特集2

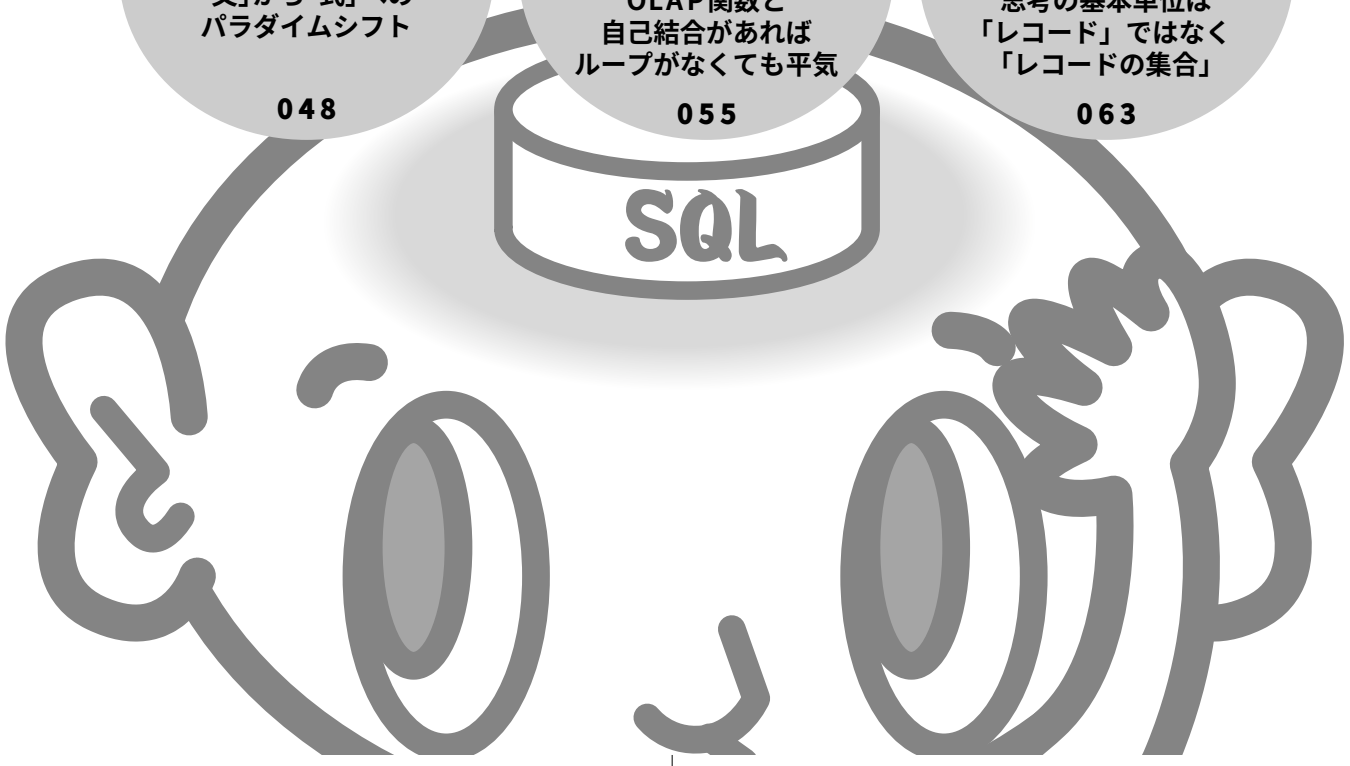
SQLアタマ養成講座

分岐とループ、集合操作…
プログラミング言語とはここが違う！

■ミック http://www.geocities.jp/mickindex/

本特集では、SQLの基礎は一通り理解しており、日々の業務でもSQLを利用している。でも何となくSQLの考え方がしっくりこない…そんな現場のエンジニアに向けて、SQL特有の原理を明らかにし、より一層SQLを使いこなしていく指針をまとめていきます。

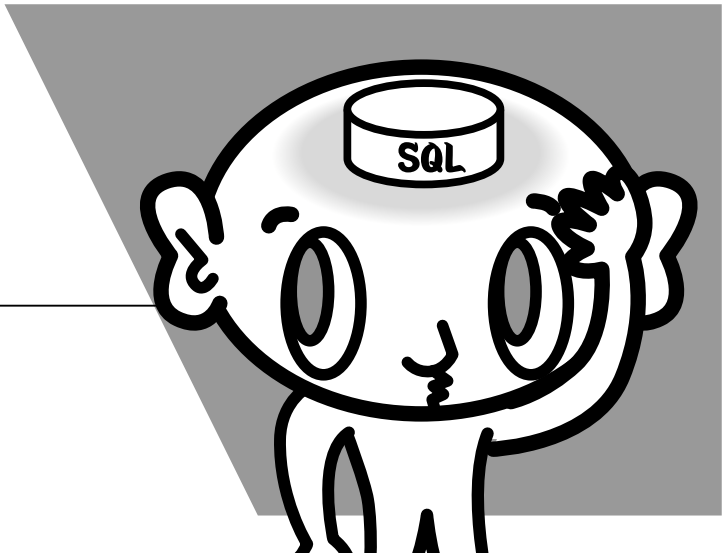
全章をとおして、C言語やPerl、Javaといった手続き型言語、あるいは手続き型に基礎を持つ言語の考え方とSQLのそれを対比して、両者の共通点と相違点を描き出します。分岐とループという手続き型になくてはならない機能をSQLで実現するにはどうするのかという点を中心に、わかりやすく解説しています。



第1章

「文」から「式」へのパラダイムシフト

SQL流条件分岐



はじめに

私たちが通常、C言語やPerl、Javaなどの手続き型言語（またそれに基礎を持つ言語）を使ってプログラミングを行う場合、最も多用する基本的な制御構造が分岐とループです。この2つを使わずにプログラミングしろ、と言われたら、それはかなりきつい制約になるでしょう。腕試しや暇つぶしに試すにはおもしろいかもしれませんが、およそ実務的なコーディングは不可能になるに違いありません。

話は、SQLとデータベースの場合でも同じです。SQLにおいても、やはり分岐とループは非常に重要な役割を果たす機能であり、SQLプログラミングの際にこの2つの機能を欠かすことはできません。しかしながら、手続き型言語を使いこなすプログラマの多くが、なぜかSQLを使う段になると思い通りの制御構造を記述できないことに苛立ちを感じ、結果、非効率的なSQL文が多く生み出されています。これはなぜでしょう？

SQLで分岐とループを表現すること自体は、何の問題もなく可能なのです。通常の手続き型言語で表現可能なアルゴリズムはすべてSQLでも表現可能なことが知られています。それなのに、なぜ多くのプログラマやエンジニアがSQLに戸惑いを感じるかといえば、その処理の基本単位の違いに無頓着なままだからです。

手続き型言語が「文（statement）」を基本単位として分岐やループを記述するのに対し、SQLの基本は「式（expression）」です。これがSQLが「宣言的」と呼ばれる特徴の1つなのですが、長い間手続き型の考え方に親しんできたエンジニア（私たちのほとんどすべて、ということですが）は、この点に無自覚なままSQLを扱い、その力を十全に引き出すことができないままフラストレーションを溜めます。これは、私たちにとってもSQLにとっても不幸なことです。

そこで本章ではまず、SQLにおいて分岐を表現する強力な道具である「CASE式」について学び、その使い方をマスターしていきたいと思います（ループについては、次章で取り上げます）。

なお、本特集で取り上げているSQLは、SQL92/99/2003準拠のもので、RDBMSはPostgreSQL 8.3、MySQL 5.0、Oracle 10g リリース2（10.2）、DB2 9.1、SQL Server 2005を対象としています。また、OLAP関数は標準SQLの新しい機能であり、PostgreSQLとMySQLではまだサポートされていません。

❶ウォーミングアップ 条件に応じて使う列を切り替える

CASE式は、SQLにおいて条件分岐を記述するために導入された非常に重要な機能です。その名前が示すとおり「式」であるため、SQLの実行時には評価されて単一のスカラ値[注1]に定まるところに特徴があります。C言語やVBなどのCASE文と見た目が似ているので同じような感覚で使われることがありますが（特に終端子の「END」が、一連の手続きの終わりを示すように錯覚しやすい）、それではCASE式の強みを十分に引き出せません。

CASE式の特性を理解するために、ちょっとした練習から始めましょう。表1のサンプルテーブルを使います。

注1）文字列値、整数値などの単一値。

ChangeColsは年単位で何らかの数値を管理しているテーブルですが、具体的に数値が何を意味するかは今は気にしないでください。このテーブル自体に特におかしなところはないのですが、ある日、あなたのもとに妙な要件が持ち込まれてきます。それは、2000年まではcol_1の値を使って、2001年からはcol_2の値を使って集計をしたい、というものです。求める結果は図1のような形です。

いわば、行によって使う列を変えて1列にまとめる、というイメージです。このテーブルがファイルで、手続き型言語で1行ずつ読み出すのであれば、各行のyear列の値によって条件を分岐させ、使うフィールドを変える、ということになるでしょう。SQLでもその考え方は変わりません。リスト1のように記述します。

条件を記述するWHEN句は手続き型言語と同じですが、CASE式に特徴的なのは、THENのあとの実行部です。C言語などでは、ここで変数new_colにcol_1（またはcol_2）の値を「代入」する文を記述しますが、SQLにおいてはcol_1、col_2を直接に戻り値としています。あたかも「2つで1つの列」のような返し方をするわけです。

式は列や定数を記述できるところには常に記述できますから、リスト2、図2のようにWHERE句で利用することも問題なく可能です。

これは、先ほどのクエリで作ったnew_colの値が4以上の年度を選択するクエリです。右辺のCASE式は、一見すると値には見えませんが、ちゃんと実行時には評価されて「10」や「3」などの単一の値（＝スカラ値）になります。だからこそ、このように比較述語の引数に取ることも可能なのです。

❷列の交換
順列と組み合わせ

2問目は、「列の切り替え」問題の発展版を練習してみましょう。

ある店舗で、売れた商品の明細を顧客ごとに記録する表2のようなテーブルがあるとします。列持ち形式のため、定義されている列数以上の商品が売れたときは記録できないという欠点を抱える設計ですが、最初はその点には目をつぶります（あとでこの欠点に対する解決策も示します）。

さて、このテーブルを使って、一緒に売れた商品のペアを単位としていろいろな分析を行うことを考えます。その際、このテーブルのままだと不都合なことが起きます。顧客IDが001と003の行を見れば

●リスト1　列の切り替え（SELECT句で）

```
SELECT year,
       CASE WHEN year <= 2000 THEN col_1
            WHEN year >= 2001 THEN col_2
            ELSE NULL END AS new_col
  FROM CahngeCols;
```

●リスト2　WHERE句での利用

```
SELECT year
  FROM CahngeCols
 WHERE 4 <= CASE WHEN year <= 2000 THEN col_1
                 WHEN year >= 2001 THEN col_2
                 ELSE NULL END;
```

●表1　ChangeColsテーブル

year	col_1	col_2
1998	10	7
1999	20	6
2000	30	5
2001	40	4
2002	50	3

●図1　求める結果

```
year  new_col
----- -------
1998  10      ← col_1
1999  20      ← col_1
2000  30      ← col_1
2001   4      ← col_2
2002   3      ← col_2
```

●図2　リスト2の実行結果

```
year
----
1998
1999
2000
2001
```

●表2　Perm2テーブル

cust_id（顧客ID）	item_1（商品1）	item_2（商品2）
001	時計	浄水器
002	携帯電話	携帯電話
003	浄水器	時計
004	携帯電話	携帯電話
005	インク	メガネ

わかるように、この2人は、商品の組み合わせとしては {時計、浄水器} という同じペアを買っています。しかしおそらく、テーブルに記録されるときは買った順番で記録したためでしょう、テーブル上では順序を入れ替えた、異なるペアとして存在しています。これでは、単純にitem_1、item_2を「SELECT DISTINCT」で選択したり、GROUP BY句で集約したとしても、正しい商品のペアが求められません。

そこで、表2のようなテーブルから、表3のように商品の並び順を無視した組み合わせを求めましょう。

言い方を変えると、順序を意識した「順序集合」を、順序を無視した「非順序集合」へ変換する、ということです。あるいは、学校で習った馴染み深い表現を使うなら、順列（Permutation）を組み合わせ（Combination）へ変換するのです。

CASE式を使えばこんなことも朝飯前です。リスト3のクエリを見てください（図3は実行結果）。

item_1とitem_2は文字列ですので、不等号で比較可能な順序を持っています。ということは、最初の列「c1」に小さいほう、2番目の列「c2」に大きいほうを配置してやれば、同じ要素を持っていて順序だけ異なる順序集合同士も、同じ並び順に配置し直すことができるわけです。あとは、重複行を排除してできあがり（リスト4、図4）。

この方法は、比較したい列が文字型でも数値型でも日付型でも、とにかく順序づけられるデータ型ならば適用できる汎用性の高い方法です。

ではこの問題を一般化します。商品列を3列に増やした場合（表4）はどうなるでしょう。

やりたいことは同じなので、求めたい結果は表5のようなものになります。

この場合も、原理的には2列の場合と同じように、

●リスト3　組み合わせ⇒順列（重複行排除前）

```
SELECT
       CASE WHEN item_1 < item_2 THEN item_1
            ELSE item_2 END AS c1,
       CASE WHEN item_1 < item_2 THEN item_2
            ELSE item_1 END AS c2
  FROM Perm2;
```

●図3　リスト3の実行結果

```
   c1         c2
--------- ---------
時計       浄水器
携帯電話   携帯電話
時計       浄水器
携帯電話   携帯電話
インク     メガネ
```

●リスト4　組み合わせ⇒順列（重複行排除後）

```
SELECT DISTINCT
       CASE WHEN item_1 < item_2 THEN item_1
            ELSE item_2 END AS c1,
       CASE WHEN item_1 < item_2 THEN item_2
            ELSE item_1 END AS c2
  FROM Perm2;
```

●図4　リスト4の実行結果

```
   c1         c2
--------- ---------
インク     メガネ
携帯電話   携帯電話
時計       浄水器
```

●表3　商品の並び順を無視した組み合わせ

item_1 (商品1)	item_2 (商品2)
時計	浄水器
携帯電話	携帯電話
インク	メガネ

●表4　Perm3テーブル

cust_id (顧客ID)	item_1 (商品1)	item_2 (商品2)	item_3 (商品3)
001	時計	浄水器	ティッシュ
002	ティッシュ	浄水器	時計
003	カレンダー	ノート	時計
004	カレンダー	ノート	インク
005	文庫本	ゲームソフト	メガネ
006	文庫本	メガネ	ゲームソフト

●表5　求めたい結果

item_1 (商品1)	item_2 (商品2)	item_3 (商品3)
インク	カレンダー	ノート
カレンダー	ノート	時計
ゲームソフト	メガネ	文庫本
ティッシュ	時計	浄水器

不等号の比較条件をCASE式で記述することができるのですが、条件がかなり複雑になりますし、4列、5列…と増えていった場合にはさらに厳しくなります。ちょっと読むに堪えないクエリになるでしょう。

こういう一般化したケースをなんとかしたいという相談が持ち込まれてきたとしたら、私ならリスト5、図5のように列持ちの形式を行持ちの形式に直す方法を奨めます。

いったんこの形式に直してしまえば、あとはリスト6のクエリで組み合わせへ変換できます（実行結果は図6）。

やはり不等号を使って商品ごとに順序づけを行う、という点はCASE式のときと同じですが、MIN関数を併用することで比較条件を非常に簡潔なものに抑えることができるのです。

このクエリの意味は、次のようなものです。

①まず1人の顧客について、3つの商品の中から最小値を選択する（MIN(CI1.item)）
②次に、その最小値を除いた集合から最小値を選択する（MIN(CI2.item)）
③最後に、その値も除いた集合から最小値（最後なので1つしか残っていないが）を選択する（MIN(CI3.item)）

これなら、商品数が何列に増えても対応できます（ただし、この方法では、同じ商品の組み合わせは選択できなくなります。これに対応する方法は、みなさんも考えてみてください）。このことからテーブル設計においては、例外的なケースを除いて、なるべく列持ちよりは行持ちの形式を採用したほうが拡張的な要件にも耐える、という教訓が得られます。安易に配列やフラットファイルをそのままテーブルの「列」に写し取る無芸な設計をしてしまうと、あとあとまで大きな禍根を抱え込むことになるので、注意が必要です。

❸表頭の複雑な集計

これはCASE式の使い方の中で最も使い勝手がよ

●リスト5　列持ち⇒行持ち

```
CREATE VIEW CustItems (cust_id, item) AS
SELECT cust_id, item_1
  FROM Perm3
UNION ALL
SELECT cust_id, item_2
  FROM Perm3
UNION ALL
SELECT cust_id, item_3
  FROM Perm3;
```

●図5　リスト5の実行結果

```
cust_id  item
-------- ----------
001      浄水器
001      時計
001      ティッシュ
002      浄水器
002      ティッシュ
002      時計
003      ノート
003      カレンダー
003      時計
004      ノート
004      カレンダー
004      インク
005      ゲームソフト
005      文庫本
005      メガネ
006      文庫本
006      ゲームソフト
006      メガネ
```

●リスト6　組み合わせ⇒順列（3列拡張版）

```
SELECT DISTINCT MIN(CI1.item) AS c1,
       MIN (CI2.item) AS c2,
       MIN (CI3.item) AS c3
  FROM CustItems CI1
          INNER JOIN CustItems CI2
                  ON CI1.cust_id = CI2.cust_id
                 AND CI1.item < CI2.item
                INNER JOIN CustItems CI3
                        ON CI2.cust_id = CI3.cust_id
                       AND CI2.item < CI3.item
 GROUP BY CI1.cust_id;
```

●図6　リスト6の実行結果

```
     c1           c2       c3
------------ ---------- -------
インク       カレンダー ノート
カレンダー   ノート     時計
ゲームソフト メガネ     文庫本
ティッシュ   時計       浄水器
```

く応用範囲の広い技術なので、ぜひマスターしてください。表6のような、性別・年齢・部署別の給与を管理する人事テーブルがあるとします。

ここからいろいろな組み合わせのクロス表を作ってみましょう。こういう要件は実務の中でも頻繁に発生すると思いますが、まずは表7のような表頭が年齢階級・性別、表側が部署で、人数を集計した表です。いま年齢階級は便宜的に30歳以下を「若手」、それ以上を「ベテラン」という簡単な区分にしておきます。

❶「ウォーミングアップ」でも見たように、CASE式をSELECT句で使うことによって、条件に応じて集計したい行を指定できます。すると、表頭の4列をCASE式で作ることが可能になります（リスト7）。

CASE式の戻り値を0/1で指定しているのは、各行に対してビットフラグを立てていると思えばわかりやすいでしょう。あとはこのフラグが1の行数をSUM関数で数えることで、条件に合致した行数だけをカウントできるのです（このトリックは第3章でもう一度見ることになります）。

では、表頭に小計・合計の列も追加して、表8の場合はどうでしょう。

これも、リスト7のクエリを簡単に修正するだけで対応できます（リスト8）。

●表6 Employeesテーブル

emp_id（社員ID）	dept（部署）	sex（性別）	age（年齢）	salary（給与：万円）
001	製造	男	32	30
002	製造	男	30	29
003	製造	女	23	19
004	会計	男	45	35
005	会計	男	50	45
006	営業	女	40	50
007	営業	女	42	40
008	営業	男	52	38
009	営業	男	34	28
010	営業	女	41	25
011	人事	男	29	25
012	人事	女	36	29

●表7 表頭：年齢階級・性別、表側：部署

	若手		ベテラン	
	男	女	男	女
製造	1	1	1	0
会計	0	0	2	0
営業	0	0	2	3
人事	1	0	0	1

●表8 表頭：年齢階級・性別 表側：部署（小計・合計あり）

	合計	若手			ベテラン		
		計	男	女	計	男	女
製造	3	2	1	1	1	1	0
会計	2	0	0	0	2	2	0
営業	5	0	0	0	5	2	3
人事	2	1	1	0	1	0	1

●リスト7 表頭：部署・性別、表側：年齢階級

```
SELECT dept,
       SUM(CASE WHEN age <= 30 AND sex = '男' THEN 1 ELSE 0 END) AS "若手（男）",
       SUM(CASE WHEN age <= 30 AND sex = '女' THEN 1 ELSE 0 END) AS "若手（女）",
       SUM(CASE WHEN age >= 31 AND sex = '男' THEN 1 ELSE 0 END) AS "ベテラン（男）",
       SUM(CASE WHEN age >= 31 AND sex = '女' THEN 1 ELSE 0 END) AS "ベテラン（女）"
  FROM Employees
 GROUP BY dept;
```

●リスト8 表頭：年齢階級・性別、表側：部署（小計・合計あり）

```
SELECT dept,
       COUNT(*),
       SUM(CASE WHEN age <= 30 THEN 1 ELSE 0 END) AS "若手（計）",
       SUM(CASE WHEN age <= 30 AND sex = '男' THEN 1 ELSE 0 END) AS "若手（男）",
       SUM(CASE WHEN age <= 30 AND sex = '女' THEN 1 ELSE 0 END) AS "若手（女）",
       SUM(CASE WHEN age >= 31 THEN 1 ELSE 0 END) AS "ベテラン（計）",
       SUM(CASE WHEN age >= 31 AND sex = '男' THEN 1 ELSE 0 END) AS "ベテラン（男）",
       SUM(CASE WHEN age >= 31 AND sex = '女' THEN 1 ELSE 0 END) AS "ベテラン（女）"
  FROM Employees
 GROUP BY dept;
```

●リスト9　全列をCOUNT関数で揃える

```
SELECT dept,
       COUNT(*),
       COUNT(CASE WHEN age <= 30 THEN 1 ELSE NULL END) AS "若手（計）",
       COUNT(CASE WHEN age <= 30 AND sex = '男' THEN 1 ELSE NULL END) AS "若手（男）",
       COUNT(CASE WHEN age <= 30 AND sex = '女' THEN 1 ELSE NULL END) AS "若手（女）",
       COUNT(CASE WHEN age >= 31 THEN 1 ELSE NULL END) AS "ベテラン（計）",
       COUNT(CASE WHEN age >= 31 AND sex = '男' THEN 1 ELSE NULL END) AS "ベテラン（男）",
       COUNT(CASE WHEN age >= 31 AND sex = '女' THEN 1 ELSE NULL END) AS "ベテラン（女）"
  FROM Employees
 GROUP BY dept;
```

合計列のCOUNT(*)はSUM(1)としても同じです。あるいはそう書いたほうが「無条件に行数をカウントしている」という意味が明確になるかもしれませんが、一般的にあまり見ない書式なので初めて見た人が驚くかもしれません。

また、リスト9のように、全列をCOUNT関数で揃えてもかまいません。その場合は、ELSE句で返すときにNULLを指定する必要があります。これは、COUNT関数が集計の際にNULLを除外してからカウントを取るという特性を利用しています（0を指定するとCOUNT関数はその行も数えてしまうのです）。

このように、どれだけ表頭が複雑でも、入れ子が深くても、CASE式を集約関数の中に埋め込むことで簡単に記述できます。今の例題では人数を集計していましたが、もし給与の合計や平均を出したいならば、CASE式のTHEN句で返す値にsalary列を指定することで問題なく可能です。

ちなみに、表側が複雑な表を作る場合は、今回見たような表頭の複雑な表を作るよりもはるかに難しくなります。表側が入れ子であったり、合計・小計も合わせて出力するには、少し工夫をこらさねばなりません。これらの話については、また別の機会にすることにしましょう。

❹集約関数の外でCASE式を使う

問題❸では、集約関数の中にCASE式を記述することで、集約する対象を柔軟に使い分けるという技術を解説しました。今度は、ある意味でその反対をしてみましょう。すなわち、すでに集約された結果をCASE式の引数にとって分岐させます。

どういうことか、具体的に見ていきましょう。まず、問題❸で使った人事テーブルから、部署ごとの人数を選択する簡単なクエリから始めます（リスト10、図7）。

●リスト10　部署ごとの人数を選択する

```
SELECT dept,
       COUNT(*) AS cnt
  FROM Employees
 GROUP BY dept;
```

●図7　リスト10の実行結果

```
dept   cnt
----- -----
人事   2
製造   3
会計   2
営業   5
```

●リスト11　集約結果に対する条件設定：集約関数を引数にとる

```
SELECT dept,
       CASE WHEN COUNT(*) <= 2 THEN '2人以下'
            ELSE '3人以上' END AS cnt
  FROM Employees
 GROUP BY dept;
```

●図8　リスト11の実行結果

```
dept    cnt
----- -------
人事   2人以下
製造   3人以上
会計   2人以下
営業   3人以上
```

これだけならごく単純な話なのですが、業務要件によっては、「2人以下の部署と、それ以上の部署を別々のグループに分けたい」ということも生じるでしょう。いわば、集約した結果に対する、さらに一段上の分岐を記述したい、ということです。

凄いことに、CASE式はこういうケースにも適用できるのです。今度は引数にCOUNT関数を取ります（リスト11、図8）。

最初にこのクエリを見たとき、少し違和感を持つ人もいるでしょう。それは、WHERE句で同じように集約関数に条件を記述しようとしてエラーになる、というSQL初心者がよく犯してしまう間違いに原因があるのでしょう。しかし落ち着いて考えれば、SELECT句では集約関数はすでに1つの定数に定まっています。だからこそ、結果は1行につき1つの値で返されています。ということは、CASE式の引数としても、1つの定数として与えられるわけですから、構文的に問題は何1つないわけです。

もっとも、このような結果の見た目を整形する処理は、本当はSQLでやるべきことではありません。ホスト言語の表示用の機能において実現すればよいことです。というのも、SQLは本来、検索のために作られた言語であるため、表示用の整形機能はそれほど充実していないからです。この例題で見たような簡単な整形であればまだ問題はないのですが、きめ細かい表示設定を行う必要が出てきたときには、SQLでは対応しきれなくなるでしょう。そのことも、頭の隅には入れておいてください。

まとめ

手続き型言語においてIF文やCASE文といった条件分岐の機能が必須であるのと同様に、SQLにおいてもCASE式は生命線の1つです。したがって、DBエンジニアはこの機能の使い方について熟知していなければなりません。またそれだけに、CASE式を使いこなせるようになることによって、SQLプログラミングでできることの幅がぐっと広がり、データベースの世界が開けていきます。この爽快感を経験してもらうことが、実は本章の一番の目的です。

◇ ◇ ◇

それでは、主なポイントをまとめておきましょう。

- 手続き型言語で表現可能なアルゴリズムは、SQLにおいても表現可能である
- その際に基本的な重要性を持つ道具が、分岐とループ
- SQLにおける分岐は、CASE式を使って表現する。「文パラダイム」から「式パラダイム」への飛躍が理解の重要な鍵
- CASE式は、見た目が「文」に見えがちだが、実際は名前が示すとおり「式」であり、文法的には通常の列や定数、あるいは「1 + 1」のような式を記述しているのと変わらない
- それゆえ、SQLのほとんどどんなところにでも記述できる汎用性の高さが最大の魅力

SQLにおける分岐についてさらに深く学びたい方は、以下の参考文献を参照してください。

『プログラマのためのSQL　第2版』（J.セルコ 著、ピアソン・エデュケーション、2001）

セルコは、米国データベース界の重鎮の一人で、特に高度なSQLプログラミング技術の解説に手腕の冴えを見せます。

本書は、中級SQLプログラミングをマスターするために必要な知識がすべて網羅された決定版の教科書です。ただし、お世辞にもあまり読みやすくはありません。「7.1 CASE式」は、CASE式についての概念的な基礎から実際の応用例まで広くカバーする必読のテキストです。セルコもこのCASE式については「SQLにとって最重要の機能である」という破格の評価を与えています。

『達人に学ぶ　SQL徹底指南書』（ミック 著、翔泳社、2008）

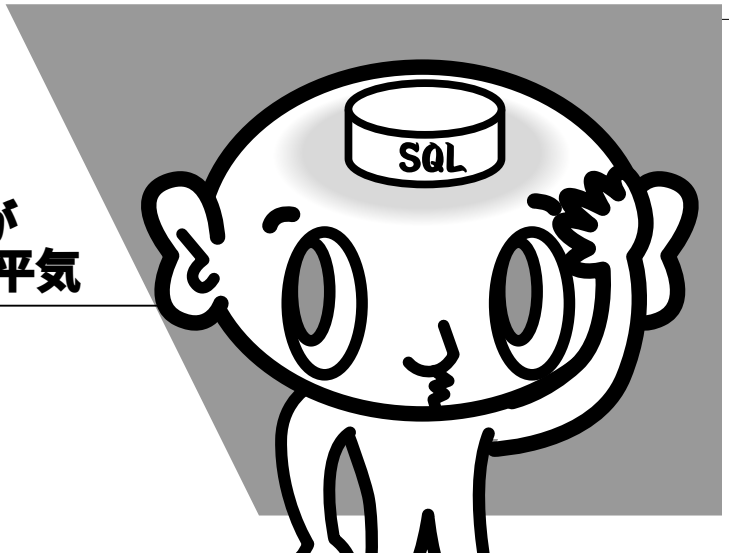
セルコの本は敷居が高くて…という方には、手前味噌ですがこちらの拙著をお薦めします。脱初級〜中級入門のレベルを噛み砕いて解説しています。

私は、SQLを教える際は、必ずCASE式の話を最初に配置するようにしています。本書も、その例に漏れず、筆頭に持ってきています（「1-1. CASE式のススメ」参照）。WHEN句の中でEXISTSなどの述語と組み合わせたり、入れ子のCASE式を使うなど、本章で紹介しきれなかった重要なテクニックも紹介しています。WD

第2章

OLAP関数と自己結合があればループがなくても平気

SQL流行間比較



はじめに

データベースを利用する目的の1つとして、過去の情報を時系列順に、文字通り「データの集積」として保持する用途があります。そうした情報は、たとえばシステムのパフォーマンスログや会社の財務状況、あるいは集団の人口推移だったりするでしょう。多くの業務では、こうした過去のデータへ遡って現在と比較する、あるいは過去の2点や期間同士を比較して将来の指針を策定する手がかりとすることが重要になります。いわゆる「データウェアハウス」と総称される用途です。

リレーショナルデータベースとSQLにおいては、そうした時系列的なデータの多くは、「時間」を表す列（時刻、日付、年度などなど）をキーとして、異なる時刻のデータは異なる行として保持するのが一般的なテーブル設計です。これを「列持ち」の形で保持することは、例外的なケースに属します[注1]。

そうすると当然、ある2点の時刻におけるデータを比較するためには行間比較が必要となります。そのために必要な記述方法の基礎を学ぶことが本章の目的です。手続き型言語でファイルを扱う場合には、こういう行間比較のためには「ループ」による処理が基本となりますが、SQLには文単位のループは一切現れません。代わりにSQLは、独自の原理に基づいた方法を用います。

具体的には、標準SQLの新しい機能であるOLAP関数を利用する方法と、従来の自己結合（相関サブクエリ）を使う方法の2通りを紹介します。この両者を学ぶことには、SQLに存在する手続き型的な側面と集合指向的な側面を一挙に学ぶことができるというメリットがあります。

まあ、そうしたことは、本章を読み進む過程で追々わかってくるでしょう。今は、頭の片隅にメモ程度に覚えておいていただければ結構です。それでは、早速始めましょう。

❶まずは基本直近を求める

まずは基本的な時系列分析から始めましょう。時系列にデータを比較する場合、基本となるのは、時系列に従って、1行ずつ過去へ遡る、または未来へ進むSQLです。サンプルに、表1のようなサーバの時間ごとの負荷量を記録したテーブルを使います。サンプリングは思いついたときに不定期に行われるため、不連続で間隔もランダムな日付が格納されています。

まずは、各行について過去の直近の行を求めてみましょう。OLAP関数を使える実装ならば、リスト1のような簡潔な書き方で実現できます（実行結果

●表1 LoadSampleテーブル

sample_date（計測日）	load（負荷量）
2008-02-01	1024
2008-02-02	2366
2008-02-05	2366
2008-02-07	985
2008-02-08	780
2008-02-12	1000

注1）もし列持ちの形式を取るとすれば、固定的な結果を出力するためのテーブルとして使用する場合でしょう。しかし、列数を拡張することはコストがかさむため、これはあくまで例外的なケースですし、また本章で見るように、行持ちから列持ちの形式へ変換することが可能なため、基本的には行持ちの形式をとるべきです。

●リスト1 過去の直近を求める(OLAP関数：現在のところ実装依存)

```
SELECT sample_date AS cur_date,
       MIN(sample_date)
          OVER (ORDER BY sample_date ASC
                ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS latest
  FROM LoadSample;
```

●リスト2 直近(相関サブクエリ：実装非依存)

```
SELECT LS0.sample_date AS cur_date,
         (SELECT MAX(sample_date)
            FROM LoadSample LS1
           WHERE LS1.sample_date < LS0.sample_date) AS latest
  FROM LoadSample LS0;
```

●図1 リスト1の実行結果

```
cur_date latest
-------- --------
08-02-01
08-02-02 08-02-01
08-02-05 08-02-02
08-02-07 08-02-05
08-02-08 08-02-07
08-02-12 08-02-08
```

は図1)[注2]。

2月1日より前のデータはこのテーブルには登録されていないので、2月1日の行については直前の日付はNULLです。2月2日以降についてはそれぞれ直前の日付が存在するので、これがlatest列に入ることになります。このクエリのポイントは、「BETWEEN 1 PRECEDING AND 1 PRECEDING」によって、OLAP関数が動く範囲をあくまでsample_dateでソートした場合の直前の1行に限定していることです。普通、「BETWEEN」というのは、複数行の範囲を指定するために使う場合が多いのですが、ここはあえて範囲を1行に限定するために利用しています。

これを実装非依存のクエリにするには、OLAP関数の部分を相関サブクエリに書き換えます(リスト2)。

これも結果はリスト1と同じです。ポイントはWHERE句の「LS1.sample_date < LS0.sample_date」という非等値結合です。この意味は、カレント行(LS0)のsample_dateより小さい日付の集合(LS1)から、最大の日付を選択する、ということです。このクエリの意味がわかりづらいなら、相関サブクエリを自己結合に変えたものを見るとわかりやすくなるでしょう(リスト3)。

この自己結合の場合も、考え方は相関サブクエリの場合とまったく同じです。試しに、集約抜きにしてヒラで結果を得てみましょう(リスト4、図2)。

このように、カレントの日付であるcur_dateに対して、それぞれより小さい日付の集合が得られることが確認できます。あとは、各集合の中からMAX関数で最大の日付(図2中の太字の箇所です)を選択すればOK。こういうある集合の中で、基準値より小さい最大の要素のことを、集合論では最大下界

●リスト3 直近(自己結合：実装非依存)

```
SELECT LS0.sample_date AS cur_date,
       MAX(LS1.sample_date) AS latest
  FROM LoadSample LS0
          LEFT OUTER JOIN LoadSample LS1
            ON LS1.sample_date < LS0.sample_date
 GROUP BY LS0.sample_date;
```

●リスト4 説明用：非集約の状態で取ると

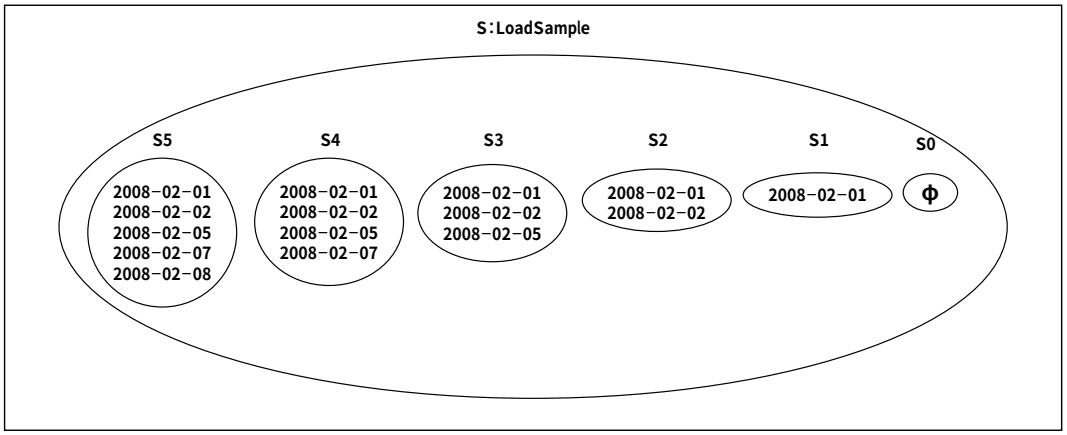
```
SELECT LS0.sample_date AS cur_date,
       LS1.sample_date AS latest
  FROM LoadSample LS0
          LEFT OUTER JOIN LoadSample LS1
            ON LS1.sample_date < LS0.sample_date;
```

●図2 リスト4の実行結果

```
cur_date latest
-------- --------
08-02-01            ← S0：2月1日より小さい日付は
                        1つもないので、0個
08-02-02 08-02-01   ← S1：2月2日より小さい日付は1個

08-02-05 08-02-01   ← S2：2月5日より小さい日付は2個
08-02-05 08-02-02

08-02-07 08-02-01   ← S3：2月7日より小さい日付は3個
08-02-07 08-02-02
08-02-07 08-02-05

08-02-08 08-02-01   ← S4：2月8日より小さい日付は4個
08-02-08 08-02-02
08-02-08 08-02-05
08-02-08 08-02-07

08-02-12 08-02-01   ← S5：2月12日より小さい日付は5個
08-02-12 08-02-02
08-02-12 08-02-05
08-02-12 08-02-07
08-02-12 08-02-08
```

注2) Oracleには、これをもっと簡略化したLAGという関数があります(未来へ進むならLEAD関数)。独自拡張の関数のためここでは紹介しませんが、Oracleユーザの方は使い方を試してみてください。

●図3 最大下界を得るためのノイマン型再帰集合



(greatest lower bound)と呼んでいます[注3]。過去の直近の日付を求めるとは、つまり最大下界を求めることと同義なのです。

S0～S5の部分集合は、図3からも明らかなように、

S5 ⊃ S4 ⊃ S3 ⊃ S2 ⊃ S1 ⊃ S0

という包含関係のある部分集合群です[注4]。こういう同心円的な集合は、数学的にはフォン・ノイマンによる再帰集合の構造にそのアイデアの源泉が求められます[注5]。

現在の日付の処理量と、直近の日付の処理量も併せて表示したいなら、リスト5のようにcur_load_amtとlatest_load_amtの2列を追加すればよいでしょう(実行結果は図4)。

ただし注意が必要なのは、latest_load_amtを求め

●リスト5 処理量も併せて表示する(執筆時点ではPostgreSQLでのみ動作)

```
SELECT LS0.sample_date AS cur_date,
       MAX(LS0.load_amt)   AS cur_load_amt,
       MAX(LS1.sample_date) AS latest,
       (SELECT MAX(load_amt)
          FROM LoadSample
         WHERE sample_date = MAX(LS1.sample_date)) AS latest_load_amt
  FROM LoadSample LS0
          LEFT OUTER JOIN LoadSample LS1
            ON LS1.sample_date < LS0.sample_date
 GROUP BY LS0.sample_date;
```

●図4 リスト5の実行結果(処理量も併せて表示)

```
cur_date   cur_load_amt  latest      latest_load_amt
---------- ------------  ----------  ---------------
2008-02-01         1024
2008-02-02         2366  2008-02-01             1024
2008-02-05         2366  2008-02-02             2366
2008-02-07          985  2008-02-05             2366
2008-02-08          780  2008-02-07              985
2008-02-12         1000  2008-02-08              780
```

注3) 反対概念はもちろん「最小上界」(least upper bound)です。非等値結合の不等号の向きを逆にして、MIN関数を使うことで取得できます。練習として試してみてください。

注4) 「⊃」は集合の包含関係を表す記号で、たとえば「哺乳類⊃人類」のように使います。
今、S0～S5は、大きな数字の集合が小さな数字の集合をすっぽり含む構造になっています(S0は空集合ですが、空集合は定義上すべての集合に含まれるとされるので、やはりS1～S5のすべてに含まれます)。

注5) この再帰集合の理論的背景については、『指南書』「1-2 自己結合の使い方」および「2-7 SQLと再帰集合」で詳細に取り上げたので、そちらも参照のほど。

るサブクエリ内のWHERE句でMAX関数を使っていることです。LS1.sample_dateは外側のGROUP BY句によって集約されているため、SELECT句では集約関数を適用する形でしか参照することができないのです。したがってこれは標準SQLに則った妥当なSQLなのですが、現在ではPostgreSQLでしか正しく動作しません[注6]。

そこで代替案として、リスト6のように最大下界を求めるロジックをWHERE句に移す方法が考えられます。

これも結果は先ほどと同じになります[注7]。

❷直近、直近の1つ前、そのまた1つ前…

これで1つ前の日付を求めることはできるようになりました。ですが実務では、もう少し比較の範囲を広げて、「直近の前」の日付や「直近の前のそのまた前」の日付、欲を言えば、一般的に「n個前の日付」と比較したい、という要望も生じることでしょう。つまり、行間比較の一般化です。

この要望に応えるため、まずはある日付を起点として、そこから順次過去へ遡った日付を求める方法を考えます。とりあえず3つ前まで求めるとすると、結果は図5のようにピラミッド型（というか階段型というか）になるでしょう。

まずはOLAP関数を使うならば、リスト7のようになります。

BETWEENによる行の指定先を「1行前」、「2行前」、「3行前」…と変更するだけなので、非常に簡単に済むのがよいところです。後は何行前でも同じやり方で拡張できますし、「行の順序」という手続き型の発想に基づく方法なので、多くのプログラマにとっても理解しやすいでしょう。

一方、OLAP関数を持たない実装でも動作する、より一般的な解法は、集合指向に基づいて考える必要があります。方法は、基本的に先ほどの直近の場合と同じで、時刻をずらした集合を複数用意して自己結合を使います。

答えはリスト8のようになります。

クエリの中に使いたいだけ集合を追加できるのが集合指向という考え方の柔軟なところです（その分、パフォーマンスを圧迫するのでやり過ぎには注意が必要ですが）。

S0が「今日」、S1が「その1つ前」、S2が「さらにそのもう1つ前」を表します。確かに2月5日の前の日付として、2月2日と2月1日が選択できています。外部結合を使っているのは、繰り返しになりますが、「前の日付」がテーブル上に存在しない日付についても結果から落とさずに一覧表示するた

●リスト6　最大下界を求めるロジックをWHERE句に記述

```
SELECT LS0.sample_date AS cur_date,
       LS0.load_amt    AS cur_load,
       LS1.sample_date AS latest,
       LS1.load_amt    AS latest_load
  FROM LoadSample LS0
         LEFT OUTER JOIN LoadSample LS1
           ON LS1.sample_date = (SELECT MAX(sample_date)
                                   FROM LoadSample
                                  WHERE sample_date < LS0.sample_date);
```

●リスト7　3つ前の日付まで出力する：OLAP関数の利用

```
SELECT sample_date AS cur_date,
       MIN(sample_date)
          OVER (ORDER BY sample_date ASC
                ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS latest_1,
       MIN(sample_date)
          OVER (ORDER BY sample_date ASC
                ROWS BETWEEN 2 PRECEDING AND 2 PRECEDING) AS latest_2,
       MIN(sample_date)
          OVER (ORDER BY sample_date ASC
                ROWS BETWEEN 3 PRECEDING AND 3 PRECEDING) AS latest_3
  FROM LoadSample;
```

●図5　想定される実行結果

```
cur_date latest_1 latest_2 latest_3
-------- -------- -------- --------
08-02-01
08-02-02 08-02-01
08-02-05 08-02-02 08-02-01
08-02-07 08-02-05 08-02-02 08-02-01
08-02-08 08-02-07 08-02-05 08-02-02
08-02-12 08-02-08 08-02-07 08-02-05
```

注6）この一見すると構文違反に見える（でも適法な）SQLの論理的な背景については、『指南書』の「2-10 SQLにおける存在の階層」を参照。

注7）Oracle 10gでは、外部結合の条件にサブクエリを指定できないという制限があるため、このクエリは動作しません。その場合は内部結合に変える必要がありますが、そうすると今度は現在の日付が「2008-02-01」の行が結果に現れないという不便が生じます。悩ましいものです。ほかの実装にはこういう不都合はありません。

めのトリックです。反対に、そういう行を結果に含める必要がないなら、単純な内部結合を使えばOKです。結果がどのように異なるか、自分で上のクエリを書き換えて確かめてみてください。

そうしたら、後は同じ要領で集合Snと結合条件を追加していけば、どれだけ過去にでも遡れます。無論、過去だけでなく未来へも自由に動けます。たとえば、2月5日の前後の直近を求めるなら、リスト9、リスト10のようなクエリで可能です（実行結果は図6）。

OLAP版、自己結合版ともに、「未来の直近」を求める関数をMAXからMINに変えているのがおわかりいただけるでしょう。たとえばカレントの日付が2月5日だとすると、集合論的な見方をすれば、これは集合を上下2つに分割する境界線（bound）を与える役割を果たしているのです（図7）。

●リスト8　3つ前の日付まで出力する：自己結合の利用

```
SELECT LS0.sample_date AS sample_date,
       MAX(LS1.sample_date) AS latest_1,
       MAX(LS2.sample_date) AS latest_2,
       MAX(LS3.sample_date) AS latest_3
  FROM LoadSample LS0
           LEFT OUTER JOIN LoadSample LS1
             ON LS1.sample_date < LS0.sample_date
             LEFT OUTER JOIN LoadSample LS2
               ON LS2.sample_date < LS1.sample_date
               LEFT OUTER JOIN LoadSample LS3
                 ON LS3.sample_date < LS2.sample_date
 GROUP BY LS0.sample_date;
```

●リスト9　現在と過去の直近を求める：OLAP版

```
SELECT MIN(sample_date)
          OVER (ORDER BY sample_date ASC
                ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS past,
       sample_date AS cur_date,
       MAX(sample_date)
          OVER (ORDER BY sample_date DESC
                ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS future
  FROM LoadSample;
```

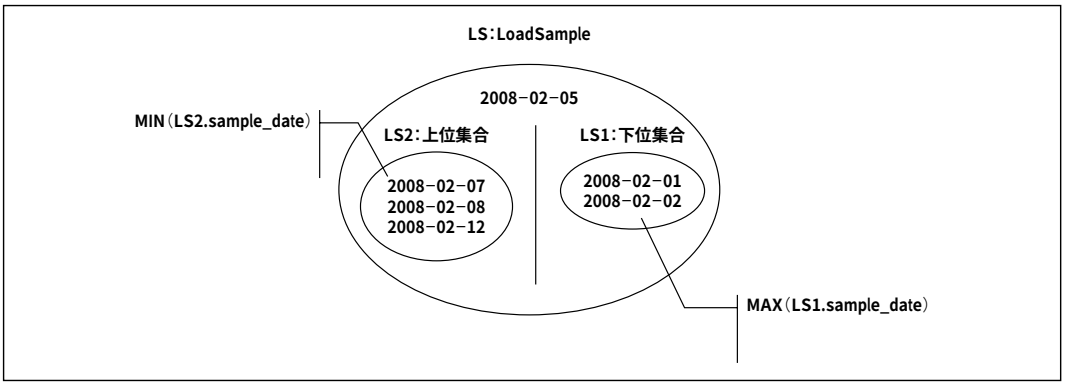
●リスト10　現在と過去の直近を求める：自己結合版

```
SELECT MAX(LS1.sample_date) AS past,
       LS.sample_date AS sample_date,
       MIN(LS2.sample_date) AS future
  FROM LoadSample LS
           LEFT OUTER JOIN LoadSample LS1
             ON LS1.sample_date < LS.sample_date
             LEFT OUTER JOIN LoadSample LS2
               ON LS2.sample_date > LS.sample_date
 GROUP BY LS.sample_date;
```

●図6　リスト9、10の実行結果

```
past     cur_date future
-------- -------- --------
         08-02-01 08-02-02
08-02-01 08-02-02 08-02-05
08-02-02 08-02-05 08-02-07
08-02-05 08-02-07 08-02-08
08-02-07 08-02-08 08-02-12
08-02-08 08-02-12
```

●図7　リスト9、10で行っていること



2月5日の前後の日付を求めるということは、上位集合の最小元（＝最小上界）と下位集合の最大元（最大下界）を求めることと同義です。もちろん、範囲は上下にいくらでも広げられます。こう考えてみると、順序の概念を使うOLAP関数の手続き型発想と、集合の概念を使う集合指向の考え方も、使う道具立てが異なるだけで、ロジックの構造として帰するところは1つであることがわかるのではないでしょうか。

このように、与えられた集合（テーブル）をカットして新たな部分集合を作り、それを操作することによってさまざまな演算を自在にこなせるようになると、SQLの使い方が一段レベルアップします。この「集合のカット」という技術については、次章でさらに別の観点から詳しく見ることになります。

さて、それでは最後に、応用問題へ進みましょう。

❸小分けにしたグループ内での行間比較

これまで使っていたLoadSampleテーブルは、ある特定のマシンに限定した内容になっていました。今度はこれにマシンを管理する列を追加して、一般化したテーブルをサンプルに使いましょう（表2）。

●表2 LoadSample2テーブル

machine（マシン名）	sample_date（計測日）	load（負荷量）
PC1	2008-02-01	1024
PC1	2008-02-02	2366
PC1	2008-02-05	2366
PC1	2008-02-07	985
PC1	2008-02-08	780
PC1	2008-02-12	1000
PC2	2008-02-01	999
PC2	2008-02-02	50
PC2	2008-02-05	328
PC2	2008-02-07	913
PC3	2008-02-01	2000
PC3	2008-02-02	1000

今回は、日付の列だけを見た場合には、テーブル内で一意になっていません。こういうケースでは、日付を時系列に並べる際の基準としても、

1．あくまでテーブル全体で日付を並べる
2．マシンごとに集合を区切り、その中で日付を並べる

という2通りが考えられます。これに対応するよう、❷「直近、直近の1つ前、そのまた1つ前…」のクエリを拡張するのが今回の目的です。まず1番目の方針に従って過去の日付を順に求めるなら、結果は❷の場合と同じく図8のようになるでしょう。

❷のクエリをそのまま適用するだけでは、上の結果は得られません。論より証拠、実験してみましょう（リスト11、図9）。

そう、結果はそのままテーブルの12行が現れてしまいます。OLAP関数そのものに集約機能はないのだから、これは当然と言えば当然の結果です。ということは、裏を返せば、正しい結果を得るには、事前に日付で一意になるような中間テーブルを作っておけばよい、ということになります。したがって、正しいクエリはリスト12のようになります。

ポイントは最後のGROUP BY句にあります。SQLはGROUP BY句で指定したキーによって一意になる集合を作ります。ここでは、sample_dateを指定しているので、これによって元のテーブルでは重複値の存在したsample_date列を一意にすることができるわけです。

なお、SELECT句のOLAP関数には、❷のクエリから手を加える必要はありません。これは、SELECT句がGROUP BY句より後に実行されるからです。GROUP BY句で❷と同じ中間テーブルを

●リスト11 ❷のクエリをそのまま実行してみる：うまくいかない

```
SELECT sample_date AS cur_date,
       MIN(sample_date)
         OVER (ORDER BY sample_date ASC
               ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS latest_1,
       MIN(sample_date)
         OVER (ORDER BY sample_date ASC
               ROWS BETWEEN 2 PRECEDING AND 2 PRECEDING) AS latest_2,
       MIN(sample_date)
         OVER (ORDER BY sample_date ASC
               ROWS BETWEEN 3 PRECEDING AND 3 PRECEDING) AS latest_3
  FROM LoadSample2;
```

●図8 1．の実行結果

```
cur_date   latest_1 latest_2 latest_3
---------- -------- -------- --------
08-02-01
08-02-02   08-02-01
08-02-05   08-02-02 08-02-01
08-02-07   08-02-05 08-02-02 08-02-01
08-02-08   08-02-07 08-02-05 08-02-02
08-02-12   08-02-08 08-02-07 08-02-05
```

●リスト12 OLAP版(machine列なし)

```
SELECT sample_date,
       MIN(sample_date)
          OVER (ORDER BY sample_date ASC
                ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS latest_1,
       MIN(sample_date)
          OVER (ORDER BY sample_date ASC
                ROWS BETWEEN 2 PRECEDING AND 2 PRECEDING) AS latest_2,
       MIN(sample_date)
          OVER (ORDER BY sample_date ASC
                ROWS BETWEEN 3 PRECEDING AND 3 PRECEDING) AS latest_3
  FROM LoadSample2
 GROUP BY sample_date;
```

●図9 リスト11の実行結果

```
cur_date latest_1 latest_2 latest_3
-------- -------- -------- --------
08-02-01
08-02-01 08-02-01
08-02-01 08-02-01 08-02-01
08-02-02 08-02-01 08-02-01 08-02-01
08-02-02 08-02-02 08-02-01 08-02-01
08-02-02 08-02-02 08-02-02 08-02-01
08-02-05 08-02-02 08-02-02 08-02-02
08-02-05 08-02-05 08-02-02 08-02-02
08-02-07 08-02-05 08-02-05 08-02-02
08-02-07 08-02-07 08-02-05 08-02-05
08-02-08 08-02-07 08-02-07 08-02-05
08-02-12 08-02-08 08-02-07 08-02-07
```

用意できれば、後の演算には影響が及ばない、という仕掛けです。

実装非依存の自己結合バージョンも同様の考え方でいけます。やはりGROUP BY句を追加します(リスト13)。

さて、それでは次に、2番目の方法、つまりマシン列ごとに集合を区切って計測日を並べる場合を考えましょう。今度は、求める結果は図10のようになります。

もともと2月12日に計測を行っていないPC2やPC3には、2月12日が現れないことに注目してください。ほかにも、PC3には2月5日や2月8日も結果に現れません。これはもとのテーブルの情報からも妥当なことです。

こうした元のテーブルを重複しない部分集合(「類」と呼びます)に分割する方法を、SQLはちゃんと持っています。それが、次章で詳しく取り上げることになるGROUP BYとPARTITION BY句です。答えはリスト14、リスト15のようになります。

OLAP版も自己結合版も、GROUP BYと

●リスト13 自己結合版(machine列なし)

```
SELECT LS0.sample_date AS cur_date,
       MAX(LS1.sample_date) AS latest_1,
       MAX(LS2.sample_date) AS latest_2,
       MAX(LS3.sample_date) AS latest_3
  FROM LoadSample2 LS0
          LEFT OUTER JOIN LoadSample2 LS1
            ON LS1.sample_date < LS0.sample_date
            LEFT OUTER JOIN LoadSample2 LS2
              ON LS2.sample_date < LS1.sample_date
              LEFT OUTER JOIN LoadSample2 LS3
                ON LS3.sample_date < LS2.sample_date
 GROUP BY LS0.sample_date;
```

●図10 2.の実行結果

```
machine cur_date latest_1 latest_2 latest_3
------- -------- -------- -------- --------
PC1     08-02-01
PC1     08-02-02 08-02-01
PC1     08-02-05 08-02-02 08-02-01
PC1     08-02-07 08-02-05 08-02-02 08-02-01
PC1     08-02-08 08-02-07 08-02-05 08-02-02
PC1     08-02-12 08-02-08 08-02-07 08-02-05
PC2     08-02-01
PC2     08-02-02 08-02-01
PC2     08-02-05 08-02-02 08-02-01
PC2     08-02-07 08-02-05 08-02-02 08-02-01
PC3     08-02-01
PC3     08-02-02 08-02-01
```

●リスト14 OLAP版(machine列あり)

```
SELECT machine,
       sample_date AS cur_date,
       MIN(sample_date)
          OVER (PARTITION BY machine ORDER BY sample_date ASC
                ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS latest_1,
       MIN(sample_date)
          OVER (PARTITION BY machine ORDER BY sample_date ASC
                ROWS BETWEEN 2 PRECEDING AND 2 PRECEDING) AS latest_2,
       MIN(sample_date)
          OVER (PARTITION BY machine ORDER BY sample_date ASC
                ROWS BETWEEN 3 PRECEDING AND 3 PRECEDING) AS latest_3
  FROM LoadSample2
 GROUP BY machine, sample_date;
```

●リスト15 自己結合版（machine列あり）

```
SELECT LS0.machine AS machine,
       LS0.sample_date AS sample_date,
       MAX(LS1.sample_date) AS latest_1,
       MAX(LS2.sample_date) AS latest_2,
       MAX(LS3.sample_date) AS latest_3
  FROM LoadSample2 LS0
           LEFT OUTER JOIN LoadSample2 LS1
             ON LS1.sample_date < LS0.sample_date
             LEFT OUTER JOIN LoadSample2 LS2
               ON LS2.sample_date < LS1.sample_date
               LEFT OUTER JOIN LoadSample2 LS3
                 ON LS3.sample_date < LS2.sample_date
 GROUP BY LS0.machine, LS0.sample_date;
```

PARTITION BYのキーとしてmachine列を使っています。これによって、日付探索の範囲を同一マシン内に制限できるわけです。

OLAP版のクエリでは、PARTITION BYとGROUP BYの両方にmachine列を追加する必要があることに注意してください。PARTITION BY句に追加し忘れると、動作はしますが結果が正しくなりません。GROUP BY句に追加し忘れると、構文エラーとなります。それぞれ、なぜ正しく動作しないか、理由を考えてみてください。これがわかれば本章の理解は十分です。

まとめ

本章ではSQLを利用した行間比較の方法を、いろいろなパターンに応じて解説してきました。

ここまで読んでいただいた方には、章の冒頭で述べた「行間比較を学ぶことで、SQLに存在する手続き型的な側面と集合指向的な側面を一挙に学ぶことができる」という言葉の意味も理解してもらえたのではないでしょうか。手続き型の考え方に基礎を置くOLAP関数と、伝統的な集合指向による自己結合。熟練したDBエンジニアには、この両者をともに使いこなし、状況に応じて使い分けることが要請されます。きっちり理解しておいてください。

それでは、本章の要点です。

- 一般的なテーブル設計では、時系列的なデータは普通、行持ちの形式を取る
  - したがって、時間をまたいだ比較を行うことは、行をまたいだ比較（＝行間比較）を必要とする
- SQLで行間比較を行う方法には次の2つがある
  - －1つは手続き型の「順序」の概念を前面に押し出したOLAP関数。簡潔に記述できるが、まだ実装に依存する
  - －もう1つは、伝統的な集合指向に基づく自己結合。実装非依存だが、パフォーマンスに注意が必要

行間比較についてさらに学びたい方は、以下の参考文献を参照してください。

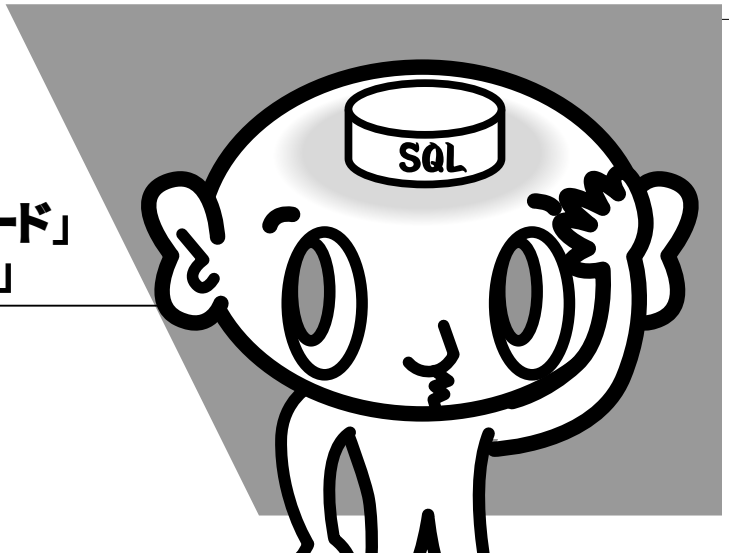
『SQLパズル 第2版』（J.セルコ著、翔泳社、2007）

「過去の直近の日付」という形で最大下界を求めるのが「パズル30 買い物の平均サイクル」、DELETE文とUPDATE文で行間比較を利用する問題が「パズル2 欠勤」と「パズル38 記録の更新」。そして相関サブクエリで行同士を比較する高度な応用問題が「パズル70 株価の動向」。いずれもSQLにおける行間比較をマスターするのに格好の練習問題です。

『達人に学ぶ SQL徹底指南書』（ミック、翔泳社、2008）

「1-6 相関サブクエリで行と行を比較する」では、主に相関サブクエリと自己結合を使った行間比較を取り上げました。移動平均や移動累計のような「集合をずらして重ね合う」考え方などは、本章の発展版としておもしろいでしょう。Web

第3章



思考の基本単位は「レコード」ではなく「レコードの集合」

SQL流集合操作

はじめに

SQLという言語の大きな特徴として、処理をレコード単位ではなく、レコードの「集合」単位でひとまとめにして記述するというものが挙げられます。具体的には、GROUP BY句とHAVING句、それに伴って利用されるSUMやCOUNTなどの集約関数の使い方が鍵になります。SQLでは、これら集合操作の機能が充実しているため、手続き型言語ならば複雑なループや分岐を使って記述せねばならない処理を、非常に簡単で見通しよくコーディングすることが可能になっています。

しかし一方で、プログラミングにおける思考の基本単位を「レコード」から「レコードの集合」に切り替えるためには、多少の発想の転換を要します。この切り替えがうまくいかないために、せっかくSQLが最も本領を発揮するフィールドであるにもかかわらず、その機能を十全に利用できないまま、もどかしい思いを抱えているエンジニアも少なくないでしょう。

本章では、このSQLの一番「SQLらしい」機能の活かし方を、これまで同様、いくつかの小さなケーススタディを通じて見ていきたいと思います。

❶複数行を1行にまとめる

SQLには、集約関数（aggregate function）という名前で、ほかのいろいろな関数とは区別されて呼ばれる関数が存在します。具体的には以下の5つです。

- COUNT
- SUM
- AVG
- MAX
- MIN

たぶん、みなさんにとってもお馴染みの関数ばかりでしょう。これ以外にも拡張的な集約関数を用意している実装もありますが[注1]、標準SQLで用意されているのはこの5つです。なぜこれらに「集約」という接頭辞がついているかというと、文字通り、複数行を1行にまとめる効果を持つからです。

この効果を体感するために、1つ例題を解いてみましょう。いま、表1のようなサンプルテーブルがあるとします。

CSVや固定長などのフラットファイルをそのままテーブルに写し取った形の、よく見かける擬似配列テーブルです。人物を管理するid列に、データの種別を管理するrec_typeを加えて、主キーとしています。data_1～data_6の列は、人物一人ひとりについての何らかの情報を表していると考えてください。

さて、テーブルの色分けに注目しましょう。data_type列がAの行については、data_1とdata_2、Bの行についてはdata_3～data_5、Cの行についてはdata_6について背景の濃度を変えています。これは、この業務において実際に使用したいデータのフィールドを示すものです。

すると、この非集約テーブルのように1人の人間についての情報が複数行に分散していると、その情報にアクセスするためのクエリも複数回発行する必

注1）たとえばOracleなら、分散や最頻値を求める集約関数も持っています。用途が統計に偏っているのは、統計もまた集団の個々の要素ではなく、集団そのものを扱う基本単位とする分野であることを考えれば当然の話です。リレーショナルデータベースと統計は、非常に相性がよいのです。

特集2 分岐とループ、集合操作… プログラミング言語とはここが違う！ SQLアタマ養成講座

●表1 NonAggTbl：非集約テーブル

id	data_type	data_1	data_2	data_3	data_4	data_5	data_6
Jim	A	100	10	34	346	54	
Jim	B	45	2	167	77	90	157
Jim	C		3	687	1355	324	457
Ken	A	78	5	724	457		1
Ken	B	123	12	178	346	85	235
Ken	C	45		23	46	687	33
Beth	A	75	0	190	25	356	
Beth	B	435	0	183		4	325
Beth	C	96	128		0	0	12

要が生じます。たとえば、Jimについての情報を得ようとする場合、リスト1〜3のように3つの異なるクエリが必要になります（実行結果は図1〜3）。

これは、3回発行するためにパフォーマンス上のコストがかかるだけでなく、結果の形式を1行にまとめることもできないうえ、クエリごとに列数が異なるためUNIONで1つのクエリにまとめることも困難という、はなはだ非効率的な方法と言わねばなりません。

●リスト1 データタイプ「A」の行に対するクエリ

```
SELECT id, data_1, data_2
  FROM NonAggTbl
 WHERE id = 'Jim'
   AND data_type = 'A';
```

●図1 リスト1の実行結果

```
id    data_1    data_2
----  -------   -------
Jim   100       10
```

●リスト2 データタイプ「B」の行に対するクエリ

```
SELECT id, data_3, data_4, data_5
  FROM NonAggTbl
 WHERE id = 'Jim'
   AND data_type = 'B';
```

●図2 リスト2の実行結果

```
id    data_3    data_4    data_5
----  ------    ------    ------
Jim   167       77        90
```

●表2 AggTbl：1人1行に集約したテーブル

id	data_1	data_2	data_3	data_4	data_5	data_6
Jim	100	10	167	77	90	457
Ken	78	5	178	346	85	33
Beth	75	0	183		4	12

したがって本当は、表2のような形のテーブルが一番使いやすい理想的なテーブルということになります。

先ほどのNonAggTblと比べれば、その違いは明らかです。非集約テーブルでは、1人についての情報が複数行に分散していたため1人の情報を参照するためにも複数の行にアクセスする必要があったのですが、集約後のテーブルを見れば、1人の人間についての情報がすべて同じ行にまとめられているので1つのクエリで済みます。

では問題です。NonAggTblからAggTblへの変換を行うSQLを考えましょう。考え方としては、まず人物単位に集約するので、GROUP BY句に書く集約キーは、id列ということになります。後は、選択する列をデータタイプによって分岐させればよいわけです。ここで、第1章で登場したCASE式が有効です。すると、まずリスト4のような形のクエリができます。

このクエリは、残念ながら構文違反のためエラーとなります[注2]。というのも、GROUP BYを使って集約操作を行った場合、SELECT句に書くことのでき

●リスト3 データタイプ「C」の行に対するクエリ

```
SELECT id, data_6
  FROM NonAggTbl
 WHERE id = 'Jim'
   AND data_type = 'C';
```

●図3 リスト3の実行結果

```
id    data_6
----  ------
Jim   457
```

注2）MySQLは、このクエリを通すような独自拡張を施していますが、標準違反でほかの実装との互換性もないため、使わないほうがよいでしょう。なぜMySQLがこのような独自拡張をしているかというと、本文にも書いたとおり、単元集合と要素を混同しているからです。詳しくは、『指南書』の「2-10 SQLにおける存在の階層」を参照。

るのは、

1．定数
2．GROUP BY句で指定した集約キー
3．集約関数

に限定されるからです。いま、CASE式の中で使われているdata_1～data_6は、このどれにも該当しません。確かに、id列でグループ化したうえ、さらにCASE式でデータタイプまで指定したなら、それによって行は一意に定まります。したがって別に集約関数を使わなくても、data_1～data_6を「裸で」書いたとしても、データベースエンジンが気を利かせてくれれば値は一意に定まります（実際、MySQLではこのクエリはエラーにはなりません）。

しかしこれは、単元集合と要素を混同する、SQLの原理（それはつまり集合論の原理でもある）に反する行いです。したがって正しくは、面倒でも集約関数を使ってリスト5のように書く必要があります（実行結果は図4）。

MAX関数を使ったのは、どうせ切り分けた集合が1行しか含まないのだから、最大値を取ろうと値が変わらないからです[注3]。あとは、この結果を別に用意したAggTblテーブルにINSERTすればOKです。または、パフォーマンスに不安がなければ、このクエリをそのままビューに保存してもよいでしょう。

いかがでしょう。まさに「複数行を1行に集約する」というGROUP BY句の特徴（そしてもちろん、CASE式の便利さも）がよく理解できる1問ではないでしょうか。

❷合わせ技1本

引き続き、集約操作の練習をもう1問やっておきましょう。問題は、『SQLパズル　第2版』の「パズル65　製品の対象年齢の範囲」を使います。表3のような、複数の製品の対象年齢ごとの値段を管理す

●リスト4　惜しいけど間違い

```
SELECT id,
       CASE WHEN data_type = 'A' THEN data_1 ELSE NULL END AS data_1,
       CASE WHEN data_type = 'A' THEN data_2 ELSE NULL END AS data_2,
       CASE WHEN data_type = 'B' THEN data_3 ELSE NULL END AS data_3,
       CASE WHEN data_type = 'B' THEN data_4 ELSE NULL END AS data_4,
       CASE WHEN data_type = 'B' THEN data_5 ELSE NULL END AS data_5,
       CASE WHEN data_type = 'C' THEN data_6 ELSE NULL END AS data_6
  FROM NonAggTbl
 GROUP BY id;
```

●リスト5　これが正解。どの実装でも通る

```
SELECT id,
       MAX(CASE WHEN data_type = 'A' THEN data_1 ELSE NULL END) AS data_1,
       MAX(CASE WHEN data_type = 'A' THEN data_2 ELSE NULL END) AS data_2,
       MAX(CASE WHEN data_type = 'B' THEN data_3 ELSE NULL END) AS data_3,
       MAX(CASE WHEN data_type = 'B' THEN data_4 ELSE NULL END) AS data_4,
       MAX(CASE WHEN data_type = 'B' THEN data_5 ELSE NULL END) AS data_5,
       MAX(CASE WHEN data_type = 'C' THEN data_6 ELSE NULL END) AS data_6
  FROM NonAggTbl
 GROUP BY id;
```

●図4　リスト5の実行結果

```
id      data_1  data_2  data_3  data_4  data_5  data_6
------  ------  ------  ------  ------  ------  ------
Jim     100     10      167     77      90      457
Ken     78      5       178     346     85      33
Beth    75      0       183     NULL    4       12
```

注3）だから別に、MINやAVG、SUMを使ってもこの場合は同じです。ただし、今回はdata_1～data_6の型が数値型なのでAVGやSUMでもかまいませんが、文字型や日付型など対応するために、MAXかMINのどちらかを使う習慣をつけておくのがよいと思います。

るテーブルがあるとします。同じ製品IDでも値段の異なる製品があるのは、対象年齢によって設定や難易度を変えたバージョンの違いによるもの、くらいに考えてください。また1つの製品について、年齢範囲の重複するレコードはないものと仮定します。

すると、このテーブルにおいては、(製品ID, 対象年齢の下限）で、レコードが一意に定まります（下限の代わりに上限を使ってもかまいません)。考えてもらう問題は、これらの製品の中から、0～100歳までのすべての年齢で遊べる製品を求めるというものです。もちろん、バージョンの相違は無視して、製品ID単位で考えます。

図5のように図示してみると、問題の意図がよりわかりやすくなるでしょう。

製品1の場合、2レコードを使って0～100までの整数の全範囲をカバーできています。したがって、製品1は今回の条件を確かに満たします。一方、製品3の数直線を見ると、3レコードも使っているにもかかわらず、21～30の間が断絶していることが見て取れます。こちらは残念ながらNGです。

このように、たとえ1レコードで全年齢範囲をカバーできなかったとしても、複数のレコードを組み合わせてカバーできたなら、「合わせ技1本」とみなす、というのがこの問題の主旨です。

そうとわかれば、あとの話は先ほどの問題と同じです。まず、集約する単位は製品ですから、集約キーは製品IDに決まります。あとは、各レコードの範囲の大きさをすべて足しこんだ合計が101に到達している製品を探し出せば任務完了です（0から100までなので、値の個数は101個であることに注意)。

答えはリスト6のようになります。

HAVING句の「high_age - low_age + 1」で、各行の年齢範囲が含む値の個数が算出されます。あとは、それを同じ製品内で足し合わせればよいわけです。

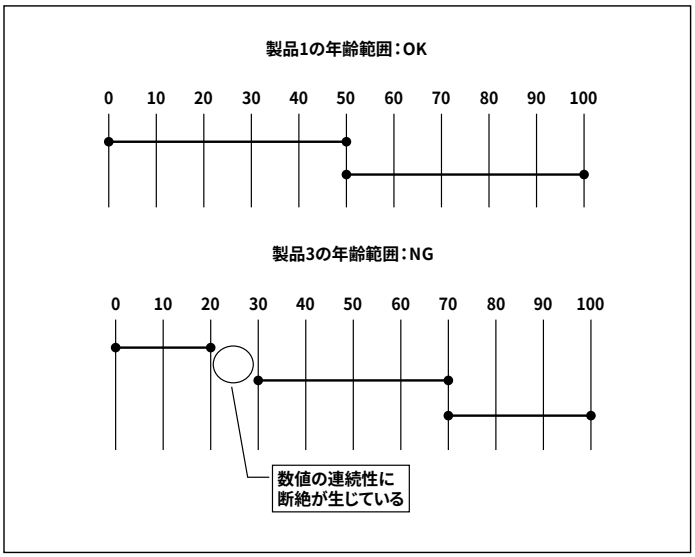
今は、サンプルとして「年齢」という数値型のデータを用いましたが、より一般的に日付や時刻に拡張することもできます。たとえば、応用問題としてこんなのはどうでしょう。ホテルの部屋ごとに、投宿日と出発日の履歴を記録するテーブルを使います（表4)。

このテーブルから、稼働日数が10日を超える部屋を選択してください。稼働日数の定義は、宿泊日数で計ることとします。だから、投宿日が2月1日、出発日が2月6日の場合は、5泊なので5日です。これは演習問題として、宿題にしておきましょう（解答は後日、筆者のWebサイト[注4]に掲載予定です)。

●表3 PriceByAge

product_id (製品ID)	low_age (対象年齢の下限)	high_age (対象年齢の上限)	price (値段)
製品1	0	50	2000
製品1	51	100	3000
製品2	0	100	4200
製品3	0	20	500
製品3	31	70	800
製品3	71	100	1000
製品4	0	99	8900

●図5 製品の対象年齢の範囲



●リスト6 正解

```
SELECT product_id
  FROM PriceByAge
 GROUP BY product_id
HAVING SUM(high_age - low_age + 1) = 101;
```

注4)「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

❸あなたは肥り過ぎ？ 痩せ過ぎ？ ～カットとパーティション～

これまでの2問では、GROUP BYの「集約」という側面を強調してその機能を調べてきました。ですが、冒頭でも少し触れたようにGROUP BYには、集約以外にも、もう1つ重要な機能があります。それが、「カット」という機能です。これは要するに、母集合である元のテーブルを小さな部分集合に切り分けることです。だからGROUP BYというのは、実はこれ1つの中に、

1．カット
2．集約

という2つの操作が組み込まれた演算なのです[注5]。1つの句の中に2つの演算が組み込まれているというのもGROUP BYに対する理解を阻む一因になっているのですが、まあそれはいま言っても始まりません。今度は、この「カット」の機能に焦点を当ててみましょう。サンプルに、表5のような個人の身長などの情報を保持するテーブルを使います。

あなたは、上司からこのテーブルを使って簡単な集計作業を依頼されたとします。まずは小手調べ。名簿のインデックスを作るために、名前の頭文字のアルファベットごとに何人がテーブルに存在するかを集計しましょう。

これはつまり、Persons集合を図6のようなS1～S4の部分集合に切り分けて、それぞれの要素数を調べる、ということです。

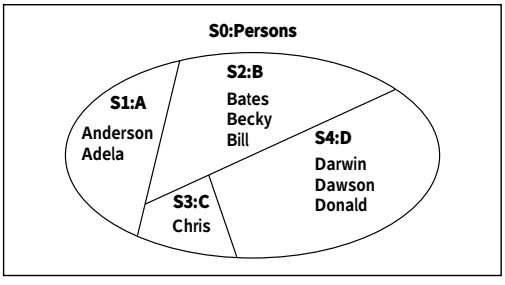
集合の要素数を調べる関数は、もちろんCOUNT。あとは、頭文字をGROUP BYのキーに指定すれば、カット完了です。SQLはリスト7です。

こういうGROUP BY句でカットして作られた1つひとつの部分集合は、数学的には「類（partition）」と呼ばれます。同じ母集合からでも、類の作り方は切り分け方によってさまざまあります。たとえば、

●表4 HotelRooms

room_nbr (部屋番号)	start_date (投宿日)	end_date (出発日)
101	2008-02-01	2008-02-06
101	2008-02-06	2008-02-08
101	2008-02-10	2008-02-12
202	2008-02-05	2008-02-06
202	2008-02-08	2008-02-09
202	2008-02-09	2008-02-10
303	2008-02-03	2008-02-17

●図6 4つの部分集合に切り分けてそれぞれの要素数を調べる



●表5 Persons

name (名前)	age (年齢)	height (身長 cm)	weight (体重 kg)
Anderson	30	188	90
Adela	21	167	55
Bates	87	158	48
Becky	54	187	70
Bill	39	177	120
Chris	90	175	48
Darwin	12	160	55
Dawson	25	182	90
Donald	30	176	53

●リスト7 頭文字のアルファベットごとに何人がテーブルに存在するか集計するSQL

```
SELECT SUBSTRING(name, 1, 1) AS label,
         COUNT(*)
  FROM Persons
 GROUP BY SUBSTRING(name, 1, 1);
```

●図7 リスト7の実行結果

```
label       COUNT(*)
------      ---------
A           2
B           3
C           1
D           3
```

注5）GROUP BYから集約の機能を取り去って、カットの機能だけを残したのがPARTITION BYです。この2つの句を比較すると、興味深いことが見えてきます。『指南書』の「2-5 GROUP BYとPARTITION BY」を参照。

年齢によって、子供（20歳未満）、成人（20～69歳）、老人（70歳以上）に分けるなら、図8のようにカットされます。

当然、GROUP BYのキーもこの3つの区分に対応する形になります。これは、CASE式を使ってリスト8のように表現します（実行結果は図9）。

カットしたい区分を、GROUP BY句とSELECT句の両方に書いてやるのがポイントです。PostgreSQLとMySQLでは、SELECT句で付けた「age_class」という別名を使って、「GROUP BY age_class」という簡潔な書き方も許しているのですが、標準違反なので勧めません。

さて、それでは最後の問題。もっと複雑な基準でPersons集合をカットしてみましょう。これが解けたら、GROUP BY句に対するみなさんの理解は十分であることを保証しましょう。

BMIによるカット

健康診断などで、BMIという体重の指標を見たことがあると思います。身長をt（メートル）、体重をw（キログラム）とすると、以下の式で求められます。

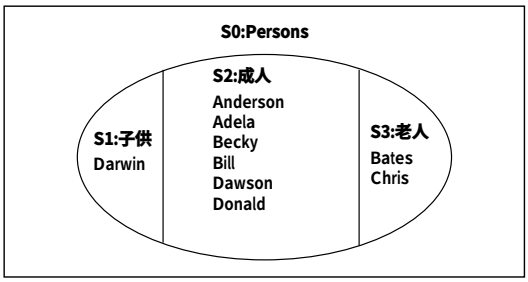
$$BMI = w / t^2$$

ここで、身長はセンチではなくメートルであることに注意してください。これによって求められた数値に基づいて、日本では18.5未満を「やせている」、18.5以上25未満を標準、25以上を肥満としています。この基準に基づいて、Personsテーブルの人々の体重を分類して、各階級の人数を求めます。ちなみに、各人のBMIは表6のとおり。

すると、カットのイメージは図10のようになります。

まずBMIを計算しましょう。これは「weight / POWER(height / 100, 2)」という式で簡単に求められます。こうして求められたBMIをCASE式で3つの階級に振り分ければ、カットする基準が作れます。あとは、これをGROUP BY句とSELECT句に書けばできあがり（リスト9。実行結果は図11）。

●図8 年齢によるカット



●リスト8 年齢による区分を実施

```
SELECT CASE WHEN age < 20 THEN '子供'
            WHEN age BETWEEN 21 AND 69 THEN '成人'
            WHEN age > 70 THEN '老人'
            ELSE NULL END AS age_class,
       COUNT(*)
  FROM Persons
 GROUP BY CASE WHEN age < 20 THEN '子供'
               WHEN age BETWEEN 21 AND 69 THEN '成人'
               WHEN age > 70 THEN '老人'
               ELSE NULL END;
```

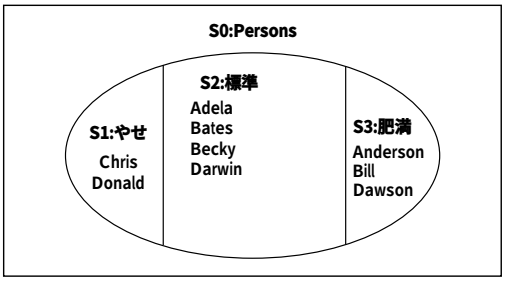
●図9 リスト8の実行結果

```
age_class       COUNT(*)
----------      --------
子供             1
成人             6
老人             2
```

●表6 BMI

名前	BMI	分類
Anderson	25.5	肥満
Adela	19.7	標準
Bates	19.2	標準
Becky	20.0	標準
Bill	38.3	肥満
Chris	15.7	やせ
Darwin	21.5	標準
Dawson	27.2	肥満
Donald	17.1	やせ

●図10 BMIによるカットのイメージ



GROUP BY句が「SQLの本領」である、という言葉の意味が、少しわかっていただけたのではないでしょうか。GROUP BY句には列名を書くものだと思い込んでいる人にとっては、こんな複雑な基準によるパーティションカットが可能であると知ることは、一種の感動をもたらします（かつて私もそうでした）。

年齢階級別のパーティションカット

それでは最後に、発展的な話をして章を締めくくりましょう。先ほど私は、注5で「GROUP BYから集約機能を取り去って、カットの機能だけ残したのがPARTITION BYだ」と述べました。実際、その1点を除けば、GROUP BYとPARTITION BYに機能的な差はありません。

ということは、です。PARTITION BY句にも、やはり単純な列名だけでなく、CASE式や計算式を利用した複雑な基準を記述できてもおかしくないはずです。そして事実、それは可能なのです。

たとえば、さっきの年齢階級別のパーティションカットを使いましょう。これをPARTITION BY句に記述して、同一年齢階級内で年齢の上下によって順位をつけるクエリは、リスト10のようになります（実行結果は図12）。

結果に横棒を引いたのは、パーティション（類）の区切りを明確にするためです。最後尾のage_rank_in_classが各パーティション内部での年齢の順位を示す列です。PARTITION BYはGROUP

●図11　リスト9の実行結果

BMI	COUNT(*)
やせている	2
標準	4
肥満	3

●図12　リスト10の実行結果

name	age	age_class	age_rank_in_class
Darwin	12	子供	1
Adela	21	成人	1
Dawson	25	成人	2
Anderson	30	成人	3
Donald	30	成人	3
Bill	39	成人	5
Becky	54	成人	6
Bates	87	老人	1
Chris	90	老人	2

●リスト9　BMIによる体重分類を求めるクエリ

```
SELECT CASE WHEN weight / POWER(height /100, 2) < 18.5     THEN 'やせている'
            WHEN 18.5 <= weight / POWER(height /100, 2)
                   AND weight / POWER(height /100, 2) < 25 THEN '標準'
            WHEN 25 <= weight / POWER(height /100, 2)      THEN '肥満'
            ELSE NULL END AS bmi,
       COUNT(*)
  FROM Persons
 GROUP BY CASE WHEN weight / POWER(height /100, 2) < 18.5     THEN 'やせている'
               WHEN 18.5 <= weight / POWER(height /100, 2)
                      AND weight / POWER(height /100, 2) < 25 THEN '標準'
               WHEN 25 <= weight / POWER(height /100, 2)      THEN '肥満'
               ELSE NULL END;
```

●リスト10　PARTITION BYに式を入れてみる

```
SELECT name,
       age,
       CASE WHEN age < 20 THEN '子供'
            WHEN age BETWEEN 21 AND 69 THEN '成人'
            WHEN age > 70 THEN '老人'
            ELSE NULL END AS age_class,
       RANK() OVER(PARTITION BY CASE WHEN age < 20 THEN '子供'
                                     WHEN age BETWEEN 21 AND 69 THEN '成人'
                                     WHEN age > 70 THEN '老人'
                                     ELSE NULL END
                   ORDER BY age) AS age_rank_in_class
  FROM Persons
 ORDER BY age_class, age_rank_in_class;
```

PARTITION BY句に式を指定している

BYと違って集約を伴わないため、もとのPersonsテーブルの行がすべてそのまま「ヒラ」で出てくることに注目してください。

GROUP BYが式を引数にとれる以上PARTITION BYもまた同様であるということは、論理的には何のためらいもなく得られる結論ではありますが、実際にクエリを目にしてみると「こんなことが可能なのか…」という感慨にとらわれるのではないでしょうか。

❹集合の性質を調べる

これまでは、主に集合の切り分け方のいろいろなバリエーションを紹介してきました。第2章で見たような入れ子の再帰集合を作る方法から、本章で見たような重なり合わないパーティション単位に切り分ける方法まで、GROUP BY句を使えば思うままに集合を組み替えられます。

ここでは、そうして作られた部分集合について、その性質を調べる方法を取り上げます。私たちが頻繁に条件を記述するWHERE句というのは、言うまでもなく「行」に対する条件を記述する場所です。しかし、「行」ではなく「行の集合」を操作の基本単位とするというSQLの原理に照らして見た場合、当然、集合について条件を設定する機能が用意されていて然るべきです。そのために活躍するのが、今からスポットを当てるHAVING句です。

サンプルに、前節❸「あなたは肥り過ぎ？　痩せ過ぎ？ ～カットとパーティション～」で使った年齢階級別に人数を求めるクエリを使いましょう。その基準に従えば、各人は表7のように階級に振り分けられるのでした。

では、ここからさらに、「その階級の全員のBMIが『標準』か『やせ』のどちらかに分類される」階級だけを選択してください。結果は、「子供」と「老人」の階級だけになります。

まず、「子供」「成人」「老人」の3階級に分割した部分集合を作るまでは、先ほどのクエリと同じです。あとはそこに、「階級」という集合自身の性質を調べる条件を付加するだけです。ここでHAVING句が活躍します（**リスト11**。実行結果は**図13**）。

ここで重要な役割を果たすトリックは、HAVING句の右辺で、CASE式で特性関数（戻り値が0または1の関数）を用いていることです。この特性関数

●リスト11　集合の性質を調べるためのクエリ

```
SELECT CASE WHEN age < 20 THEN '子供'
            WHEN age BETWEEN 21 AND 69 THEN '成人'
            WHEN age > 70 THEN '老人'
            ELSE NULL END AS age_class,
       COUNT(*)
  FROM Persons
 GROUP BY CASE WHEN age < 20 THEN '子供'
               WHEN age BETWEEN 21 AND 69 THEN '成人'
               WHEN age > 70 THEN '老人'
               ELSE NULL END
 HAVING COUNT(*) = SUM(CASE WHEN weight / POWER(height /100, 2) < 25 THEN 1
                            ELSE 0 END);
```

●表7　階級の振り分け

name (名前)	age (年齢)	age_class (年齢階級)	Height (身長 cm)	weight (体重 kg)
Anderson	30	成人	188	90
Adela	21	成人	167	55
Bates	87	老人	158	48
Becky	54	成人	187	70
Bill	39	成人	177	120
Chris	90	老人	175	48
Darwin	12	子供	160	55
Dawson	25	成人	182	90
Donald	30	成人	176	53

●図13　リスト11の実行結果

```
age_class    count(*)
------------ ----------
老人           2
子供           1
```

によって、BMIが25未満の人物については「1」、25以上の人物については「0」という、一種のビットフラグを立てています。あとはこのビットフラグを合計した結果が集合の要素数と一致すれば、「全員のビットフラグが1だった」ことが保証されるわけです。

もしもっと直截に「各階級の肥満ではない人物の数」を結果として求めたいなら、リスト12のようにSELECT句でビットフラグの合計を数えてやればよいでしょう（実行結果は図14）。

こうすることで、先ほどは条件を満たさないために結果から除外されていた「成人」階級も結果に現れます。こちらのほうが、成人はAnderson、Bill、Dawsonの3人が「肥満」に該当してしまうのだ、という事実も明確に示す結果となっていることがおわかりいただけるでしょう。

おわりに

SQLの集合操作の勘所を学ぶために、GROUP BY（とPARTITION BY）を中心に解説をしてきました。いかがでしょう。少し、SQLの寄って立つ基礎について理解を深められたでしょうか。もう一度、重要なポイントをまとめておきましょう。

- SQLの基本的な処理単位は、レコード（行）ではなく「レコードの集合」
- GROUP BY句は元のテーブルを小さな部分集合（類）に切り分ける「カット」の機能を持つ
- GROUP BY句はさらに、カットした集合単位に行をまとめる「集約」の機能も持つ
- GROUP BY句から集約の機能を取り去って、カットだけを残したのがPARTITION BY句
- どちらの句も、列名に限らず「式」を引数に取れる柔軟さが強み。これによって、SQLは集合をどんな複雑な基準によってでも切り刻み、料理することが可能となっている

GROUP BYやPARTITION BYは、もちろんそれ単独で見ても、強力で使いでのある武器です。しかし、本章でも見たように、CASE式などのほかの武器と組み合わせたときに、最もその真価を発揮します。そして実は、GROUP BY句と最も相性のよい武器はHAVING句です。最後で少しだけ紹介しましたが、この2つを組み合わせたときの記述力には目を見張るものがあります。これについては、取り上げようとするとかなり紙幅を必要とするので、また機会を改めてお話しすることにしましょう[注6]。

本章の内容について、さらに踏み込んで学びたい方は、以下の参考文献を参照してください。

『SQLパズル　第2版』（J.セルコ 著、翔泳社、2007）

GROUP BYの応用として興味深いのは、「カット」の機能をうまく利用した「パズル29　最頻値を求める」や入れ子の再帰集合を作って累計を求める「パズル35　在庫調整」。HAVING句については、特性関数を駆使する「パズル11　作業依頼」や集合指向の極致と言うべき芸術的な欠番探索の方法を紹介す

●リスト12　集合の性質を調べるためのクエリ

```
SELECT CASE WHEN age < 20 THEN '子供'
            WHEN age BETWEEN 21 AND 69 THEN '成人'
            WHEN age > 70 THEN '老人'
            ELSE NULL END AS age_class,
       COUNT(*) AS all_cnt,
       SUM(CASE WHEN weight / POWER(height /100, 2) < 25 THEN 1
                ELSE 0 END) AS not_fat_cnt
  FROM Persons
 GROUP BY CASE WHEN age < 20 THEN '子供'
               WHEN age BETWEEN 21 AND 69 THEN '成人'
               WHEN age > 70 THEN '老人'
               ELSE NULL END;
```

「肥満」以外の人数を数える式

●図14　リスト12の実行結果

age_class	all_cnt	not_fat_cnt
成人	6	3
老人	2	2
子供	1	1

注6）HAVING句については、『指南書』の中でかなり詳しく取り上げています。先に知りたい人は「1－4．HAVING句の力」や「1－10．帰ってきたHAVING句」を参照。

る「パズル57　欠番探し――バージョン1」など。

『達人に学ぶ　SQL徹底指南書』（ミック、翔泳社、2008）

本書ではHAVING句をかなり重点的に取り上げました。「1-4 HAVING句の力」や「1-10 帰ってきたHAVING句」を参照。一方、HAVING句の復権に力を入れるあまり、GROUP BY句についての説明が薄くなってしまったので、今回の特集と併せて読んでいただくと、より一層効果的でしょう。

特集のおわりに

以上、3つの章にわたって、SQLの基本的なロジックを解説してきました。なるべく手続き型言語の考えにひきつけて、SQLとの架橋（かきょう）を果たそうと試みてきたつもりですが、いかがだったでしょう。

SQLは、基本的には集合と述語という、それ自身、大変しっかりした数学的・論理的な基礎を持つ道具立てをうまく応用した言語ですが、おそらく私たちエンジニアにとっては、正面からこうしたなじみの薄い概念に取り組むよりも、なるべく慣れ親しんだ手続き型の考えからSQLの集合指向に接近するほうが理解しやすいのではないか、と考えて、このような構成をとりました。

この特集を読んで、SQLについて、よりハイレベルな技術を学びたいと思った方（あるいは全然物足りなかった方）は、まずは拙著『SQL徹底指南書』を参照されることを勧めます。その後に、『指南書』の巻末の「参考文献」に挙げた書籍を紐解くことで、より高度で興味深いデータベースの世界が開けていくことでしょう。Web