AS name3,
       MAX(CASE WHEN OtherCols.cnt = 4
                THEN OtherCols.final_name
                ELSE NULL END) AS name4,
       MAX(CASE WHEN OtherCols.cnt = 5
                THEN OtherCols.final_name
                ELSE NULL END) AS name5,
       MAX(CASE WHEN OtherCols.cnt = 6
                THEN OtherCols.final_name
                ELSE NULL END) AS name6
  FROM (SELECT N1.name
          FROM Names AS N1, Names AS N2
         WHERE N1.name >= N2.name
         GROUP BY N1.name
        HAVING MOD(COUNT(*), 6) = 1) AS FirstCol
        LEFT OUTER JOIN
        (SELECT N3.name, N5.name AS final_name, COUNT(*) AS cnt
           FROM Names AS N3, Names AS N4, Names AS N5
          WHERE N3.name < N5.name
            AND N4.name BETWEEN N3.name AND N5.name
          GROUP BY N3.name, N5.name) AS OtherCols
        ON FirstCol.name = OtherCols.name
 GROUP BY FirstCol.name
 ORDER BY FirstCol.name;
```

この方法だと、列が増えた場合にも、SELECT句のリストの列を増やし、MOD関数の剰余の数を変えるだけで対応できる。クエリの残りの部分は、まったく変える必要がない。

パ|ズ|ル|
63

連続的なグループピング

結合条件でサブクエリを実行する

ドナルド・ハロランは、一見すると簡単そうな問題を出してきた。

```
CREATE TABLE T
(num INTEGER NOT NULL PRIMARY KEY,
 data CHAR(1) NOT NULL);
```

```
INSERT INTO T
VALUES (1, 'a'),
       (2, 'a'),
       (3, 'b'),
       (6, 'b'),
       (8, 'a');
```

問題とは、各data値がnumの何番から何番まで連続しているのかを、その出現順で集約するというものである。上のサンプルを例にとると、求める結果は次のようになる。

```
low   high    data
==================
  1      2    'a'
  3      6    'b'
  8      8    'a'
```

以下にハロランの解法を示すが、彼自身、どうも冗長なコードを書いているのではないかと感じていた。というのも、アルゴリズムは2つのステップから構成されているように見えるのだが、SQLでは3つのステップを踏んでいるような気がしたからだ。そのアルゴリズムとは、次のとおりである。

1. 各行(r)について、R.num > r.numかつR.data <> r.dataとなる最初の行(R)を求める
2. (R)で(r)をグループ化する

その1

答え ハロランが最初に考えたクエリはこうだ。

```
SELECT MIN(T1.num) AS low,
       MAX(T1.num) AS high,
       T1.data
  FROM T AS T1
       LEFT OUTER JOIN
       T AS T2
       ON T2.num
          = (SELECT MIN(num)
               FROM T
              WHERE num > T1.num
                AND data <> T1.data)
 GROUP BY T1.data, T2.num;
```

その2

答え 私が別バージョンの方法として考えたのは、範囲の中身をチェックするためにALL述語を使うものである。

```
SELECT X.data, MIN(X.low) AS low, X.high
  FROM (SELECT T1.data, T1.num, MAX(T2.num)
          FROM T AS T1, T AS T2
         WHERE T1.num <= T2.num
           AND T1.data
               = ALL(SELECT T3.data
                       FROM T AS T3
                      WHERE T3.num BETWEEN T1.num AND T2.num)
         GROUP BY T1.data, T1.num) AS X(data, low, high)
 GROUP BY X.data, X.high;
```

率直に言って、自分でもこの方法がオリジナルより優れているとは思わない。

その3

答え　スティーブ・カシュは、次のような解法を示してきた。カシュは、オリジナルより速いかどうかは分からないと言っていたが、この方法なら別のアプローチによる高速化が可能だ（(num, data) に対するクラスタ索引が使える）。

```
SELECT MIN(num) AS low, MAX(num) AS high, data
  FROM (SELECT A.num,
               SUM(CASE WHEN A.data = B.data
                        THEN 1
                        ELSE 0 END)
               - COUNT(B.num) AS ct,
               A.data
          FROM T AS A, T AS B
         WHERE A.num >= B.num
         GROUP BY A.num, A.data) AS A (num, ct, data)
GROUP BY data, ct;
```

1つしかデータの値を持たない範囲を調べるために、ちょっとした計算を行っているのもこの解法のポイントだ。

パ|ズ|ル|
64 ボックス
多次元の重複範囲を見つける

この面白い問題を教えてくれたのは、ミキト・ハラキリという人物である。あるデカルト空間（座標空間）を想像してほしい。そして、その中にn次元のボックスが何個か含まれているとする。各ボックスは、次のようなテーブルで定義される。

```
CREATE TABLE Boxes
(box_id CHAR(1) NOT NULL,
 dim    CHAR(1) NOT NULL,
 PRIMARY KEY (box_id, dim),
 low INTEGER NOT NULL,
 high INTEGER NOT NULL);
```

Boxes：ボックステーブル	box_id：ボックスID	dim：次元
low：低位	high：高位	

問題は、「互いに交わるボックス（3次元の直方体など）の組み合わせをすべて見つける」というものだ。

```
A = {(x,0,2),(y,0,2),(z,0,2)}
B = {(x,1,3),(y,1,3),(z,1,3)}
C = {(x,10,12),(y,0,4),(z,0,100)}
```

例えば、上の3つのボックス[訳注1]について見ると、ボックスAとBは交わるが、CはAともBとも交わらない。さらに、問題が解けたらボーナスポイントも付けよう。この問題を解くクエリに何か変わった特徴はないだろうか？ それも見つけてほしい。

その1

答え

ボブ・バドゥアールから寄せられた解答はこうだ。

```
SELECT B1.box_id AS box1, B2.box_id AS box2
  FROM Boxes AS B1, Boxes AS B2
```

訳注1：A、B、Cはbox_idを、(x,0,2)などの各項目はそれぞれdim、low、highの値を表しており、(x,0,2)は「x軸の座標0から座標2まで」を意味する。

```
WHERE B1.box_id < B2.box_id
  AND NOT EXISTS
      (SELECT *
         FROM Boxes AS B3, Boxes AS B4
        WHERE B3.box_id = B1.box_id
          AND B4.box_id = B2.box_id
          AND B4.dim = B3.dim
          AND (B4.high < B3.low OR B4.low > B3.high))
GROUP BY B1.box_id, B2.box_id;
```

出題者のミキト・ハラキリは、この解が関係除算の一般化になっている点に興味を持った。SQLにおける関係除算は、2段階にネストしたサブクエリや、EXCEPT演算子を使ったネストのないサブクエリ、あるいはHAVING句の中でCOUNT関数を使うクエリとして表現できた[訳注2]。だが、ボブのクエリはそのいずれとも違っている。HAVING句を使って書けば、さしずめこうなるだろう。

```
SELECT B1.box_id AS box1, B2.box_id AS box2
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.low BETWEEN B2.low AND B2.high
   AND B1.dim = B2.dim
   AND B1.box_id <> B2.box_id
 GROUP BY B1.box_id, B2.box_id
HAVING COUNT(*)
         = (SELECT COUNT(*)
              FROM Boxes AS B3
             WHERE B3.box_id = B1.box_id);
```

その2

答え　異なるアプローチも試してみよう。まずは1次元の、より強固なDDL文からだ。

```
CREATE TABLE Boxes
(box_id CHAR(1) NOT NULL,
 dim CHAR(1) NOT NULL,
 PRIMARY KEY (box_id, dim),
 low INTEGER NOT NULL,
 high INTEGER NOT NULL,
 CHECK (low < high));
```

訳注2：関係除算については、パズル21を参照。

```
INSERT INTO Boxes VALUES ('A', 'x', 0, 2);
INSERT INTO Boxes VALUES ('B', 'x', 1, 3);
INSERT INTO Boxes VALUES ('C', 'x', 10, 12);
```

```
--1次元の場合
SELECT B1.box_id, B2.box_id
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.box_id < B2.box_id
   AND (   (B1.high BETWEEN B2.low and B2.high)
        OR (B2.high BETWEEN B1.low and B1.high));
```

このクエリの意味は、「2つの線分をつなげたときの長さが、両方の線分の長さを足した値より小さければ、線分は重複している」である。BETWEENの代用のような計算だと思ってほしい。線分A＝{(x, 0, 2)}とB＝{(x, 1, 3)}は重複するが、線分C＝{(x, 10, 12)}はどの線分とも重ならない[訳注3]。

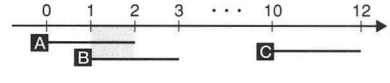
では次に、問題を2次元へ拡張しよう[訳注4]。

```
INSERT INTO Boxes VALUES ('A', 'y', 0, 2);
INSERT INTO Boxes VALUES ('B', 'y', 1, 3);
INSERT INTO Boxes VALUES ('C', 'y', 0, 4);
```

```
-- 2次元の場合：第1案
SELECT B1.box_id, B2.box_id
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.box_id < B2.box_id
   AND B1.dim = B2.dim
   AND (   (B1.high BETWEEN B2.low and B2.high)
        OR (B2.high BETWEEN B1.low and B1.high));

B1.box_id   B2.box_id
=====================
'A'         'B'         ←dim列 = 'x'
'A'         'B'         ←dim列 = 'y'
'A'         'C'         ←dim列 = 'y'
'B'         'C'         ←dim列 = 'y'
```

訳注3：この様子を図で表現してみると、右図のようになる。



訳注4：SELECT文の実行結果は訳者が追加。

これだと、y座標だけに重なる範囲のある (A, C) や (B, C) も選択されてしまう。しかし、今度はx座標とy座標のどちらにも重なる範囲が形成する共通領域 (x, y) を探さなければならない[訳注5]。それには次のように、集約結果の行数が2行存在するbox_idの組み合わせを取得すればよい。

```
-- 2次元の場合：第2案
SELECT B1.box_id, B2.box_id
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.box_id < B2.box_id
   AND B1.dim = B2.dim
   AND (   (B1.high BETWEEN B2.low and B2.high)
        OR (B2.high BETWEEN B1.low and B1.high));
 GROUP BY B1.box_id, B2.box_id
HAVING COUNT(B1.dim) = 2;
```

最後に、3次元の場合へ進もう。

```
--3次元の場合
INSERT INTO Boxes VALUES ('A', 'z', 0, 2);
INSERT INTO Boxes VALUES ('B', 'z', 1, 3);
INSERT INTO Boxes VALUES ('C', 'z', 0, 100);
```

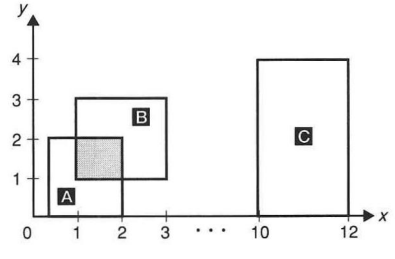
もうお分かりだろうが、HAVING句を次のように変える。

```
COUNT(B1.dim) = 3
```

もし対象となっている空間の次元数が不明なら、HAVING句を次のように変えることで一般化できる。

```
COUNT(B1.dim) = (SELECT COUNT(DISTINCT dim) FROM Boxes)
```

訳注5：この様子を図で表現してみると、右図のようになる。



パ|ズ|ル|
65

製品の対象年齢の範囲

範囲の合算と網羅性チェック

デヴィッド・プールは、彼にとって“シンプル”だという問題に取り組んでいた。それは自社製品に対し、その対象年齢によって値段を分析するというものだった。

```
CREATE TABLE PriceByAge
(product_id CHAR(10) NOT NULL,
 low_age INTEGER NOT NULL,
 high_age INTEGER NOT NULL,
 CHECK (low_age < high_age),
 product_price DECIMAL(12,4) NOT NULL,
 PRIMARY KEY (product_id, low_age));
```

PriceByAge：購入者の対象年齢テーブル	product_id：製品ID	low_age：最低年齢
high_age：最高年齢	product_price：価格	

```
INSERT INTO PriceByAge VALUES ('Product1', 5, 15, 20.00);
INSERT INTO PriceByAge VALUES ('Product1', 16, 60, 18.00);
INSERT INTO PriceByAge VALUES ('Product1', 65, 150, 17.00);
INSERT INTO PriceByAge VALUES ('Product2', 1, 5, 20.00);
...
```

そのために彼がまず求めたかったのは、全年齢に適合する製品のリストである。

その1

答え

まず、「すべての年齢」を定義しよう。おそらくこれは、1歳から世界の最高年齢まで——例えば、150歳——を指す。存命中の最高齢者と信じられているドミニカのエリザベス・イスラエル（マ・パンポの名前でも知られる）が1875年1月27日生まれだというから、これくらい範囲をとっておけば十分に違いない。

答えは、連番補助テーブル[訳注1]を使えばあっという間に得られる[訳注2]。

訳注1：連番補助テーブルの詳細は、パズル48を参照。

訳注2：この問題の面白いのは、1つのレコードでは年齢範囲がカバーできなくても、複数のレコードを合わせてカバーできたのなら「合わせ技一本」と見なすことだ。

```
SELECT P.product_id
  FROM PriceByAge AS P, Sequence AS S
 WHERE S.seq BETWEEN P.low_age AND P.high_age
   AND S.seq <= 150
 GROUP BY P.product_id
HAVING COUNT(seq) = 150;
```

その2

答え また、次のような連番補助テーブルを使わない簡潔な別解も考えられる。

```
SELECT product_id
  FROM PriceByAge
 GROUP BY product_id
HAVING SUM(high_age - low_age + 1) = 150;
```

パ|ズ|ル|
66 数独パズル
2次元配列を扱う

私はかねてから、最近流行のパズル「数独」は、SQLプログラミングには格好の問題だと考えていた。数独パズルでは、まず9×9のマス目を3×3マスずつ9個のブロックに分割する。各マスには1～9の数字を1つずつ入れていくのだが、マスのいくつかには最初から数字が入っている。最終的に、9×9マスの縦・横の各列、および3×3のブロック内に同じ数字が入らないようにして全マスを埋めることができたら、パズルが解けたことになる。

妙なことに、数独パズルは1979年に米国で生まれ、1986年に日本で人気が出て、2005年に国際的な流行を見た。現在では、ほとんどの新聞が毎日のようにこのパズルを載せている。

さて、どうしたらこのパズルをSQLで解くことができるだろうか？ まずは、パズルのマス目を配列 (i, j) として設計することから始めるのがよいだろう。多くの人が最初に試す方法では、エリアの情報を列として持っていない。また、数独パズルでは3×3のブロックに名前がないので、これに名前を与える方法を考える必要もある。

```
CREATE TABLE SudokuGrid
(i INTEGER NOT NULL
  CHECK (i BETWEEN 1 AND 9),
 j INTEGER NOT NULL
  CHECK (j BETWEEN 1 AND 9),
 val INTEGER NOT NULL
  CHECK (val BETWEEN 1 AND 9),
 region_nbr INTEGER NOT NULL,
 PRIMARY KEY (i, j, val));
```

SudokuGrid：数独テーブル	i：マス目の列	j：マス目の行
val：1～9の数字	region_nbr：エリア番号	

それではこのテーブルを埋めていこう。各 (i, j) マスでは、1から9まですべての数字が必要だ。そこで、1～9の数字を持つDigitsテーブルを作り、それをSudokuGridテーブルとクロス結合してマスごとに1～9の数字を1行ずつ挿入する。しかし、ブロックの番号はどうやって入れよう？

ブロックの命名方法として分かりやすいのは、その位置を (x, y) 座標で表すことだろう。x={1, 2, 3}、y={1, 2, 3}とすると、xを10の位、yを1の位の数字に使うことで、

{11, 12, 13, 21, 22, 23, 31, 32, 33}という9つのブロックの識別番号が得られる。これにはちょっとした整数の計算が必要だが、さほど難しくはない。

```
INSERT INTO SudokuGrid (i, j, val, region_nbr)
SELECT D1.d, D2.d, D3.d,
       10 * ((D1.d + 2) / 3) + ((D2.d + 2) / 3) AS region_nbr
  FROM Digits AS D1
       CROSS JOIN Digits AS D2
                  CROSS JOIN Digits AS D3;
```

次に、既知の値をINSERTする方法と、縦や横、ブロックの値をクリアする方法が必要になる。既知の値を少なくすればするほど、このパズルの唯一の解である81マス分の行を持ったテーブル（つまり正解）を得られる期待が高くなる。

その1

答え

第1案としてありがちなのは、「縦、横、ブロックに対応した3つのDELETE文を書く」というものだろう。

```
BEGIN
DELETE FROM SudokuGrid -- 横
WHERE :my_i = i
  AND :my_j <> j
  AND :my_val = val;

DELETE FROM SudokuGrid -- 縦
WHERE :my_i <> i
  AND :my_j = j
  AND :my_val = val;

DELETE FROM SudokuGrid -- エリア
WHERE i <> :my_i
  AND j <> :my_j
  AND region_nbr = 10 * ((:my_i + 2) / 3)
                   + ((:my_j + 2) / 3)
  AND :my_val = val);
END;
```

その2

答え だが、答えその1は実行時間を無駄にしている。1つのSQL文でできることを、あえて3つに分けておく理由はない。ここは強制的にコードを結合しよう。

```
DELETE FROM SudokuGrid
WHERE (((:my_i = i AND j <> :my_j)
        OR (:my_i <> i AND j = :my_j))
       AND :my_val = val)
   OR (i <> :my_i
       AND j <> :my_j
       AND region_nbr = 10 * ((:my_i + 2) / 3)
                          + ((:my_j + 2) / 3)
       AND :my_val = val);
```

しかし、このネストしたORは、はっきり言ってみにくい！ しかも、:my_val = valという式が2度現れている。落ち着いてよく考えてみよう。(i, j)のペアは、入力された引数に対して相互に排他的な4通りに分類される。その入力によって、値をマスに残す('Keep')かクリアする('Delete')かが決定される。これは、ANDとORをネストする代わりに、CASE式を使って書くことができる。

```
DELETE FROM SudokuGrid
 WHERE CASE WHEN :my_i = i AND :my_j = j    -- 私たちの入力
            THEN 'Keep'
            WHEN :my_i = i AND :my_j <> j   -- 横
            THEN 'Delete'
            WHEN :my_i <> i AND :my_j = j   -- 縦
            THEN 'Delete'
            WHEN i <> :my_i AND j <> :my_j -- エリア
                 AND region_nbr = 10 * (:my_i + 2) / 3)
                                  + (:my_j + 2) / 3)
            THEN 'Delete'
            ELSE NULL END = 'Delete'
   AND :my_val = val);
```

その3

答え 残念ながら、答えその2をテストしてみると、うまくいかないことが分かる。すでに数字が分かっているマスには特別な注意が必要だ —— つまりマスには数字が入っているものとこれから入るものの2種類あるということだ。それを反映したのが次のクエリである。

```
DELETE FROM SudokuGrid
 WHERE CASE WHEN :my_i = i AND :my_j = j AND my_val = val
            THEN 'Keep'
            WHEN :my_i = i AND :my_j = j AND my_val <> val
            THEN 'Delete'
            WHEN :my_i = i AND :my_j <> j  -- 横
            THEN 'Delete'
            WHEN :my_i <> i AND :my_j = j  -- 縦
            THEN 'Delete'
            WHEN i <> :my_i AND j <> :my_j -- エリア
                 AND region_nbr = 10 * (:my_i + 2) / 3)
                                  + (:my_j + 2) / 3)
            THEN 'Delete'
            ELSE NULL END = 'Delete'
   AND :my_val = val);
```

解法をさらに改良するなら、パズルの履歴を残すために、すでに数字が分かっているマスを専用のテーブルに入れることが考えられるだろう。だがこれは、皆さんへの課題として残しておこう。

パ|ズ|ル|
67 安定な結婚

手続き型言語と宣言型言語の違いを知る

手続き型言語の授業で必ず習う古典的な問題を考えてみよう。準備は至って簡単だ。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・彼女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？ 手短かに言えば、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るとすると、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、男性に好みの順位付けを行っている。

したがって、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良かった」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。

例えば、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、私たち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。

この安定な結婚の問題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、2つ以上の解があるということである。この問題をSQLで解くのは、本当に難しい。SQLではバックトラックができないからだ。

安定な結婚とは、必ずしも幸福な結婚を意味しない、ということも覚えておいてほしい。実際、この問題においては、どんな集合にも安定な結婚の組み合わせが最低1つは存在するが、大抵の場合、複数の異なる組み合わせ方が見つけられる。それぞれの結婚の集合は、男性か女性どちらかの幸福を最大化しようとする傾向がある。

それでは、4組のカップルを使った小規模なサンプルから考え始めよう。

```
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：夫候補の名前　　woman：妻候補の名前
ranking：順位

```
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);
```

```
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：妻候補の名前　　man：夫候補の名前
ranking：順位

```
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);
```

このサンプルデータにおいては、

```
('Abe', 'Lynn')
('Bob', 'Joan')
('Chuck', 'Molly')
('Dave', 'Kathy')
```

という組み合わせ方は正しくない。なぜなら、('Abe', 'Joan') という「ブロッキングペア[訳注1]」が存在するからである。

```
Wives ('Joan', 'Abe', 1)
Husbands ('Abe', 'Joan', 1)
```

という行から、AbeはJoanの夫候補第1位で、JoanはAbeの妻候補第1位であることが分かる。だが、彼らはほかの人々と組み合わせられている。これを踏まえて上の組み合わせを単純に入れ替えるだけで、次のような安定状態が得られる。

```
('Abe', 'Joan')
('Bob', 'Lynn')
('Chuck', 'Molly')
('Dave', 'Kathy')
```

訳注1：ここでは、夫候補と妻候補が相思相愛、つまり「互いが互いに満足」しているペアのこと。データの組み合わせを決定付けるペア。

バックトラッキングを使えば、すべての可能な結婚の集合をわざわざ作る必要はない。ひとたびブロッキングペアが見つかれば、再度それを作る必要もない。それに対してSQLは、まず膨大なすべての組み合わせを作り、それに対してフィルタをかけねばならない。速度はバックトラッキングのほうがかなり高速だ。

SQLの唯一の利点は、この問題を解くほかのアルゴリズムは、普通、最初の解を見つけた時点でストップするのに対し、SQLはすべての組み合わせを網羅することである。だがそれも、利点と言うには些細なものだ。

その1

答え

ここで挙げる解は、リチャード・レムレーが送ってくれた。簡単に説明すると、このクエリでは「すべての可能な結婚を作った後に、条件を満たさない組み合わせを除去する」という方法がとられている。だが、ちょっとした最適化のための“トリック”も含まれている。

私が『DBMS』誌の1998年6月号（http://www.dbmsmag.com/9806d06.html）に書いたコラム[訳注2]を見てもらえば、レムレーの解でも同様に、補助テーブルを使って順列を生成するという巧みな方法を使っていることが分かるだろう。

```
CREATE TABLE Wife_Perms
(wife_name VARCHAR(5) NOT NULL PRIMARY KEY,
 perm INTEGER NOT NULL);
```

Wife_Perms：「妻の順列」補助テーブル　　wife_name：妻の名前
perm：重み付けビット

```
INSERT INTO Wife_Perms
VALUES ('Joan', 1), ('Kathy', 2), ('Lynn', 4), ('Molly', 8);
```

クエリは次のようなものだ。

```
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
```

訳注2：当該コラムに登場するパズルは、本書のパズル40とほぼ同じ内容である。

```
 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));
```

最初の述語は、特定の夫候補に対する妻候補のすべての順列を生成している。そしてEXISTS述語で、当該の行におけるブロッキングペアの有無を検査している。このクエリは、特に性能の低いマシン上では、少し実行時間がかかる。順列のトリックを使っているため、nの値が大きくなりすぎると、とうてい処理しきれなくなるだろう。

このコードを最適化するために考えられるトリックは、結合した文字列のリストを、作られたばかりの行にブロッキングペアがあるかどうか調べるのに使っていることである。このトリックはもっと短く書くこともできる。

```
SELECT *
  FROM Foo as F1, Bar as B1
 WHERE F1.city = B1.city
   AND F1.state = B1.state;
```

のようなコードを、

```
SELECT *
  FROM Foo as F1, Bar as B1
 WHERE F1.city || F1.state = B1.city || B1.state;
```

と書き換えればよい。あるいは、SQL-92のフルレベルに対応しているRDBMSなら、

```
SELECT *
  FROM Foo as F1, Bar as B1
 WHERE (F1.city, F1.state) = (B1.city, B1.state);
```

と書くこともできる。こう書いたほうが、実行速度もアップする。

私は、このような結合した名前は、それ自身を不可分で“原子的”な単位として扱うのが合理的だと考える。なぜなら、このような名前を分解すると、持ち合わせている独自の意味が破壊されてしまうからだ。その意味では、(緯度, 経度) のような組み合わせも原子的な単位を成していると言えよう。

このサンプルでは、結婚の集合は全体で4! (= 24) 個しかなかったので、低性能なマシンでもそこそこ速く動くはずだ。そこで今度は、問題を n = 8に拡張してみよう。すると、可能な結婚の集合は8! (= 4万320) に膨れ上がる。だが、最終的に求める答えの集合に含まれる行数は、非常に少ないのだ。

その2

答え

以下に、n = 8の場合の安定な結婚の問題を考えるためのコードを示す。この例は、ニクラウス・ヴィルトの著書『アルゴリズム＋データ構造＝プログラム』から取った。この問題については、ドナルド・クヌースの書いた小冊子も参考になる (最後の参考文献を参照)。

```
CREATE TABLE Husbands
(man CHAR(2) NOT NULL,
 woman CHAR(2) NOT NULL,
 PRIMARY KEY (man, woman),
 ranking INTEGER NOT NULL);
```

```
CREATE TABLE Wives
(woman CHAR(2) NOT NULL,
 man CHAR(2) NOT NULL,
 PRIMARY KEY (woman, man),
 ranking INTEGER NOT NULL);
```

```
CREATE TABLE Wife_Perms
(perm INTEGER NOT NULL PRIMARY KEY,
 wife_name CHAR(2) NOT NULL);
```

```
-- 男性の女性に対する好みの順位
INSERT INTO Husbands -- 夫 #1
VALUES ('h1', 'w1', 5), ('h1', 'w2', 2),
       ('h1', 'w3', 6), ('h1', 'w4', 8),
       ('h1', 'w5', 4), ('h1', 'w6', 3),
       ('h1', 'w7', 1), ('h1', 'w8', 7);

INSERT INTO Husbands -- 夫 #2
VALUES ('h2', 'w1', 6), ('h2', 'w2', 3),
       ('h2', 'w3', 2), ('h2', 'w4', 1),
       ('h2', 'w5', 8), ('h2', 'w6', 4),
       ('h2', 'w7', 7), ('h2', 'w8', 5);

INSERT INTO Husbands -- 夫 #3
VALUES ('h3', 'w1', 4), ('h3', 'w2', 2),
       ('h3', 'w3', 1), ('h3', 'w4', 3),
       ('h3', 'w5', 6), ('h3', 'w6', 8),
       ('h3', 'w7', 7), ('h3', 'w8', 5);

INSERT INTO Husbands -- 夫 #4
VALUES ('h4', 'w1', 8), ('h4', 'w2', 4),
       ('h4', 'w3', 1), ('h4', 'w4', 3),
       ('h4', 'w5', 5), ('h4', 'w6', 6),
       ('h4', 'w7', 7), ('h4', 'w8', 2);

INSERT INTO Husbands -- 夫 #5
       VALUES ('h5', 'w1', 6), ('h5', 'w2', 8),
       ('h5', 'w3', 2), ('h5', 'w4', 3),
       ('h5', 'w5', 4), ('h5', 'w6', 5),
       ('h5', 'w7', 7), ('h5', 'w8', 1);

INSERT INTO Husbands -- 夫 #6
VALUES ('h6', 'w1', 7), ('h6', 'w2', 4),
       ('h6', 'w3', 6), ('h6', 'w4', 5),
       ('h6', 'w5', 3), ('h6', 'w6', 8),
       ('h6', 'w7', 2), ('h6', 'w8', 1);

INSERT INTO Husbands -- 夫 #7
VALUES ('h7', 'w1', 5), ('h7', 'w2', 1),
       ('h7', 'w3', 4), ('h7', 'w4', 2),
       ('h7', 'w5', 7), ('h7', 'w6', 3),
       ('h7', 'w7', 6), ('h7', 'w8', 8);

INSERT INTO Husbands -- 夫 #8
VALUES ('h8', 'w1', 2), ('h8', 'w2', 4),
       ('h8', 'w3', 7), ('h8', 'w4', 3),
```

```
       ('h8', 'w5', 6), ('h8', 'w6', 1),
       ('h8', 'w7', 5), ('h8', 'w8', 8);

-- 女性の男性に対する好みの順位
INSERT INTO Wives -- 妻 #1
VALUES ('w1', 'h1', 6), ('w1', 'h2', 3),
       ('w1', 'h3', 7), ('w1', 'h4', 1),
       ('w1', 'h5', 4), ('w1', 'h6', 2),
       ('w1', 'h7', 8), ('w1', 'h8', 5);

INSERT INTO Wives -- 妻 #2
VALUES ('w2', 'h1', 4), ('w2', 'h2', 8),
       ('w2', 'h3', 3), ('w2', 'h4', 7),
       ('w2', 'h5', 2), ('w2', 'h6', 5),
       ('w2', 'h7', 6), ('w2', 'h8', 1);

INSERT INTO Wives -- 妻 #3
VALUES ('w3', 'h1', 3), ('w3', 'h2', 4),
       ('w3', 'h3', 5), ('w3', 'h4', 6),
       ('w3', 'h5', 8), ('w3', 'h6', 1),
       ('w3', 'h7', 7), ('w3', 'h8', 2);

INSERT INTO Wives -- 妻 #4
VALUES ('w4', 'h1', 8), ('w4', 'h2', 2),
       ('w4', 'h3', 1), ('w4', 'h4', 3),
       ('w4', 'h5', 7), ('w4', 'h6', 5),
       ('w4', 'h7', 4), ('w4', 'h8', 6);

INSERT INTO Wives -- 妻 #5
VALUES ('w5', 'h1', 3), ('w5', 'h2', 7),
       ('w5', 'h3', 2), ('w5', 'h4', 4),
       ('w5', 'h5', 5), ('w5', 'h6', 1),
       ('w5', 'h7', 6), ('w5', 'h8', 8);

INSERT INTO Wives -- 妻 #6
VALUES ('w6', 'h1', 2), ('w6', 'h2', 1),
       ('w6', 'h3', 3), ('w6', 'h4', 6),
       ('w6', 'h5', 8), ('w6', 'h6', 7),
       ('w6', 'h7', 5), ('w6', 'h8', 4);

INSERT INTO Wives -- 妻 #7
VALUES ('w7', 'h1', 6), ('w7', 'h2', 4),
       ('w7', 'h3', 1), ('w7', 'h4', 5),
       ('w7', 'h5', 2), ('w7', 'h6', 8),
       ('w7', 'h7', 3), ('w7', 'h8', 7);
```

```
INSERT INTO Wives -- 妻 #8
VALUES ('w8', 'h1', 8), ('w8', 'h2', 2),
       ('w8', 'h3', 7), ('w8', 'h4', 4),
       ('w8', 'h5', 5), ('w8', 'h6', 6),
       ('w8', 'h7', 1), ('w8', 'h8', 3);
```

```
-- この補助テーブルによって、妻のすべての順列を作成できる
INSERT INTO Wife_Perms
VALUES (1, 'w1'), (2, 'w2'), (4, 'w3'), (8, 'w4'),
       (16, 'w5'), (32, 'w6'), (64, 'w7'), (128, 'w8');
```

正しい結果を求めるクエリは、次のようになる。

```
SELECT A.wife_name AS h1, B.wife_name AS h2,
       C.wife_name AS h3, D.wife_name AS h4,
       E.wife_name AS h5, F.wife_name AS h6,
       G.wife_name AS h7, H.wife_name AS h8
  FROM Wife_Perms AS A, Wife_Perms AS B,
       Wife_Perms AS C, Wife_Perms AS D,
       Wife_Perms AS E, Wife_Perms AS F,
       Wife_Perms AS G, Wife_Perms AS H
 WHERE A.perm + B.perm + C.perm + D.perm
       + E.perm + F.perm + G.perm + H.perm = 255
   AND NOT EXISTS
     (SELECT *
        FROM Husbands AS W, Husbands AS X, Wives AS Y, Wives AS Z
       WHERE W.man = X.man
         AND W.ranking > X.ranking
         AND X.woman = Y.woman
         AND Y.woman = Z.woman
         AND Y.ranking > Z.ranking
         AND Z.man = W.man
         AND W.man || W.woman
             IN ('h1' || A.wife_name, 'h2' || B.wife_name,
                 'h3' || C.wife_name, 'h4' || D.wife_name,
                 'h5' || E.wife_name, 'h6' || F.wife_name,
                 'h7' || G.wife_name, 'h8' || H.wife_name)
        AND Y.man || Y.woman
            IN ('h1' || A.wife_name, 'h2' || B.wife_name,
                'h3' || C.wife_name, 'h4' || D.wife_name,
                'h5' || E.wife_name, 'h6' || F.wife_name,
                'h7' || G.wife_name, 'h8' || H.wife_name));
```

結果はこうである。

```
h1    h2    h3    h4    h5    h6    h7    h8
=================================================
'w2'  'w4'  'w3'  'w1'  'w7'  'w5'  'w8'  'w6'
'w2'  'w4'  'w3'  'w8'  'w1'  'w5'  'w7'  'w6'
'w3'  'w6'  'w4'  'w1'  'w7'  'w5'  'w8'  'w2'
'w3'  'w6'  'w4'  'w8'  'w1'  'w5'  'w7'  'w2'
'w6'  'w3'  'w4'  'w1'  'w7'  'w5'  'w8'  'w2'
'w6'  'w3'  'w4'  'w8'  'w1'  'w5'  'w7'  'w2'
'w6'  'w4'  'w3'  'w1'  'w7'  'w5'  'w8'  'w2'
'w6'  'w4'  'w3'  'w8'  'w1'  'w5'  'w7'  'w2'
'w7'  'w4'  'w3'  'w8'  'w1'  'w5'  'w2'  'w6'
```

その3

答え

この問題の肝は、NOT EXISTSによる比較処理のパフォーマンスを最大化することにある。以下に示すUnstableテーブルは、不安定な関係（bad_marriage）の一覧である。続くINSERT文は、最後に記すクエリのLIKE述語の引数として使うことを意図して書かれている。挿入データのアンダースコア（_）は、LIKEのワイルドカードとして使う点に注目してほしい。

```
CREATE TABLE Unstable
(bad_marriage CHAR(16) NOT NULL);
```

```
INSERT INTO Unstable
SELECT DISTINCT
       CASE WHEN W.man = 'h1' THEN W.woman
            WHEN Y.man = 'h1' THEN Y.woman
                              ELSE '__' END
    || CASE WHEN W.man = 'h2' THEN W.woman
            WHEN Y.man = 'h2' THEN Y.woman
                              ELSE '__' END
    || CASE WHEN W.man = 'h3' THEN W.woman
            WHEN Y.man = 'h3' THEN Y.woman
                              ELSE '__' END
    || CASE WHEN W.man = 'h4' THEN W.woman
            WHEN Y.man = 'h4' THEN Y.woman
                              ELSE '__' END
    || CASE WHEN W.man = 'h5' THEN W.woman
            WHEN Y.man = 'h5' THEN Y.woman
                              ELSE '__' END
    || CASE WHEN W.man = 'h6' THEN W.woman
            WHEN Y.man = 'h6' THEN Y.woman
```

```
                              ELSE '__' END
    || CASE WHEN W.man = 'h7' THEN W.woman
            WHEN Y.man = 'h7' THEN Y.woman
                              ELSE '__' END
    || CASE WHEN W.man = 'h8' THEN W.woman
            WHEN Y.man = 'h8' THEN Y.woman
                              ELSE '__' END
  FROM Husbands AS W, Husbands AS X,
       Wives AS Y, Wives AS Z
 WHERE W.man = X.man
   AND W.ranking > X.ranking
   AND X.woman = Y.woman
   AND Y.woman = Z.woman
   AND Y.ranking > Z.ranking
   AND Z.man = W.man;
```

```
SELECT A.wife_name AS h1, B.wife_name AS h2, C.wife_name AS h3,
       D.wife_name AS h4, E.wife_name AS h5, F.wife_name AS h6,
       G.wife_name AS h7, H.wife_name AS h8
  FROM wife_perms AS a, wife_perms AS b, wife_perms AS c,
       wife_perms AS d, wife_perms AS e, wife_perms AS f,
       wife_perms AS g, wife_perms AS h
 WHERE B.wife_name NOT IN (A.wife_name)
   AND C.wife_name NOT IN (A.wife_name, B.wife_name)
   AND D.wife_name NOT IN (A.wife_name, B.wife_name, C.wife_name)
   AND E.wife_name NOT IN (A.wife_name, B.wife_name, C.wife_name,
       D.wife_name)
   AND F.wife_name NOT IN (A.wife_name, B.wife_name, C.wife_name,
       D.wife_name, E.wife_name)
   AND G.wife_name NOT IN (A.wife_name, B.wife_name, C.wife_name,
       D.wife_name, E.wife_name, F.wife_name)
   AND H.wife_name NOT IN (A.wife_name, B.wife_name, C.wife_name,
       D.wife_name, E.wife_name, F.wife_name, G.wife_name)
   AND NOT EXISTS
      (SELECT * FROM Unstable
        WHERE A.wife_name || B.wife_name || C.wife_name ||
              D.wife_name || E.wife_name || F.wife_name ||
              G.wife_name || H.wife_name
              LIKE bad_marriage);
```

結果は次のようになる。

```
h1    h2    h3    h4    h5    h6    h7    h8
=================================================
'w3'  'w6'  'w4'  'w8'  'w1'  'w5'  'w7'  'w2'
'w6'  'w4'  'w3'  'w8'  'w1'  'w5'  'w7'  'w2'
'w6'  'w3'  'w4'  'w8'  'w1'  'w5'  'w7'  'w2'
'w3'  'w6'  'w4'  'w1'  'w7'  'w5'  'w8'  'w2'
'w6'  'w4'  'w3'  'w1'  'w7'  'w5'  'w8'  'w2'
'w6'  'w3'  'w4'  'w1'  'w7'  'w5'  'w8'  'w2'
'w7'  'w4'  'w3'  'w8'  'w1'  'w5'  'w2'  'w6'
'w2'  'w4'  'w3'  'w8'  'w1'  'w5'  'w7'  'w6'
'w2'  'w4'  'w3'  'w1'  'w7'  'w5'  'w8'  'w6'
```

リチャードがこのクエリの実行時間を計ったところ、4万行を4秒で処理したという。1秒につき、平均1万行前後を処理した計算だ。私としては、これ以上のパフォーマンスは望めないと思う。

■参考文献

① Gusfierld, Dan, and Irving, Robert W., The Stable Marriage Problem: Structure & Algorithms, ISBN 0-262-07118-5.

② Knuth, Donald E., CRM Proceedings & Lecture Notes, Vol #10, "Stable Marriage and Its Relation to Other Combinatorial Problems," ISBN 0-8218-0603-3.
この小冊子は、著者のドナルド・クヌースが1975年に行った講演を採録したもので、「安定な結婚」の美しい理論を使った分析に対するやさしい解説となっている。このテーマの基本的なパラダイムを説明してくれている。

③ Wirth, Niklaus, Section 3.6, Algorithms + Data Structures = Programs, ISBN 0-13-022418-9.（邦訳『アルゴリズム＋データ構造＝プログラム』科学技術出版社, 1979）
この本のセクション3.6ではPascal言語を使った解と、アルゴリズムについての短い分析が示されている。今回は、この本のデータをサンプルに使わせてもらった。著者のニクラウス・ヴィルトは、さまざまに異なる夫と妻の「幸福の程度」を与える複数の解答を与えている。

パ|ズ|ル|
68 バスを待ちながら

時間データの扱い方（その1）

人がバス停へやってきて、最初に来たバスに乗っていくとしよう。このバス停に止まるバスには、「A」と「B」の2つの運行ルートしかないとする。また、どちらのルートのバスも、ちょうど1時間おきにバス停に到着する。ここまで聞くと、バス停にやってくる人は、どちらのルートのバスを待つ時間も大体同じぐらいだと思うだろう。ところが実際には、バスBよりもAのほうが圧倒的に待ち時間が長いのだ。なぜか？

実は、そのバス停では、バスAは毎時00分に出発し、バスBは毎時05分に出発する。したがって、バスBに乗るためには、00分と05分の5分間の間にバス停に到着しなければならない。残りの時間に到着した人は、バスAが来るまで座って待つほかない。

さて、このように次に出発するバスを見つけるという問題は、SQLを使えば割と簡単に解くことができる。以下に、架空のバスの運行スケジュールを保持する簡単なテーブルを示す。なお、出発地と目的地の情報は問題と関係ないので省略する。

```
CREATE TABLE Schedule
(route_nbr INTEGER NOT NULL,
 depart_time TIMESTAMP NOT NULL,
 arrive_time TIMESTAMP NOT NULL,
 CHECK (depart_time < arrive_time),
 PRIMARY KEY (route_nbr, depart_time));
```

Schedule：運行スケジュールテーブル　　route_nbr：運行便番号
depart_time：出発時刻　　arrive_time：到着時刻

```
INSERT INTO Schedule
VALUES (3, '2006-02-09 10:00', '2006-02-09 14:00'),
       (4, '2006-02-09 16:00', '2006-02-09 17:00'),
       (5, '2006-02-09 18:00', '2006-02-09 19:00'),
       (6, '2006-02-09 20:00', '2006-02-09 21:00'),
       (7, '2006-02-09 11:00', '2006-02-09 13:00'),
       (8, '2006-02-09 15:00', '2006-02-09 16:00'),
       (9, '2006-02-09 18:00', '2006-02-09 20:00');
```

例えば、'2006-02-09 15:30' にバス停に到着した人に対しては、route_nbr = 4（ルート4番）が返すべき結果となる。バス停に到着した時刻が '2006-02-09 16:30' の場合は、同時刻に出発するルート5番とルート9番が答えになる。

その1

答え　さて、どうすればこの問題を解けるだろうか？ 計算を使う方法なら、到着時刻以降のバスの出発時刻を求めることになる。すると、次のようなクエリが考えられる。

```
SELECT route_nbr, depart_time, arrive_time
  FROM Schedule
 WHERE depart_time
       = (SELECT MIN(depart_time)
            FROM Schedule
           WHERE :my_time <= depart_time);
```

出発時刻（depart_time）は主キーの一部なのでインデックスが使用され、MIN関数は高速に計算される[注1]。したがって、このクエリはパフォーマンスも良好なはずだ。

その2

答え　これで一応は解けたわけだが、「最小値を求めずに解く」方法はないだろうか？

あるバスが出発するまでに、待たねばならない時刻を示す列をテーブルに追加してみよう。ルート3番は始発なので、深夜0時00分から朝の10時00分までにいつ到着してもかまわない。10時00分から11時00分の間に到着した場合には、ルート7番に乗ることになる。そのほかのバスについても同様だ。すると、テーブル定義は次のようになる。

```
CREATE TABLE Schedule
(route_nbr INTEGER NOT NULL,
 wait_time TIMESTAMP NOT NULL,
 depart_time TIMESTAMP NOT NULL,
 arrive_time TIMESTAMP NOT NULL,
 CHECK (depart_time < arrive_time),
 PRIMARY KEY (route_nbr, depart_time));
```

Schedule：運行スケジュールテーブル	route_nbr：順路番号	wait_time：待ち時間
depart_time：出発時刻	arrive_time：到着時刻	

注1：ツリーインデックスを基本とする古いRDBMS製品では、(route_nbr, depart_time)と(depart_time, route_nbr)のインデックスは別物だったが、ハッシュやビットベクトルなどのインデックスも備えた製品では、このような"キーの並び"による制限はない。また、各キー列の統計情報を保持する製品もあり、その最小値や最大値、COUNT(*)、COUNT(DISTINCT <列名>)などの情報を簡単に取得できる。

サンプルデータはこうなる。

```
INSERT INTO Schedule
VALUES (3, '2006-02-09 00:00', '2006-02-09 10:00',
        '2006-02-09 14:00'),
       (7, '2006-02-09 10:00', '2006-02-09 11:00',
        '2006-02-09 13:00'),
       (8, '2006-02-09 11:00', '2006-02-09 15:00',
        '2006-02-09 16:00'),
       (4, '2006-02-09 15:00', '2006-02-09 16:00',
        '2006-02-09 17:00'),
       (5, '2006-02-09 16:00', '2006-02-09 18:00',
        '2006-02-09 19:00'),
       (9, '2006-02-09 16:00', '2006-02-09 18:00',
        '2006-02-09 20:00'),
       (6, '2006-02-09 18:00', '2006-02-09 20:00',
        '2006-02-09 21:00');
```

テーブル設計を変えたことで、サブクエリなしでクエリを書くことが可能になった。

```
SELECT route_nbr, depart_time, arrive_time
  FROM Schedule
 WHERE :my_time > wait_time AND :my_time <= depart_time;
```

実行効率という面から見て、このクエリはどうだろう？ まず、このスケジュール表の各行について、TIMSTAMP型1列分が追加で必要になる。するとシステム全体では、(365日×ルート本数×TIMSTAMP型のサイズ) の記憶領域を追加で必要とする。これは悪い話ではない。バスの運行表の場合、少なくとも数ヶ月の間は一定だと考えられる。本当の疑問は、「wait_timeの計算にどれぐらいの時間がかかるか」だ。恐らくそれには、最初のクエリを1度実行するのと同じだけの時間、およびテーブルデータをUPDATE文ないしINSERT文の一部で処理する時間がかかるだろう。

データベースの問題を考える際に、テーブル駆動型 (table-driven) の解法[訳注1]を見つけるためには、あなた自身の考え方を少し変えてみるだけでよいのだ。

訳注1：テーブル駆動型の解法とは、計算の代わりに参照テーブルを使用するものである。

パズル 69

後入れ先出しと先入れ先出し

部分和問題の解き方

1種類の商品 —— 何かの部品だとしておこう —— だけしか保管しない、とても単純な倉庫を想像してほしい。この倉庫には1日1回、在庫が追加（入庫）される。同様に、この倉庫は1日1回やってくる要求に応じて部品を出庫する。この、ちょっとした問題で使用するテーブルを以下に示す。

```
CREATE TABLE WidgetInventory
(receipt_nbr INTEGER NOT NULL PRIMARY KEY,
 purchase_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
 qty_on_hand INTEGER NOT NULL
 CHECK (qty_on_hand >= 0),
 unit_price DECIMAL (12,4) NOT NULL);
```

WidgetInventory：在庫テーブル	receipt_nbr：受入番号
purchase_date：仕入日	qty_on_hand：入庫数（その日の仕入数）
unit_price：仕入単価	

データは次のようなものを使おう。

```
WidgetInventory

receipt_nbr   purchase_date   qty_on_hand   unit_price
=========================================================
          1   '2005-08-01'            15        10.00
          2   '2005-08-02'            25        12.00
          3   '2005-08-03'            40        13.00
          4   '2005-08-04'            35        12.00
          5   '2005-08-05'            45        10.00
```

仮に、2005年8月5日に部品が100個売れたとする。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。なお、正解は1つではなく、原価の計算方法には複数の選択肢がある。

1. 2005年8月5日現在の取替原価[訳注1]を用いる。この方法だと、現在は仕入単価

訳注1：現時点で同じ資産を調達するのにかかるコスト。

が落ち込んでいるので、部品の仕入原価は1000.00ドルで済む。

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1840.00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1150.00ドルになる。

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。

```
2005年8月5日仕入れ分
…… 45個×10.00ドル＝450.00ドル（出荷数45個）
2005年8月4日仕入れ分
…… 35個×12.00ドル＝420.00ドル（出荷数80個）
2005年8月3日仕入れ分
…… 20個×13.00ドル＝260.00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1130.00ドルとなる。

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。

```
2005年8月1日仕入れ分
…… 15個×10.00ドル＝150.00ドル（出荷数15個）
2005年8月2日仕入れ分
…… 25個×12.00ドル＝300.00ドル（出荷数40個）
2005年8月3日仕入れ分
…… 40個×13.00ドル＝520.00ドル（出荷数80個)
2005年8月4日仕入れ分
…… 20個×12.00ドル＝240.00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1210.00ドルとなる。
これら4つのシナリオのうち、最初の2つをプログラミングするのは簡単だ。

```
CREATE VIEW ReplacementCost(current_replacement_cost)
AS
SELECT unit_price
  FROM WidgetInventory
 WHERE purchase_date
       = (SELECT MAX(purchase_date) FROM WidgetInventory);
```

current_replacement_cost：2005年8月5日現在の取替原価

```
CREATE VIEW AverageCost(average_replacement_cost)
AS
SELECT SUM(unit_price * qty_on_hand) / SUM(qty_on_hand)
  FROM WidgetInventory;
```

average_replacement_cost：現時点での平均原価

LIFOとFIFOのケースはもっと歯ごたえがある。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。

その1

答え

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 W1.purchase_date, W1.unit_price,
       SUM(W2.qty_on_hand), SUM(W2.qty_on_hand * W2.unit_price)
  FROM WidgetInventory AS W1, WidgetInventory AS W2
 WHERE W2.purchase_date >= W1.purchase_date
 GROUP BY W1.purchase_date, W1.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)
        * 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);
```

ここでは、要求数（:order_qty）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。要求数以上の在庫数がある場合には、要求数を満たすだけの在庫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）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。

一方、FIFOについても類似の方法で計算することができる。まずは、次のSQL文でビューを作成する。

```
CREATE VIEW FIFO
(stock_date, unit_price, tot_qty_on_hand, tot_cost)
AS
SELECT W1.purchase_date, W1.unit_price,
       SUM(W2.qty_on_hand), SUM(W2.qty_on_hand * W2.unit_price)
  FROM WidgetInventory AS W1, WidgetInventory AS W2
 WHERE W2.purchase_date <= W1.purchase_date
 GROUP BY W1.purchase_date, W1.unit_price;
```

そして、次のクエリを実行する。

```
SELECT (tot_cost - ((tot_qty_on_hand - :order_qty)
        * 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);
```

その2

答え　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LIFOのSQL文では、CASE式により、在庫の累計が:order_qty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(W3.v)でこれらの計算結果を合計している。

```
SELECT SUM(W3.v) AS cost
  FROM (
    SELECT W1.unit_price
           * CASE WHEN SUM(W2.qty_on_hand) <= :order_qty
                  THEN W1.qty_on_hand
                  ELSE :order_qty
                       - (SUM(W2.qty_on_hand) - W1.qty_on_hand)
             END
      FROM WidgetInventory AS W1, WidgetInventory AS W2
     WHERE W1.purchase_date <= W2.purchase_date
     GROUP BY W1.purchase_date, W1.qty_on_hand, W1.unit_price
    HAVING (SUM(W2.qty_on_hand) - W1.qty_on_hand) <= :order_qty
  ) AS W3(v);
```

なお、答えその1、その2で示したクエリとビューからは、部品の在庫数しか分からない。実際に出庫を行うわけではない点に注意しよう。ここでは、購入日の古い部品を優先的に出庫するやり方で考える。

その3

答え

では、在庫を更新するUPDATE文はどう書くのだろう？ これまでの答えでは、部品が出庫されたときにも在庫の変更までは行わなかった。別のビューを作って、変更処理を容易にしよう。

```
CREATE VIEW StockLevels (purchase_date, previous_qty,current_qty)
AS
SELECT W1.purchase_date,
       SUM(CASE WHEN W2.purchase_date < W1.purchase_date
                THEN W2.qty_on_hand ELSE 0 END),
       SUM(CASE WHEN W2.purchase_date <= W1.purchase_date
                THEN W2.qty_on_hand ELSE 0 END)
  FROM WidgetInventory AS W1, WidgetInventory AS W2
 WHERE W2.purchase_date <= W1.purchase_date
 GROUP BY W1.purchase_date, W1.unit_price;
```

StockLevels：在庫水準ビュー	purchase_date：仕入日
previous_qty：仕入れ前の在庫累計	current_qty：仕入れ後の在庫累計

```
StockLevels

purchase_date   previous_qty   current_qty
=========================================
'2005-08-01'              0            15
'2005-08-02'             15            40
'2005-08-03'             40            80
'2005-08-04'             80           115
'2005-08-05'            115           160
```

CASE式を使えば、自己結合を行わなくて済む。

```
CREATE PROCEDURE RemoveQty (IN my_order_qty INTEGER)
LANGUAGE SQL
BEGIN

IF my_order_qty > 0
THEN
UPDATE WidgetInventory
   SET qty_on_hand
       = CASE WHEN my_order_qty
                   >= (SELECT current_qty
                         FROM StockLevels AS L
                        WHERE L.purchase_date
                              = WidgetInventory.purchase_date)
              THEN 0
              WHEN my_order_qty
                   < (SELECT previous_qty
                        FROM StockLevels AS L
                       WHERE L.purchase_date
                             = WidgetInventory.purchase_date)
              THEN WidgetInventory.qty_on_hand
              ELSE (SELECT current_qty
                      FROM StockLevels AS L
                     WHERE L.purchase_date
                           = WidgetInventory.purchase_date)
                   - my_order_qty
         END;
END IF;

-- 部品の仕入数が0になった日の行を削除する
DELETE FROM WidgetInventory
 WHERE qty_on_hand = 0;

END;
```

その4

答え 在庫に関するもう1つの問題は、「ピックアップするビン[訳注2]の数を最小、あるいは最大にして、要求された数の出庫を行うにはどうするか」である。ここでは毎日の入庫ごとに1つのビンを作るが、特に順序は設定せず、どのビンからでも出庫できるものとしよう。先のサンプルデータでは、「部品80個」という出庫要求に対し、受入番号(1, 2, 3)と受入番号(4, 5)というビンの組み合わせが考えられる。たまたま組み合わせが受入番号順に並んだが、そうである必要はない。

この問題は、数学者の間で(論理的には紛れもなく)「ビン詰め問題[訳注3]」と呼ばれるもので、NP完全問題の一種である。ただ、この種の問題を一般化して解くことはできない。解を出すにはすべての組み合わせを調べることになるのだが、コンピュータが処理しきれないほど、計算量が膨大になってしまうのだ。

しかし、心配には及ばない。ほぼ最適と言える解を大方得ることのできる「欲張り法」という解法が存在するのだ。そのアルゴリズムは、「目標の数値に達するか超えるかするまで、その時点で取り得る"一番大きな値"を選んでいく」というものだ。

手続き型言語で欲張り法を実行する場合、目標値を超過したときにバックトラックが行える。そのため、要求数と出庫数がぴたり一致する組み合わせをあくまで探すのか、あるいは完全に一致しない組み合わせであっても出庫数が要求数を超えた時点で解とするか、を選ぶことができる。

一方、SQLは宣言的で集合指向的な言語であるため、これを実現するのは容易ではない。手続き型言語であれば、「要求数は満たした」という解を見つけた時点で処理をストップできる。だが、SQLのクエリは、どんなに時間がかかろうとも、すべての正答を見つけようとする。もし、ピックアップするビンの数に上限を設けられるなら、配列をテーブルで擬似的に表現するという方法がある。

```
CREATE TABLE Picklists
(order_nbr INTEGER NOT NULL PRIMARY KEY,
 goal_qty INTEGER NOT NULL
    CHECK (goal_qty > 0),
 bin_1 INTEGER NOT NULL UNIQUE,
```

訳注2：ビンは、在庫を管理上の都合などで分けたときのひとまとまり。ピックアップは、出庫するために在庫を集める作業のこと。この問題ではWidgetInventoryテーブルの1行＝1ビンと考えてよい。

訳注3：部分和問題という計算複雑性理論の分野における問題の一種。より一般化した形の「ナップサック問題」という名前のほうが有名かもしれない。

```
qty_on_hand_1 INTEGER DEFAULT 0 NOT NULL
   CHECK (qty_on_hand_1 >= 0),
bin_2 INTEGER NOT NULL UNIQUE,
qty_on_hand_2 INTEGER DEFAULT 0 NOT NULL
   CHECK (qty_on_hand_2 >= 0),
bin_3 INTEGER NOT NULL UNIQUE,
qty_on_hand_3 INTEGER DEFAULT 0 NOT NULL
   CHECK (qty_on_hand_3 >= 0),
CONSTRAINT not_over_goal
   CHECK (qty_on_hand_1 + qty_on_hand_2 + qty_on_hand_3
          <= goal_qty),
CONSTRAINT bins_sorted
   CHECK (qty_on_hand_1 >= qty_on_hand_2
          AND qty_on_hand_2 >= qty_on_hand_3));
```

Picklists：ピックアップリストテーブル　order_nbr：注文番号　goal_qty：出庫要求個数
bin_1～bin_3：ビン番号　qty_on_hand_1～qty_on_hand_3：各ビンに含まれる部品の個数

では、WidgetInventoryテーブルにデータを詰めていこう。最初の“トリック”は、WidgetInventoryテーブルにビンのダミーデータを追加することだ。もし、ビンのピックアップ回数が最大nならば、n－1個のダミーデータを追加する。Picklistsテーブルではピックアップ回数を3回までとしているので、ダミーデータを2つ追加しておく。

```
INSERT INTO WidgetInventory VALUES (-1, '1990-01-01', 0, 0.00);
INSERT INTO WidgetInventory VALUES (-2, '1990-01-02', 0, 0.00);
```

続いて次のコードで、ビン3個までの全組み合わせパターンとそのときの部品個数をリストアップする共通表式、あるいはビューを作る。

```
CREATE VIEW PickCombos (total_pick, bin_1, qty_on_hand_1, bin_2,
                        qty_on_hand_2, bin_3, qty_on_hand_3)
AS
SELECT DISTINCT
      (W1.qty_on_hand + W2.qty_on_hand + W3.qty_on_hand)
       AS total_pick,
       CASE WHEN W1.receipt_nbr < 0
            THEN 0
            ELSE W1.receipt_nbr
            END AS bin_1,
       W1.qty_on_hand,
       CASE WHEN W2.receipt_nbr < 0
            THEN 0
```

```
            ELSE W2.receipt_nbr
            END AS bin_2,
       W2.qty_on_hand,
       CASE WHEN W3.receipt_nbr < 0
            THEN 0
            ELSE W3.receipt_nbr
            END AS bin_3,
       W3.qty_on_hand
  FROM WidgetInventory AS W1, WidgetInventory AS W2,
       WidgetInventory AS W3
 WHERE W1.receipt_nbr NOT IN (W2.receipt_nbr, W3.receipt_nbr)
   AND W2.receipt_nbr NOT IN (W1.receipt_nbr, W3.receipt_nbr)
   AND W1.qty_on_hand >= W2.qty_on_hand
   AND W2.qty_on_hand >= W3.qty_on_hand;
```

ここまで来たら、あとは出庫要求数に一致する、あるいは最も近いビンの組み合わせを見つけ出すプロシージャを書くだけだ。

```
CREATE PROCEDURE OverPick (IN goal_qty INTEGER)
LANGUAGE SQL
BEGIN

IF goal_qty > 0
THEN
SELECT goal_qty, total_pick,
       bin_1, qty_on_hand_1,
       bin_2, qty_on_hand_2,
       bin_3, qty_on_hand_3
  FROM PickCombos
 WHERE total_pick
         = (SELECT MIN (total_pick)
              FROM PickCombos
             WHERE total_pick >= goal_qty)
END IF;
END;
```

SQL-99の構文を使えば、ビューを共通表式の中に入れられるので、ビューを使わないクエリが作成できる。このクエリを使い、サンプルデータに対して73個という要求を満たす組み合わせを探してみると、{3, 4}と{4, 2, 1}という、ともに75個になる組み合わせが見つかる。

反対に、要求された個数以下で最も近い部品数を得るための組み合わせを見つけるクエリもあるが、それは演習問題として残しておこう。

パ|ズ|ル|
70

株価の動向

相関サブクエリで行同士を比較する

ピュアSQL[訳注1]のRDBMSでは、計算した列をテーブルで持たないことになっている。私もピュアSQLの守護者の立場から、そのような所業には反対しなければならない。とはいえ、基本的に反対ではあるものの、それが便利な局面があるのも分かるし、ビューを通して実現する分には技術的にも問題はない。

計算列の第1のタイプは、同じ行にあるほかの列から算出されるものだ。かつてパンチカードが使われていた時代には、掛け算や足し算をするためのマシンに一組のカードをかけて、計算結果はカードの右端に打ち出されていた。例えば、ある順序で並んだ各行のトータルコストを(closing_price × quantity)で計算する、といった具合だ。

こういう計算の仕方を採っていた理由は、カードを処理するマシンが2次記憶を持っていなかった、というごく単純なものである。そのため、データはカード自身に保持するしかなかった。また、1枚のカードが持つ80桁全部を一度に主記憶に読み込むほうが、ハードウェア内で計算するより速かった。

だが今日では、この方式を採る理由はない。結果を2次記憶から読み込むより、データを再計算するほうがずっと高速だからだ。ハードディスクはマイクロ秒の世界で動作するが、CPUはナノ秒の世界で動く。

計算列の第2のタイプは、同じテーブルのデータを使うが、必ずしも同じ行のデータとは限らないもの、第3のタイプは、同じデータベース内の複数テーブルを使うものだ。これら2種類のタイプの計算列が使われるのは、計算のコストが単純な読み出しのコストより高い場合である。特にデータウェアハウスでは、処理時間を節約するためにこの種の計算列を持つことを好む。

SQLでは何をするにつけ、いつどのように行うかが重要である。これから示す例題は、あるSQL Serverのディスカッショングループのスレッドで交わされていた議論がもとになっている。テーブルは少し変えてあるし、議論に加わっていた当事者たちの名前も伏せるが、アイデアは同じままだ。

まず、次のようなテーブルが与えられており、同じテーブルにあるほかの行に基づいて列を計算する必要があるとする。

訳注1：ANSIによる標準SQL規格のこと。したがって、ベンダの独自拡張機能や、ストアドプロシージャなどを含まない。

```
CREATE TABLE StockHistory
(ticker_sym CHAR(5) NOT NULL,
 sale_date DATE DEFAULT CURRENT_DATE NOT NULL,
 closing_price DECIMAL (10,4) NOT NULL,
 trend INTEGER DEFAULT 0 NOT NULL
   CHECK(trend IN(-1, 0, 1)),
 PRIMARY KEY (ticker_sym, sale_date));
```

StockHistory：株価履歴テーブル	ticker_sym：銘柄コード	sale_date：取引日
closing_price：終値	trend：トレンド	

このテーブルは、いろいろな種類の株の終値（おわりね）を記録している。trend列には、ある行の終値が前回の取引日における終値より値を上げていれば＋1、下げていれば－1、横ばいなら0が入る。このtrend列が問題なのだが、その理由は計算が難しいからではなく、複数の計算方法が存在するからだ。以下、この列を計算する方法を見ていくとしよう

その1

答え

第1に、新しい行が挿入されたタイミングで起動するトリガーを書く方法が考えられる。ただし、一応、トリガーを書くためのISO標準のSQL/PSM言語はあるものの、各ベンダのトリガー言語は独自仕様で、互換性がないのが実情だ。そのため、使える機能も製品によってさまざまに異なるし、基礎としているデータモデルも全く異なる。もしトリガーを使うなら、実装依存で非関係的なコードを使うことになり、いくつかの問題に直面せざるを得ない。

一例を挙げるなら、トリガーがバルク（一括）インサートに対してどう動くか、という問題がある。例として、同時に2行を挿入するSQL文を見てみよう。

```
INSERT INTO StockHistory (ticker_sym, sale_date, closing_price)
VALUES ('XXX', '2000-04-01', 10.75),
       ('XXX', '2000-04-03', 200.00);
```

まず、テーブル定義のDEFAULT句によって2行ともtrend列にはゼロがセットされる。さて、このときトリガーは、'2000-04-03' の行のtrendが＋1にするか否かを正しく判断できるだろうか？ これは五分五分といったところだろう。なぜなら、挿入された2行が、必ずトリガーの起動前にコミットされるとは限らないからだ。また、'2000-04-01' の行のtrend列はどうなるのだろう？ こちらは、既存の行に応じて変わる。

だが、とにかくトリガーは正しく動作すると仮定しよう。すると、次のような中間日のデータを挿入するINSERT文に対してはどう制御するだろう。

```
INSERT INTO StockHistory (ticker_sym, sale_date, closing_price)
VALUES ('XXX', '2000-04-02', 313.25);
```

果たして、'2000-04-03'の行のtrend列は変更されるだろうか？ また、行を削除したときに、それによって影響を受ける行が変更されるだろうか？ 多分、そうはなるまい。

練習問題として、この問題に対処するトリガーを書いてみてほしい。

その2

答え

ところで、この問題はトリガーを使わなくても、INSERT文だけで解ける。自分の腕を見せびらかしている気がしなくもないが、次に示すのは1度に1行だけINSERTする方法の1つである。

```
INSERT INTO StockHistory
(ticker_sym, sale_date, closing_price, trend)
VALUES (:new_ticker_sym, :new_sale_date, :new_closing_price,
        COALESCE(SIGN(:new_closing_price
        - (SELECT H1.closing_price
             FROM StockHistory AS H1
            WHERE H1.ticker_sym = StockHistory.ticker_sym
              AND H1.sale_date
                  = (SELECT MAX(sale_date)
                       FROM StockHistory AS H2
                      WHERE H2.ticker_sym = :new_ticker_sym
                        AND H2.sale_date < :new_sale_date))),0));
```

これは見た目ほど悪いコードではない。一番内側のサブクエリはカレント行の取引日の直前の取引を探し、そのときの終値を返す。SIGN関数は、その終値から追加行の終値を引いた値が正か負かゼロかによって、trend列に入る値（正なら1、負なら−1、ゼロなら0）を返してくれる。このクエリはちょっと自慢の一品だ。

しかし、この解法にはトリガーを使用する場合と同じ問題が残っている。既存の2行に挟まれた行を削除したり、あるいは中間に挿入した場合にどうなるのか。この文はただのINSERT文なので、当然のことながら既存の行に変更を加えたりしない。

しかも、問題はそれだけではない。このINSERT文は1度の実行で1行だけを挿入す

る場合しかうまく動作しない。だから、もしこれを使うなら、営業日の終わりに1行ずつStockHistoryテーブルへ挿入するためのループをプロシージャで書かねばならない。

これをどう改善するかは、第2の練習問題にしておこう。

その3

答え

テーブルをUPDATEする解法も考えられる。DEFAULT句によって、trend列にはすでに0がセットされているので、先ほどのINSERT文と同じロジックのUPDATE文を書くことができる。

```
UPDATE StockHistory
   SET trend
       = COALESCE(SIGN(closing_price
         - (SELECT H1.closing_price
              FROM StockHistory AS H1
             WHERE H1.ticker_sym = StockHistory.ticker_sym
               AND H1.sale_date =
                   (SELECT MAX(sale_date)
                      FROM StockHistory AS H2
                     WHERE H2.ticker_sym
                           = StockHistory.ticker_sym
                       AND H2.sale_date
                           < StockHistory.sale_date))),0);
```

この文は正しいが、テーブル全体のtrend列を再計算してしまうのが玉に瑕(きず)だ。そこで、値がゼロの行だけを参照するようにしてはどうだろう。あるいは、trend列にNULLを許可し、更新対象とすべき行を把握するためにNULLを使うのはどうか。

```
UPDATE StockHistory
   SET trend = ...
 WHERE trend IS NULL;
```

ただし、この場合もやはり既存の2行の間に挿入される行に問題が残る。この点を修正するのが3つ目の練習問題だ。

最後に、ビューを使う方法を紹介しよう。これによって、StockHistoryテーブルがtrend列を持つ必要がなくなる。ビューはそれ以外の列を使って作成する[訳注2]。

訳注2：このビュー定義では自己外部結合を使って、各銘柄の初日の行も保存している（trend列は0）。初日の行が不要な場合には、外部結合を内部結合に変えればよい。

```
CREATE TABLE StockHistory
(ticker_sym CHAR(5) NOT NULL,
 sale_date DATE DEFAULT CURRENT_DATE NOT NULL,
 closing_price DECIMAL (10,4) NOT NULL,
 PRIMARY KEY (ticker_sym, sale_date));
```

```
CREATE VIEW StockTrends
(ticker_sym, sale_date, closing_price, trend)
AS SELECT H1.ticker_sym, H1.sale_date, H1.closing_price,
          COALESCE(SIGN(MAX(H2.closing_price)
          - H1.closing_price), 0) AS trend
     FROM StockHistory AS H1
          LEFT OUTER JOIN StockHistory AS H2
          ON H1.ticker_sym = H2.ticker_sym
          AND H2.sale_date < H1.sale_date
    GROUP BY H1.ticker_sym, H1.sale_date, H1.closing_price;
```

この方法ならば、挿入と削除が何行でも、どんな順番で起きようとも問題ない。trend列は、各時点で存在するデータから計算されるからだ。このクエリでは、主キー（ticker_sim, sale_date, closing_price）がWHERE句の条件をカバーしているため、インデックスが使える。そのためパフォーマンスも向上する。

この方法に対する主な反論は、StockHistoryテーブルの規模が大きかった場合、その都度ビューを計算するのは時間がかかるかもしれない、というものだろう。

その4

SQL-99のOLAP関数を使えば、答えその3のビューはずっと簡単になる。

```
CREATE VIEW StockTrends
(ticker_sym, sale_date, closing_price, trend)
AS SELECT ticker_sym, sale_date, closing_price,
          COALESCE(SIGN(closing_price - MAX(closing_price)
             OVER(PARTITION BY ticker_sym
             ORDER BY sale_date ASC
             ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)), )
     FROM StockHistory;
```

パズル 71

計算

自己結合でクエリの見通しをよくする

この問題は、2006年にあるニュースグループに持ち込まれたものだが、こんなひどいコードは今後誰にも書いてほしくない。目的は「3つの条件に基づいて、同一行内の3つの異なる値を選択する」ことなのだが、問題のクエリの計算が間違っていた上に、DDL文も存在しないという有様だったのだ。

まずは、オリジナルのクエリからお見せしよう。

```
SELECT DISTINCT SUM(A.calc_rslt_val + A.calc_adj_val),
       SUM(A.unit_rslt_val + A.unit_adj_val),
       SUM(OT1.calc_rslt_val + OT1.calc_adj_val),
       SUM(OT1.unit_rslt_val + OT1.unit_adj_val),
       SUM(OT2.calc_rslt_val + OT2.calc_adj_val),
       SUM(OT2.unit_rslt_val + OT2.unit_adj_val)
  FROM Table1 AS A, Table1 AS OT1, Table1 AS OT2, Table2 AS B
 WHERE OT1.emp_id = A.emp_id
   AND OT2.emp_id = A.emp_id
   AND OT1.pin_num = B.pin_num
   AND OT2.pin_num = B.pin_num
   AND A.empl_rcd = 0
   AND A.pin_num = B.pin_num
   AND A.emp_id = 'xxxxxx'
   AND B.pin_num IN ('52636','52751','52768')
   AND A.pin_num = '52636'
   AND OT1.pin_num = '52751'
   AND OT2.pin_num = '52768'
```

その1

答え

USASQLという人物は次のような解答を送ってきた。これは、複数のUNIONを使うことを前提としている。

```
SELECT SUM(A_val1),   SUM(A_val2),   SUM(OT1_val1),
       SUM(OT1_val2), SUM(OT2_val1), SUM(OT2_val2)
       ((SELECT DISTINCT
                SUM(A.calc_rslt_val + A.calc_adj_val) AS A_val1,
                SUM(A.unit_rslt_val + A.unit_adj_val) AS A_val2,
                0 AS OT1_val1, 0 AS OT1_val2,
                0 AS OT2_val1, 0 AS OT2_val2
           FROM Table1 AS A,Table1 AS OT1,
```

```
                Table1 AS OT2, Table2 AS B
          WHERE -- Aテーブルの選択条件のみが入る
UNION
SELECT DISTINCT 0, 0,
                SUM(OT1.calc_rslt_val + OT1.calc_adj_val),
                SUM(OT1.unit_rslt_val + OT1.unit_adj_val),
                0, 0
  FROM Table1 AS A, Table1 AS OT1, Table1 AS OT2, Table2 AS B
 WHERE -- OT1テーブルの選択条件のみが入る
UNION
SELECT DISTINCT 0, 0,
                0, 0,
                SUM(OT2.calc_rslt_val + OT2.calc_adj_val),
                SUM(OT2.unit_rslt_val + OT2.unit_adj_val)
  FROM Table1 AS A, Table1 AS OT1, Table1 AS OT2, Table2 AS B
WHERE -- OT2テーブルの選択条件のみが入る
```

その2

答え

この悪夢のような問題にはDDL文がないので、キーや制約、参照整合性制約、データ型などについては推測するしかない[訳注1]。このコードを読めるようきれいにするのは一苦労だった。次に、このコードで使われているロジックを観察して、どの部分が（SQLの計算では最も高コストである）直積の原因となっているかを調べた。ところで、

```
a = b, b = c, c = 2
```

という3つの式は、

```
a = 2, b = 2, c = 2
```

という形に還元できる。代数の基本だが、ここではこれを応用して、クロス結合を引き起こすだけで全く使われていないテーブルを削除しよう。次に示すのが、オリジナルのコードをきれいに掃除したバージョンである。

訳注1：参考までに、妥当と思われるテーブル定義を挙げておく。

```
CREATE TABLE Foobar
(empl_id  CHAR(6) NOT NULL, pin_nbr CHAR(5) NOT NULL,
 empl_rcd  INTEGER NOT NULL, calc_rslt_val INTEGER NOT NULL,
 calc_adj_val  INTEGER NOT NULL, unit_rslt_val INTEGER NOT NULL,
 unit_adj_val  INTEGER NOT NULL, PRIMARY KEY (empl_id, pin_nbr));
```

```
SELECT DISTINCT
       SUM(F1.calc_rslt_val + F1.calc_adj_val) AS calc_1,
       SUM(F1.unit_rslt_val + F1.unit_adj_val) AS unit_1,
       SUM(F2.calc_rslt_val + F2.calc_adj_val) AS calc_2,
       SUM(F2.unit_rslt_val + F2.unit_adj_val) AS unit_2,
       SUM(F3.calc_rslt_val + F3.calc_adj_val) AS calc_3,
       SUM(F3.unit_rslt_val + F3.unit_adj_val) AS unit_3
  FROM Foobar AS F1, Foobar AS F2, Foobar AS F3
 WHERE F1.empl_id = 'xxxxxx'
   AND F2.empl_id = 'xxxxxx'
   AND F3.empl_id = 'xxxxxx'
   AND F1.empl_rcd = 0
   AND F1.pin_nbr = '52636'
   AND F2.pin_nbr = '52751'
   AND F3.pin_nbr = '52768';
```

どれだけ読みやすくなったか、お分かりいただけるだろうか。SELECT句の計算列にも名前を付ける必要があることを忘れないように。

答え

その3

しかし、同じことをするなら、おそらく以下に示す方法が一番正しい。このクエリは、計算列の列展開を行わないようにしたものだ。もちろん、DDL文がないのだから、NULLやキー、および各列の意味については想像するしかない。

```
SELECT F1.pin_nbr,
       SUM(F1.calc_rslt_val + F1.calc_adj_val) AS calc_val,
       SUM(F1.unit_rslt_val + F1.unit_adj_val) AS unit_val
  FROM Foobar AS F1
 WHERE F1.empl_id = 'xxxxxx'
   AND F1.empl_rcd = 0
   AND F1.pin_nbr IN ('52636', '52751', '52768')
 GROUP BY F1.pin_nbr;
```

多層アーキテクチャでシステムを開発するときの原則を覚えているだろうか。データ表示のレイアウトは、バックエンドではなくフロントエンドでやること。複数の画面にまたがってデータを表示するための処理も、フロントエンドでやること。

このクエリは答えその2に比べ、少なくとも3倍の速度で動く。

パ|ズ|ル|
72

サービスマンの予約管理

時間データの扱い方（その2）

これは、サミーがSQL Serverのニュースグループに投稿した問題である。彼はテーブル設計について助言を求めていた。顧客のもとへ訪問する社員（サービスマン）の予約を登録するテーブルがあるのだが、彼らが休日に予約されることのないように修正したい、ということだった。

業務の流れは非常に明快だ。顧客からサービスマンに来てほしいと連絡が入ると、手配係が希望日時を聞いてシステムに入力するとともに、最初に都合のつくサービスマンを探して予約を入れる。顧客のもとを訪れたサービスマンは、そのまま30分単位でサービス時間を延長できる。

サミーが最初に出してきた設計案には、かなり問題があった。まず、彼はIDENTITYをキーとして使っていた（IDENTITYは自動採番を行うT-SQL独自の“機能”で、関係モデルとは無縁である）。さらに、彼はISO-11179の命名規則にも違反していた――サミーはテーブル名の後に「～table」と付けていたのだ（蛇足としか言いようがない！）。

彼の設計したテーブルはこうである。

```
CREATE TABLE CallsTable
(call_id INTEGER IDENTITY(1,1) NOT NULL PRIMARY KEY, --実装時の
 client_id INTEGER NOT NULL,                          データ型
 employee_id INTEGER NOT NULL,       -- 列名が長くて覚えにくい
 call_date SMALLDATETIME NOT NULL,   -- RDBMS依存のデータ型
 duration INTEGER NOT NULL,
 start_time SMALLDATETIME NOT NULL, -- RDBMS依存のデータ型
 end_time INTEGER NOT NULL);
```

CallsTable：顧客コールテーブル
client_id：顧客ID
call_date：顧客から連絡のあった日付
start_time：サービス開始時刻
end_time：サービス終了時刻

call_id：コールID
employee_id：社員ID
duration：サービス時間
start_time：サービス開始時刻

その1

答え　問題点はまだある。このテーブルには計算列が含まれているし、サミーはT-SQLのDATETIME型がどういう働きをするのかも理解していなかった。なぜ日付ないし時刻の列がINTEGER型で定義されているのだろう？ また、サービス時間（duration）列は全く不要だ。サービス時間は、開始時刻と終了時刻から計算できる。

また、終了した仕事についても保存する必要があったため、オリジナルのテーブルではダブルブッキングを許していた。その場合、終了時刻をNULLにしておき、後からCOALESCE関数を使って現在の日時に変換するのが正しい処理の方法である。

スキーマの残りの部分もいただけない。列が無意味に長いので、不正なデータが入り込むおそれがあるし、社員番号とは別にIDENTITY列を使って顧客からのコールに番号を振っている。テーブル名が奇妙なのは、彼がテーブルをファイルだと考えているからだろう。普通、何かの集合を考えるときには「社員（Personnel）」とか「従業員（Employees）」と言うのではないか？ これがテーブル名に集合名詞を使う理由だ[訳注1]。

このテーブルを再設計してみよう。標準SQL（T-SQLのDATETIME型はTIMESTAMP型になる。これはT-SQLのTIMESTAMP型とは異なる）とISO-11179の命名規則を使うと、次のようになる。

```
CREATE TABLE ScheduledCalls
(client_id INTEGER NOT NULL
   REFERENCES Clients (client_id),
 scheduled_start_time TIMESTAMP NOT NULL,
 scheduled_end_time TIMESTAMP NOT NULL,
   CHECK (scheduled_start_time < scheduled_end_time),
 emp_id CHAR(9) DEFAULT '{xxxxxxx}' NOT NULL
   REFERENCES Personnel (emp_id),
 PRIMARY KEY (client_id, emp_id, scheduled_start_time));
```

ScheduledCalls：手配済み顧客コールテーブル
scheduled_start_time：手配したサービス開始時刻
scheduled_end_time：手配したサービス終了時刻
client_id：顧客ID
emp_id：社員ID

訳注1：単数と複数の区別が曖昧な日本語ではニュアンスが伝わりにくいが、要するにテーブル名は常にそれが複数の物の集まりであることを示すような名前（Employeesのような普通名詞の複数形や、Personnelのような集合名詞）にすべきだ、ということである。

'{xxxxxx}'をダミーの社員番号として使っているが、これは手配係がサービスマンを検索したときに、どの社員かが必ず見つかるようにするためのものである。それゆえ、整合性制約が正しく動作するように、Personnelテーブルにダミー社員を表す値を入れておき、そのダミー社員は1日24時間×週7日間いつでも"対応可能"としておく必要がある。「{」と「}」の括弧は、画面上やレポートでダミー社員が常に一番最後に表示されるようにするための"トリック"だ。列にNULLを許可することでも同じことができるが、ここではこのプログラミングトリックを紹介しておきたい。

```
CREATE TABLE Clients
(client_id INTEGER NOT NULL PRIMARY KEY,
 first_name VARCHAR(15) NOT NULL,
 last_name VARCHAR(20) NOT NULL,
 phone_nbr CHAR(15) NOT NULL,
 phone_nbr_2 CHAR(15),
 client_street VARCHAR(35) NOT NULL,
 client_city_name VARCHAR(20) NOT NULL);
```

Clients：顧客テーブル	client_id：顧客ID
first_name：名	last_name：姓
phone_nbr：電話番号1	phone_nbr_2：電話番号2
client_street：住所（町名／番地）	client_city_name：住所（市町村名）

```
CREATE TABLE Personnel
(emp_id CHAR(9) NOT NULL PRIMARY KEY,
 first_name VARCHAR(15) NOT NULL,
 last_name VARCHAR(20) NOT NULL,
 home_phome_nbr CHAR(15) NOT NULL,
 cell_phone_nbr CHAR(15) NOT NULL,
 street_addr VARCHAR(35) NOT NULL,
 city_name VARCHAR(20) NOT NULL,
 zip_code CHAR(5) NOT NULL);
```

Personnel：社員テーブル	emp_id：社員ID
first_name：名	last_name：姓
home_phome_nbr：自宅の電話番号	cell_phone_nbr：携帯電話番号
street_addr：住所（町名／番地）	city_name：住所（市町村名）
zip_code：郵便番号	

```
CREATE TABLE Services
(client_id INTEGER NOT NULL REFERENCES Clients,
 emp_id CHAR(9) NOT NULL REFERENCES Personnel,
 start_time TIMESTAMP NOT NULL,
 end_time TIMESTAMP,  -- まだ仕事中の場合はNULL
  CHECK (start_time < end_time),
 sku INTEGER NOT NULL,
 PRIMARY KEY (client_id, emp_id, start_time, sku));
```

Services：サービステーブル	client_id：顧客ID	emp_id：社員ID
start_time：サービス開始時刻	end_time：サービス終了時刻	sku：サービス番号

Servicesテーブルの長い自然キー[訳注2]に注目してほしい。主キーをこのように宣言しないと、かえってデータの整合性が失われてしまう。だが、新米エンジニアは長い自然キーを使うのを怖がり、データの整合性の心配をしなくて済むと思って、IDENTITYのような機能をキーに使う傾向にある。

ここで本当に重要なトリックは、1人1人のサービスマンが対応可能なすべての日付を保持する「社員スケジュール」テーブルを作ることにある。

```
CREATE TABLE PersonnelSchedule
(emp_id CHAR(9) NOT NULL
   REFERENCES Personnel(emp_id),
 avail_start_time TIMESTAMP NOT NULL,
 avail_end_time TIMESTAMP NOT NULL,
  CHECK (avail_start_time < avail_end_time),
 PRIMARY KEY (emp_id, avail_start_time));
```

PersonnelSchedule：社員スケジュールテーブル	emp_id：社員ID
avail_start_time：空き時間の時刻開始	avail_end_time：空き時間の終了時刻

訳注2：システムで扱うデータ項目から構成されるキー。逆に、データの識別だけを目的に作成されるキーを「代理キー（surrogate key）」と言う。自然キーと代理キーの設計理論上の優劣については賛否両論あるが、著者は代理キーや物理記憶のロケーターを主キーとして使うことに、極めて批判的なようだ。その理由は、これらがデータの属性ではないため、関係モデルの原理に反するということによる（この見解を支持する理論家は少なくない）。

その2

答え　私たちは、顧客が来てほしいと言う時間に訪問可能なサービスマンを見つけなければならない。次のクエリでは、「訪問可能な時間がサービスコールの期間と重複している、あるいは完全に含んでいること」という条件を指定している。先ほど作成したダミー社員（{xxxxxxx}）は、手配係が主キーと外部キーのリレーションシップで訪問可能なサービスマンを見つけるための便利なトリックとして、ここで活きてくる。

```
SELECT P.emp_id,
       S.client_id,
       S.scheduled_start_time,
       S.scheduled_end_time
  FROM ScheduledCalls AS S,
       PersonnelSchedule AS P
 WHERE S.emp_id = P.emp_id
   AND P.emp_id <> '{xxxxxxx}'
   AND S.scheduled_start_time
       BETWEEN P.avail_start_time AND P.avail_end_time
   AND S.scheduled_end_time
       BETWEEN P.avail_start_time AND P.avail_end_time;
```

このクエリで注意すべき点は、このクエリが訪問可能な全サービスマンを選択することだ。ただ、これは手配係が誰かを選んで仕事を割り当てれば済む話である。

パ|ズ|ル
73 データのクリーニング

COALESCE関数で擬似配列を扱う

この問題は最初、SQLServerCentral.comで「スタンジ」という人物がデータクリーニングの問題として提供したものだ。彼は、「全列の値がNULLである行」を含むデータソースからデータをインポートしていた。残念なことに、データソース側で「オールNULL」の行を取り除くことはできない。そこで彼は、このデータをデータベースへインポートしたあとでオールNULLの行を判別し、除外しようと考えた。ただ、彼が恐れていたのは、次のように条件式をハードコーディングする必要があるのではないか、ということだった。

```
SELECT *
  FROM Staging
 WHERE col1 IS NULL
   AND col2 IS NULL
   AND col3 IS NULL
   ..
   AND col100 IS NULL;
```

答え

その1

スタンジは<テーブル名>を引数として渡して、スキーマ情報テーブルからそのテーブルの列名リストを取得する方法を考えていた。そのリストを使って、全列の値がNULLである行を検索するクエリを作ろうというわけである。

スキーマ情報を得るためのクエリは、SQL Serverでは次のようなものになる。このクエリはRDBMSによって微妙に異なってくる。

```
SELECT *
  FROM syscolumns
WHERE id
      = (SELECT id
           FROM sysobjects
          WHERE name = <テーブル名>);
```

その2

答え　クリス・ティーターとジェスパーの2人は、カーソルでスキーマ情報テーブルを走査し、動的SQLを組み立てて実行するという解答を提案した。だが、これはRDBMSにどっぷり依存するし、処理もRDBらしくない。おまけに、実行速度がとても遅い。

そんなわけで、ここには彼らのコードは載せないが、このことは、プログラマがいかに手続き型の考え方に傾きがちかを物語っている。

その3

答え　あるテーブルの列名をすべて取得したいなら、システムユーティリティ関数を使えばよい。どのRDBMSでも、そういう関数を持っている（例えば、SQL Serverなら EXEC sp_columns）。あとは、それを次に示すSQL文のCOALESCE関数に入れるだけだ。

```
DELETE FROM Staging
 WHERE COALESCE(col1, col2, col3, .., col100) IS NULL;
```

このSQL文は可能な限り高速に動く。この問題のもう1つの教訓は、マニュアルを読み、現在使用しているRDBMSが提供する機能を理解しておくことは大切だ、ということである。こうしたユーティリティの大半は、行についての列定義およびオプション（NULLを許可するか、DEFAULT値は何か、キーやインデックス等々）を与えてくれる。だから、ユーザはただ列名を切り出し、後ろにカンマを付けるだけでよい。

このクエリを書くのに、大きなテーブルでも5秒とかからない。だが、使っているデータベースの新しいバージョンがリリースされて、スキーマ情報テーブルが微妙に変化したりすると、古いコードが失敗するかもしれない。その場合は、コードを書き直すためにもう少し時間がかかるだろう。

しかし、テーブルに変更が生じたり、このシステムの新しいバージョンがリリースされたりしてSQL文が動かなくなるたびにそれを書き直さねばならない、という点は忘れないように。そうしたことは、スキーマ情報テーブルに変更が生じるよりも頻繁に起こるだろう。

パ|ズ|ル|
74 導出テーブルを減らせ
複数の外部結合を効果的に使う

アレン・ディヴィドソンは、3つのテーブルを結合し、そのいくつかの列の合計値(SUM)を取得するために、2つの左外部結合を使おうとしていた。彼のクエリを、なるべく導出テーブルを使わないように書き換えることはできるだろうか。

```
CREATE TABLE Accounts
(acct_nbr INTEGER NOT NULL PRIMARY KEY);
```

Accounts：計算書テーブル	acct_nbr：計算書番号

```
INSERT INTO Accounts VALUES(1), (2), (3), (4);
```

次のFooおよびBarテーブルには主キーがないので、どちらも厳密にはテーブルでないことに注意してほしい。

```
CREATE TABLE Foo
(acct_nbr INTEGER NOT NULL
   REFERENCES Accounts(acct_nbr),
 foo_qty INTEGER NOT NULL);
```

Foo：Fooテーブル	acct_nbr：計算書番号	foo_qty：Fooでの数量

```
INSERT INTO Foo VALUES (1, 10);
INSERT INTO Foo VALUES (2, 20);
INSERT INTO Foo VALUES (2, 40);
INSERT INTO Foo VALUES (3, 80);
```

```
CREATE TABLE Bar
(acct_nbr INTEGER NOT NULL
   REFERENCES Accounts(acct_nbr),
 bar_qty INTEGER NOT NULL);
```

Bar：Barテーブル	acct_nbr：計算書番号	bar_qty：Barでの数量

```
INSERT INTO Bar VALUES (2, 160);
INSERT INTO Bar VALUES (3, 320);
INSERT INTO Bar VALUES (3, 640);
INSERT INTO Bar VALUES (3, 1);
```

彼が送ってきたクエリは次のとおりだ。FROM句で、導出テーブル2つを左外部結合している。

```
SELECT A.acct_nbr,
       COALESCE(F.foo_qty, 0) AS foo_qty_tot,
       COALESCE(B.bar_qty, 0) AS bar_qty_tot
  FROM Accounts AS A
       LEFT OUTER JOIN
       (SELECT acct_nbr, SUM(foo_qty) AS foo_qty
          FROM Foo
         GROUP BY acct_nbr) AS F
       ON F.acct_nbr = A.acct_nbr
       LEFT OUTER JOIN
       (SELECT acct_nbr, SUM(bar_qty) AS bar_qty
          FROM Bar
         GROUP BY acct_nbr) AS B
       ON F.acct_nbr = B.acct_nbr;

acct_nbr   foo_qty_tot   bar_qty_tot
====================================
       1            10             0
       2            60           160
       3            80           961
       4             0             0
```

これはこれでうまくいくのだが、ほかのやり方を考えてほしい。

その1

答え R.シャーマは、導出テーブルを1つ減らす方法を見つけたが、2つともなくすことはできなかった。

```
SELECT A.acct_nbr,
       COALESCE(SUM(F.foo_qty), 0) AS foo_qty_tot,
       COALESCE(MAX(B.bar_qty), 0) AS bar_qty_tot
  FROM (SELECT * FROM Accounts) AS A
```

```
        LEFT OUTER JOIN
       (SELECT * FROM Foo) AS F
        ON A.acct_nbr = F.acct_nbr
        LEFT OUTER JOIN
       (SELECT acct_nbr, SUM(bar_qty) AS bar_qty
          FROM Bar
         GROUP BY acct_nbr) AS B
        ON A.acct_nbr = B.acct_nbr
GROUP BY A.acct_nbr;
```

このクエリは、導出テーブルBの結果が1つのacct_nbrにつき必ず1行になり、さらに3行目のMAX関数が正しい値を保証しているので、求める結果をきちんと返してくれる。また、Accountsテーブルと Fooテーブルが1対多の関係にあり、Accounts.acct_nbrでグループ化 (GROUP BY A.acct_nbr) しているので、アレンのSELECT文にある1つ目の導出テーブルを不要にできた。

答え

その2

次に、私の答えをお見せしよう。まずは、テーブルではなかったFooとBarをAccountsテーブルと外部結合することにより、れっきとしたテーブルが得られる。その後に計算用の情報を付け加える。OUTER UNIONオプションが使えるともっと簡単に書けるのだが、残念なことに、このオプションはまだほとんどのRDBMSでサポートされていない。そのため、UNIONと外部結合を別々に使う必要がある。

```
SELECT acct_nbr,
       COALESCE (SUM(foo_qty), 0) AS foo_qty_tot,
       COALESCE (SUM(bar_qty), 0) AS bar_qty_tot
  FROM ((SELECT A1.acct_nbr, foo_qty, bar_qty
           FROM Accounts AS A1
                LEFT OUTER JOIN
                (SELECT acct_nbr, foo_qty, 0 AS bar_qty
                   FROM Foo) AS F
                ON A1.acct_nbr = F.acct_nbr) -- Fooのデータ
        UNION ALL
        (SELECT A2.acct_nbr, foo_qty, bar_qty
           FROM Accounts AS A2
                LEFT OUTER JOIN
                (SELECT acct_nbr, 0 AS foo_qty, bar_qty
                   FROM Bar) AS B
                ON A2.acct_nbr = B.acct_nbr) -- Barのデータ
```

```
        ) AS X (acct_nbr, foo_qty, bar_qty) -- FooとBarが1つに
GROUP BY acct_nbr;
```

FooとBarをマージするUNION ALLはソートが不要なので高速に動くだろうし、FooとBarのクエリは並行処理が可能だ。そのため、全体として、かなり良好なパフォーマンスを期待できる。

パズル 75

もう1軒行こう

座標と距離を扱う

地図を使ったよくある問題を出そう。もともとのバージョンは町中の居酒屋の場所を使っていて、1軒の居酒屋から追い出されたときに、近くにある次の店へたどり着く方法を見つけるというものだった。

地図には、デカルト座標系 (x, y) を使う。これは次のようなテーブルで表される。

```
CREATE TABLE PubMap
(pub_id CHAR(5) NOT NULL PRIMARY KEY,
 x INTEGER NOT NULL,
 y INTEGER NOT NULL);
```

PubMap：居酒屋マップテーブル	pub_id：居酒屋ID
x：x座標	y：y座標

求めたいのは、近くにある別の居酒屋を見つけるための効率的な方法である。

その1

答え

直接的な解答としては、デカルトの距離の公式 $d = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$ を使うものがある。これによって近傍[訳注1]の座標を、追い出された居酒屋から特定の半径を持った範囲として定義できる。

```
SELECT B.pub_id, B.x, B.y
  FROM PubMap AS A,
       PubMap AS B
 WHERE A.pub_id = :my_pub
   AND :my_pub <> B.pub_id
   AND SQRT (POWER((A.x - B.x), 2) + POWER((A.y - B.y), 2))
       <= :crawl_distance;
```

だが、計算部分をよく調べると、処理のコストを多少節約できることが分かる。両辺とも2乗すればよいのだ。

訳注1：「近傍 (neighborhood)」は、任意の半径を持つ円内に含まれる点全体の集合を意味する幾何学の用語。

```
SELECT B.pub_id, B.x, B.y
  FROM PubMap AS A,
       PubMap AS B
 WHERE :my_pub = A.pub_id
   AND :my_pub <> B.pub_id
   AND (POWER((A.x - B.x), 2) + POWER((A.y - B.y), 2))
       <= POWER(:crawl_distance, 2);
```

数の2乗は整数の掛け算として実行できるため、大抵の場合は非常に高速だ。

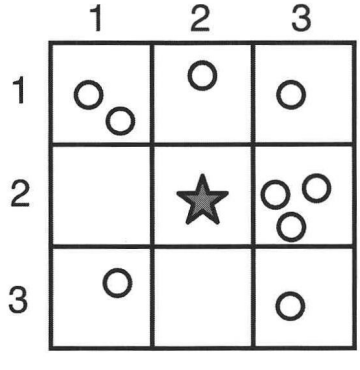
その2

答え 直線距離（円近傍モデル）を使わなくてかまわないなら、代わりに平方近傍を使うことで、演算が簡単になる。

```
SELECT A.pub_id, B.pub_id
  FROM PubMap AS A, PubMap AS B
 WHERE :my_pub <> B.pub_id
   AND :my_pub = A.pub_id
   AND ABS(A.x - B.x) <= :distance
   AND ABS(A.y - B.y) <= :distance;
```

その3

答え 平方近傍の方法からヒントを得た別解をお見せしよう。これは、平面を巨大な正方形の格子に分割する方法だ。各格子のマスは、いくつかのノード（居酒屋）を含む。要するに、ごく普通の地図帳のマス目をイメージしてほしい。



★は現在地のパブ、○は周辺のパブを示す。
1つのパブは、必ず1つのマスに含まれる

```
CREATE TABLE PubMap
(pub_id CHAR(5) NOT NULL PRIMARY KEY,
 x INTEGER NOT NULL,
 y INTEGER NOT NULL,
 cell_x INTEGER NOT NULL,
 cell_y INTEGER NOT NULL);
```

PubMap：居酒屋マップテーブル	pub_id：居酒屋ID
x：x座標	y：y座標
cell_x：マス目の行	cell_y：マス目の列

この方法では、マスの位置を示す余計な座標を保持することになるが、すべてのノードを検索する必要はない。マップ全体に15万ノード存在していたとしても、実際に検索対象となるのは近隣9マスのノードだけで済む[訳注2]。

```
SELECT N2.pub_id, N2.x, N2.y
  FROM PubMap AS N1, PubMap AS N2
 WHERE :my_pub <> N2.pub_id
   AND :my_pub =  N1.pub_id
   AND N2.cell_x IN (N1.cell_x-1, N1.cell_x, N1.cell_x+1)
   AND N2.cell_y IN (N1.cell_y-1, N1.cell_y, N1.cell_y+1);
```

最初のクエリでBテーブルの代わりに、このSELECT文を制限された近隣のノードを持つ導出テーブルとして使えば、地図が大きくなったときにも高いパフォーマンスが期待できる。

訳注2：クエリの条件にあるように、検索範囲のx座標はcell_x－1、cell_x、cell_x＋1の3つ、y座標も同様にcell_y－1、cell_y、cell_y＋1の3つなので、検索範囲は必ず9マスに限定される。

Index

た行

な行

は行

ま行

や行

ら行

わ行

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1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 for Smarties: Advanced SQL Programming, Third Edition』『Joe Celko's Trees and Hierarchies in SQL for Smarties』（ともにMorgan Kaufmann刊）など。

訳者プロフィール

■ミック

金融系SI企業に勤務するDBエンジニア。主にOracleを使ったデータウェアハウス業務に従事している。（株）翔泳社が主宰するWebマガジン「CodeZine（http://codezine.jp/）」にて、SQLに関する記事を連載中。自身のホームページ「リレーショナル・データベースの世界（http://www.geocities.jp/mickindex/database/idx_database.html）」では、SQLとデータベース技術についての情報を公開している。最近は、集合指向というSQLの特性を活かした技術の応用や、木構造データをSQLで扱う技術に関心がある。

DTP　有限会社風工舎
装丁　轟木 亜紀子（株式会社トップスタジオ）

SQL（エスキューエル）パズル 第2版（だいはん） ～ プログラミングが変（か）わる書（か）き方（かた）／考（かんが）え方（かた）

2007年11月1日　初版第1刷発行

著　者　Joe Celko（ジョー・セルコ）
訳　者　ミック
発行人　佐々木 幹夫
発行所　株式会社翔泳社（http://www.shoeisha.co.jp）
印刷・製本　大日本印刷株式会社

ISBN978-4-7981-1413-2　Printed in Japan

ISBN978-4-7981-1413-2
C3055 ¥2800E

株式会社翔泳社
定価：本体2,800円+税

地球にやさしい本づくりを目指します

SQLパズル 第2版

プログラミングが変わる書き方／考え方

SQL Puzzles and Answers 2nd Edition