はじめに

本書は、ずばり「楽しく学ぶSQL中級入門」です。著者はこれまで何冊かSQL中級者(およびそれを目指す初級者)向けの本を書いてきました。幸いなことにいずれも好評をいただき、ちょっとしたロングセラーとなりましたが、やはり中には「内容が難しい」「理論的な話がとっつきづらかった」という感想をいただくことも少なくありませんでした。そこで、何とか技術的なレベルを維持したまま読者が読みやすくなるように敷居を下げる方法はないものかと長い間思案していました。

その問題を解決する試みとして考え出したのが、初級者と上級者の対話形式というスタイルです。初級者の素朴な疑問にメンター(と呼ぶには少し人格的に難があるのですが)となる上級者が的確な回答を返してより高みへと導いていくことで、読者と一緒に成長していくような物語を書けたらと思って作り上げたのが本書です。コミカルで軽妙なやりとりの中で、でも時にはSQLの本質に迫る真剣な技術論を交わす3人の物語を楽しんで最後まで読んでいただければ幸いです。

各章は独立しているので興味を持った箇所から読んでいただいてかまいませんが、もしSQLの構文やCASE式とウィンドウ関数の知識があいまいだと思う方は、序章だけは最初に読むと本書を十分に楽しめるでしょう。この2つの道具はSQLプログラミングを中級レベルに引き上げるためには必須の技術で、本書でも多用するからです。

また、本書では必ずと言ってよいほど、SQL文の実行計画を読むことでそのSQL文の良し悪しを評価します。これは、実行計画がSQL文のパフォーマンスを決定すると言ってよいからです。近年、データベースで扱うデータ量は指数関数的な増加を見せており、SQLに対してもハイパフォーマンスが求められるようになりました。そのため、SQL中級者にはパフォーマンスを意識したコーディングが求められます。と言ってもそれはコードのエレガントさや単純さと矛盾する話ではなく、エレガントでシンプルな

コードほど高速でリソース消費も少ないのだということを、本書を通じて見ていきます。実行計画は、全部理解しようとするとそれだけで一冊の本になるくらい大きなテーマですが(実際、著者は『SQL実践入門』[注1]という実行計画の読み方を解説した本を書いています)、本書ではそれほど複雑な実行計画は出てこないので心配しないでください。

なお、本書の執筆にあたっては『Web+DB PRESS』連載当時から書籍化に至るまで技術評論社の池田大樹氏に一貫してお世話になりました。また、関口裕士氏(「Mac De Oracle」[注2]の中の人)と有限会社アートライの坂井恵氏に本書の査読を行っていただきました。ここに謝辞を記します。

さて、それでは楽しくも奥深いSQLの世界を愉快な3人組と一緒に探検しに行きましょう。

2024年6月20日

ミック

注1 ミック著『SQL実践入門――高速でわかりやすいクエリの書き方』技術評論社、2015年

注2 https://discus-hamburg.cocolog-nifty.com/mac_de_oracle/

本書を読む際の注意事項

動作確認環境

本書のSQL文は、原則として標準SQLに準拠しています。そのため、主要なDBMSの最新版であればほとんどのサンプルコードは動作します。一部、実装依存の箇所については本文中で注意書きをしています。

本書のSQL文およびJavaのコードは主に以下の環境にて動作確認を行いました。

- SQL
  - Oracle Database 21c Express Edition
  - Microsoft SQL Server 2022
  - Db2 11.5
  - PostgreSQL 16.3
  - MySQL Community Server 9.0
- Java
  - JDK：Java SE Development Kit 21.0.1 (64bit)
  - JDBCドライバ[注1]：postgresql-42.7.3.jar
- Python
  - Python 3.12.4
  - psycopg2[注2] 2.9.9

相関名を定義するAS

文中のSQL文において、テーブルの相関名を定義する際に使用するキーワードASを省略しています。これはOracle Database（以下Oracleと呼称）に

注1 PostgreSQL向けのJDBCドライバは下記よりダウンロードできます。
https://jdbc.postgresql.org/

注2 psycopg2は下記よりダウンロードできます。
https://pypi.org/project/psycopg2/

おいてエラーが発生するのを回避するためです。ほかのDBMSにおいても、AS を省略してもエラーにはなりませんので安心してください。

本書に出てくる主要な人名

本書には、何人か頻繁に言及する実在の人物がいます。知らなくても内容の理解には支障はありませんが、予備知識として簡単に解説します。

- E.F. コッド(E.F.Codd:1923-2003)
  IBM社に勤務していた1969年、RDBとSQLの原型となる言語のアイデアを考案した。現代のリレーショナルデータベースの生みの親
- J. セルコ(Joe Celko:1947-)
  RDB/SQLを専門とするコンサルタント。SQLに関する優れた解説書『プログラマのためのSQL』[注3](邦訳版タイトル)を書いている。著者はJ. セルコの本でSQLを勉強して、翻訳も務めた

サンプルコードのダウンロード

本書で利用しているサンプルコードはWebで公開しています。詳細は本書サポートページを参照してください。補足情報や正誤情報なども掲載しています。

- https://gihyo.jp/book/2024/978-4-297-14405-0/support

実行計画の取得方法

本書では、登場するほとんどすべてのSQL文の実行計画を読んでその良し悪しを評価します。実行計画を取得する方法はDBMSによって違いがあり、それぞれ以下のようになっています。それぞれコマンドラインインタフェース(OracleのSQL*PlusやPostgreSQLのpsqlなど)から実行します。

```
Oracle
SET AUTOTRACE TRACEONLY
```

注3 J. セルコ著、ミック監訳『プログラマのためのSQL 第4版——すべてを知り尽くしたいあなたに』翔泳社、2013年

```
Microsoft SQL Server
SET SHOWPLAN_TEXT ON

Db2
EXPLAIN ALL WITH SNAPSHOT FOR <SQL文>

PostgreSQL
EXPLAIN <SQL文>

MySQL
EXPLAIN ANALYZE <SQL文>
```

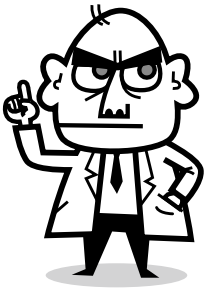
OracleとSQL Serverでは、いずれもコマンド実行後に対象のSQL文を実行します。なおOracleの場合、実行計画の取得においても実際にSQL文が実行されるため、UPDATE文などの更新文を実行した場合はROLLBACKコマンドでデータを元に戻すようにしてください。

また、実行計画を取得する際は**統計情報を最新化**することを忘れないでください。統計情報更新のコマンドは、各製品のマニュアルを参照してください。「統計情報とは何か？」ということがわからない方は、拙著『おうちで学べるデータベースのきほん 第2版』[注4]などの入門向けの書籍を先に読むことをお勧めします。

注4　ミック、木村明治著『おうちで学べるデータベースのきほん 第2版』翔泳社、2024年

本書の登場人物

本書に登場する3人の医師(とそのタマゴ)を紹介します。



ロバート

救命室部長。腕の立つエンジニアだが、口が悪く性格はもっと悪い四十オヤジ

ヘレン

救命室副部長。若いながらもロバートに次ぐ実力を持つ才媛。救命室の良心

ワイリー

インターンで救命室に配属された不運な学生。治療から雑務全般にこき使われる。エンジニアとしては新人に毛が生えたレベル

初出一覧

本書は、技術評論社より定期刊行されていた『Web+DB PRESS』(現在は休刊中) Vol.62〜67に掲載された同名の連載に加筆修正を加えたものです。本連載はgihyo.jpでも公開しています。

- https://gihyo.jp/list/group/SQL緊急救命室

上記以外に初出や出典がある場合は本文中に記します。

章	タイトル	WEB+DB PRESS初出
序章	本書を読むにあたってのSQLの基礎——モダンなSQLの必須技術、CASE式とウィンドウ関数	新規
第1章	サブクエリ・パラノイア——サブクエリの功罪	Vol.62
第2章	冗長性症候群——条件分岐をUNIONで表現するなかれ	Vol.63
第3章	ループ依存症——手続き型の呪縛を打ち破れ！	Vol.64
第4章	スーパーソルジャー病——すべての問題をやみくもにコーディングで解くべからず	Vol.65
第5章	時代錯誤症候群——進化し続けるSQLに取り残されるな！	Vol.66
第6章	ロックイン病——実装依存の罠にはまるな！	新規
第7章	SQLグレーノウハウ——毒と薬は紙一重	新規
第8章	集合指向アレルギー——なぜSQLはエンジニアにとってわかりにくいのか	新規
第9章	リレーショナル原理主義病——ウィンドウ関数は邪道なのか	新規
第10章	更新時合併症——冗長なサブクエリ、性能劣化、実装依存	Vol.67
第11章	ライトスタッフ——正しい資質	新規

SQL緊急救命室——非効率なコードを改善せよ！

目次

第1章

サブクエリ・パラノイア
サブクエリの功罪 43

第2章

冗長性症候群
条件分岐をUNIONで表現するなかれ 81

第6章

ロックイン病
実装依存の罠にはまるな！ 199

第7章

SQLグレーノウハウ
毒と薬は紙一重 229

第8章

集合指向アレルギー
なぜSQLはエンジニアにとってわかりにくいのか 281

第9章

リレーショナル原理主義病
ウィンドウ関数は邪道なのか 309

第10章

更新時合併症
冗長なサブクエリ、性能劣化、実装依存 333

序章

本書を読むにあたってのSQLの基礎

モダンなSQLの必須技術、CASE式とウィンドウ関数

本章で学ぶ内容

本章では、SQLプログラミングに必須の技術であるCASE式とウィンドウ関数の基本的な構文と使い方を学びます。この2つを覚えることによって、SQL初級者から中級者への階段を上ることができます。

CASE式はSQL文のあらゆる句で書ける柔軟なユーザー定義関数であり、ウィンドウ関数は特に分析業務で利用するこちらも強力な表現力を持った関数です。CASE式はUNIONを、ウィンドウ関数は相関サブクエリを消去できるため、クエリが簡潔になり可読性が上がるだけでなく、パフォーマンス向上に大きな力を発揮します。

出会い

ここは、とある街の総合病院。多くの患者が来院し、医師たちがあわただしく診療を行っている。この病院には通常の診療科のほかに、一風変わった診療科が存在する。何軒もの病院をたらいまわしにされた、手の施しようのないSQLや、今すぐに改善が必要なSQLが担ぎ込まれる救命室である。それが**SQL緊急救命室**、略してSER(*SQL Emergency Room*)である。そう、ここは国内でも唯一のプログラミング専門外来である。

今日もさまざまなSQLが持ち込まれているなか、一人のインターンの医学生がやってきた。彼の名前はワイリー、SERに派遣された不運な大学生である。

(9:00、休憩室。ワイリーが持参した白衣に着替えている。)

今日からここで働くのかあ。どんな先生たちがいるんだろうな。いい人たちだといいな。というか、僕に患者の治療なんてできるのかなあ。学校の成績もあんまり良くないのに。ううん、考えたら緊張してきた。

（男女のペアが話しながら休憩室に入ってくる。）


そのおろしたての白衣は……お前か。今日から配属になったインターンというのは。


はい。ワイリーと言います。よろしくお願いします。


ワシはロバート。救命室の部長だ。こっちはヘレン。副部長だ。


よろしくね、ワイリー。


はい。よろしくお願いします。


わざわざこのキツイ救命室を志願してくるとはなかなか度胸のある奴だ。脅しじゃないがここは厳しいぞ。途中でつぶれた学生は数知れん。


つぶれた学生が多いのはあなたがきつく当たるからでしょうが。


さっそくだが、もう今日１件目の患者が来ている。ちょうどいい。お前の今の実力を計ってやるから一緒に手術室に来い。


え、もうですか。その、なんというかチュートリアル的なものは。


ここでは実戦がチュートリアルだ！ つべこべ言わず来い！


ふええ。

CASE式
――SQLが誇る強力なユーザー定義関数

（手術室。ワイリーが慣れない手つきでカルテを用意する。）

ええと、カルテによると患者の容態はこうです。

CASE式の基本的な使い方――ラベルの読み替え

カルテ1 倉庫テーブル（**リスト0-1**）に格納されている倉庫IDから倉庫の存在する都市の名前へ変換するクエリがある（**リスト0-2**、**図0-1**）。今までDBMSとしてOracleを利用していたが今度MySQLにマイグレーションすることになったため、MySQLでも動作するようにクエリを修正したい。

リスト0-1　**倉庫テーブル**

```
CREATE TABLE Warehouse
(warehouse_id  INTEGER NOT NULL PRIMARY KEY,
 region        CHAR(32) NOT NULL);
```

Warehouse:倉庫テーブル　warehouse_id:倉庫ID　region:地域

Warehouse

warehouse_id（倉庫ID）	region（地域）
1	East Coast
2	East Coast
3	West Coast
4	West Coast
5	West Coast

リスト0-2　倉庫IDから拠点の都市を割り出すクエリ

```
SELECT DECODE(warehouse_id, 1, 'New York',
                            2, 'New Jersey',
                            3, 'Los Angels',
                            4, 'Seattle',
                            5, 'San Francisco',
                               'Non domestic') AS city,
       region
  FROM  Warehouse;
```

図0-1　実行結果

```
CITY                      REGION
------------------------- -----------
New York                  East Coast
New Jersey                East Coast
Los Angels                West Coast
Seattle                   West Coast
San Francisco             West Coast
```

テーブル設計からしていろいろ文句をつけたいところはあるが、まあ根本的なところはいったん置いておこう。ワイリー、お前ならどうクエリを治す？

DECODEはOracle固有のコード読み替えの関数ですよね。だからほかのDBMSでは使えない、と。汎用的なコード読み替えの関数ってあったかなあ。

お前、大学で何を習ってきたんだ。いいか、よく見ておけ（**リスト0-3**）。

リスト0-3　CASE式による汎用的なクエリ（検索CASE式）

```
SELECT CASE WHEN warehouse_id = 1 THEN 'New York'
            WHEN warehouse_id = 2 THEN 'New Jersey'
            WHEN warehouse_id = 3 THEN 'Los Angels'
            WHEN warehouse_id = 4 THEN 'Seattle'
            WHEN warehouse_id = 5 THEN 'San Francisco'
            ELSE NULL END AS city,
       region
  FROM Warehouse;
```


ああ、CASE式。そうそう大学で習いました。なんか地味な存在なんで忘れてました。


間抜けなことを言いおって！ CASE式はこれがなければSQLプログラミングができないほどの最重要機能だぞ。

2つのCASE式の構文——単純CASE式と検索CASE式


CASE式には2つの書き方があるわ。1つが**検索CASE式**。さっき見たCASE式の書き方で、WHENのあとに条件分岐の式を記述するわ。もう1つが**単純CASE式**。WHEN句に複雑な条件は記述できないけれど、冗長な表現を省略して書くことができるわ。さっきのCASE式を単純CASE式で書き換えると**リスト0-4**のようになる。

リスト0-4　**単純CASE式の書き方**

```
SELECT CASE warehouse_id
               WHEN 1 THEN 'New York'
               WHEN 2 THEN 'New Jersey'
               WHEN 3 THEN 'Los Angels'
               WHEN 4 THEN 'Seattle'
               WHEN 5 THEN 'San Francisco'
       ELSE NULL END AS city,
      region
  FROM Warehouse;
```


単純CASE式はwarehouse_idの値がWHEN句の値と一致するかどうかだけを調べているのですね。書ける場合は単純CASE式を使うのがよいのでしょうか？


むしろ逆ね。基本的に**単純CASE式は使わないほうがいいわ**。理由は、大した記述量の省略にならないのと、単純CASE式で書ける条件はすべて検索CASE式で書けるから。基本的には常に検索CASE式を使う癖をつけておいたほうがいいわ。


なるほど。まあたしかに単純CASE式にしたからといってものすごく読みやすくなった、というほどでもないですね。

別に単純CASE式にしたことで高速になるといったボーナスもないしな。機能として用意はされているが出番はないと思っていい。

CASE式の注意点

あと、CASE式にはいくつか使ううえでの注意点があるわ。まず1つ目の注意点として、最初に条件が合致したWHEN句が見つかった時点で検索が打ち切られて、後続のWHEN句は参照されない。これを**短絡評価**(*short-circuit evaluation*)と言うわ。JavaやPythonなど一般的な手続き型言語でもIF文などの条件分岐は同じ方式で実装されているから、これは違和感のない評価方式だと思うわ。だから、CASE式を使うときは各WHEN句が必ず排他的な条件になるように記述しないと、「あれ？」と思う動作をしてしまう。たとえば、**リスト0-5**のCASE式ではNew Jerseyに評価されることは絶対にないわ。warehouse_idが2のケースもNew Yorkに吸収されてしまうから(**図0-2**)。

リスト0-5　**CASE式は短絡評価**

```
SELECT CASE WHEN warehouse_id IN (1, 2) THEN 'New York'
            WHEN warehouse_id = 2 THEN 'New Jersey'
            WHEN warehouse_id = 3 THEN 'Los Angels'
            WHEN warehouse_id = 4 THEN 'Seattle'
            WHEN warehouse_id = 5 THEN 'San Francisco'
            ELSE NULL END AS city,
       region
  FROM Warehouse;
```

図0-2　**短絡評価の結果**

```
     city      |             region
---------------+---------------------------------
 New York      | East Coast
 New York      | East Coast
 Los Angels    | West Coast
 Seattle       | West Coast
 San Francisco | West Coast
```

2つ目の注意点は、CASE式の戻り値のデータ型よ。各WHEN句に対応するTHEN句は、すべて同じデータ型である必要があるわ。こんな風に文字列型と数値型を混在させることはできない（**リスト0-6**）。構文エラーになるわ（**図0-3**）[注1]。

リスト0-6　**戻り値は同じデータ型でなければエラーになる**

```
SELECT CASE WHEN warehouse_id = 1 THEN 'New York'
            WHEN warehouse_id = 2 THEN 100
            WHEN warehouse_id = 3 THEN 'Los Angels'
            WHEN warehouse_id = 4 THEN 500
            WHEN warehouse_id = 5 THEN 'San Francisco'
            ELSE NULL END AS city,
       region
  FROM Warehouse;
```

図0-3　**実行結果：エラーメッセージ（PostgreSQL）**

```
ERROR:  "integer"型の入力構文が不正です: "New York"
行 1: SELECT CASE WHEN warehouse_id = 1 THEN 'New York'
```

まあたしかにこれはそのとおりでしょうね。同じ式の戻り値が条件によってデータ型が違ったら混乱してプログラミングやりづらくてしかたないですしね。

あと最後にもう1つ。どんな場合でも**必ずELSE句は書く**こと。文法的にはELSE句はなくてもエラーにはならない。ただその場合、暗黙にELSE NULLが指定されたものとみなされるので、意図しないNULLの発生を引き起こすことになるの。**NULLはSQLの鬼門**だから、可能な限りその発生は抑制するべきよ。仮にNULLが戻り値で良い場合でも明示的にELSE NULLを書くべきね。

NULLってそんなにやっかいなんですか？ 値がわからない場合にNULLを入力するのはすごく直観的でわかりやすいと思うのですけど。

その一見したわかりやすさがNULLの困ったところよ。するっと私たちの認識に入り込んできて面倒な問題をもたらす。まあNULLの

注1　MySQLでは暗黙の型変換が行われるためエラーになりませんが、汎用性のない記述方法のため使用するべきではありません。

怖さは治療にあたっていればおいおい見ることになるわ(NULLのやっかい者ぶりについては第6章を参照)。

CASE式の構文と注意点については理解しました。でも、CASE式のやってることって、単に**ラベルの読み替え**ですよね。1ならNew York、2ならNew Jerseyって具合に一対一対応させてコードから名前に読み替えてるわけですよね。言ってみれば値のラベルを張り替えているだけじゃないですか。これがそんなに重要な機能なんでしょうか。

お前の言うとおりだ。CASE式はラベルの読み替えを行っているにすぎない。だがそのラベルの読み替えがお前の思っている以上に強力なのだ。ちょうどいい。次の患者が来たぞ。いい勉強になるケースだ。

SELECT句でCASE式を使う
――CASE式による行列変換(ピボット)

(救急隊員が手術台の上に次の患者を担ぎ上げる)

ええとこの患者のカルテは……

カルテ2 都市テーブル(**リスト0-7**)に保存されている人口データを次のようなフォーマットで表示するクエリを考えたい(**図0-4**、**リスト0-8**)。

リスト0-7 **都市テーブル**

```
CREATE TABLE City
(city   CHAR(32) NOT NULL ,
 population INTEGER NOT NULL,
   CONSTRAINT pk_City PRIMARY KEY (city));
```

City:都市テーブル　city:都市名　population:人口

City

city (都市名)	population (人口)
New York	8,460,000
Los Angels	3,840,000
San Francisco	815,000
New Orleans	377,000

図0-4　**求めたい結果**

```
New York | Los Angels | San Francisco | New Orleans
---------+------------+-------------- +------------
8460000  | 3840000    | 8115000       | 377000
```

リスト0-8　**患者のコード(SQL Server)**

```
SELECT [New York], [Los Angels], [San Francisco], [New Orleans]
FROM
(
  SELECT city, population
  FROM City
) AS SourceTable
PIVOT
(
  SUM(population)
  FOR city IN ([New York], [Los Angels], [San Francisco], [New Orleans])
) AS PivotTable;
```


へええ、SQLで行列変換ができるんですね。行持ちのテーブルから列持ちの結果を得ているんですね。おもしろい。


SQLは本来データのフォーマッティングをするための言語ではないので、こういう使い方は少し**正道から外れているわ**。そもそもSQLでピボットなんかするべきではないの。でも実際の開発現場では求められることの多い要件であることも事実ね。だからベンダーも関数を用意していることがあるわ。


それで、このクエリの何が問題なんでしょう。いまいちピンとこないんですけど。

問題はPIVOT関数が実装依存でSQL ServerやRedshift、Snowflakeなど一部のDBMSでしか使えないことだ[注2]。標準SQLにもPIVOT関数は入っていないし、実装ごとに構文も微妙に違う。まあやっかい者だ。

ああ、そういうことか。汎用性がないんですね。

うむ。こういう実装依存の関数は**ロックイン病**の原因となり、マイグレーションのときに泣くことになる。SQLにおけるピボットはCASE式を使ってやるのがセオリーだ(**リスト0-9**、**図0-5**)。

リスト0-9　**ロバートの解：CASE式を使うピボット**

```
SELECT SUM(CASE WHEN city = 'New York'
                THEN population ELSE 0 END)  AS "New York",
       SUM(CASE WHEN city = 'Los Angels'
                THEN population ELSE 0 END)  AS "Los Angels",
       SUM(CASE WHEN city = 'San Francisco'
                THEN population ELSE 0 END)  AS "San Francisco",
       SUM(CASE WHEN city = 'New Orleans'
                THEN population ELSE 0 END)  AS "New Orleans"
  FROM City;
```

図0-5　**実行結果**

```
New York | Los Angels | San Francisco | New Orleans
---------+------------+-------------- +------------
8460000  | 3840000    | 8115000       | 377000
```

これは驚きました。CASE式をSUM関数の中に入れる使い方ができるんですね。求めたい都市のラベルのときだけpopulation列の値

注2　PIVOT関数は比較的新しいデータベースで採用されており、もしかすると近いうちに標準SQLに入る可能性もあります。
SQL Serverでのピボット関数は以下を参照。
「FROM - PIVOT および UNPIVOT の使用」
https://learn.microsoft.com/ja-jp/sql/t-sql/queries/from-using-pivot-and-unpivot?view=sql-server-ver16
Snowflakeでの PIVOT関数は以下を参照。
「PIVOT - Snowflake Documentation」
https://docs.snowflake.com/ja/sql-reference/constructs/pivot
Redshiftでの PIVOT関数は以下を参照。
「PIVOT と UNPIVOT の例 - Amazon Redshift」
https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_FROM_clause-pivot-unpivot-examples.html

を持ってきて、それ以外の都市だったら0を指定して集計対象外にする、というわけですね。ところでこのSUM関数って必要なんでしょうか。別にこのクエリ、人口を**合計しているわけではない**ですよね。各都市に人口は一つしかないわけだし。

SUM関数は必要よ。正確に言うと、ここはAVGでもMAXでもMINでもいいのだけどね。なぜ必要かはSUM関数なしの結果を見てみればわかるわ(**リスト0-10**、**図0-6**)。

リスト0-10　**SUM関数なしでCASE式を使うと**

```
SELECT CASE WHEN city = 'New York'
            THEN population ELSE 0 END  AS "New York",
       CASE WHEN city = 'Los Angels'
            THEN population ELSE 0 END  AS "Los Angels",
       CASE WHEN city = 'San Francisco'
            THEN population ELSE 0 END  AS "San Francisco",
       CASE WHEN city = 'New Orleans'
            THEN population ELSE 0 END  AS "New Orleans"
  FROM City;
```

図0-6　**実行結果**

```
New York | Los Angels | San Francisco  | New Orleans
---------+ -----------+----------------+------------
0        | 0          | 3840000        | 0
0        | 0          | 0              | 377000
8460000  | 0          | 0              | 0
0        | 815000     | 0              | 0
```

あー……なるほど、SUM関数がないと集約されないから全部の行が出力されてしまうのか。これは余計ですね。

そういうこと。SELECT句でCASE式を使うときは集約関数と組み合わせることが多いから、覚えておくことね。

UNIONで条件分岐するのは正しいのか

次の患者もなかなかおもしろい。ワイリー、お前ならどうやって解くか考えてみろ。


はい、ええとカルテはこれか。

カルテ3 商品を管理するテーブルItemPriceが存在する。各商品について、税抜き価格(外税)／税込み価格(内税)の両方を保持している。2004年から、法改正によって価格表示に税込み価格(内税)を表示することが義務付けられた(いわゆる「総額表示の義務付け」)。そこで、2003年までは税抜き価格を、2004年からは税込み価格を「価格」列として表示する結果(**図0-7**の色の付いていない部分)を求めたい(**図0-8**)。

図0-7 **ItemPrice(商品価格テーブル)**

ItemPrice(商品価格テーブル)

item_id (商品 ID)	year (年)	item_name (商品名)	price_tax_ex (価格 _ 外税)	price_tax_in (価格 _ 内税)
100	2002	カップ	500	525
100	2003	カップ	520	546
100	2004	カップ	600	630
100	2005	カップ	600	630
101	2002	スプーン	500	525
101	2003	スプーン	500	525
101	2004	スプーン	500	525
101	2005	スプーン	500	525
102	2002	ナイフ	600	630
102	2003	ナイフ	550	577
102	2004	ナイフ	550	577
102	2005	ナイフ	400	420

図0-8 **求める結果**

```
 item_name | year | price
-----------+------+-------
 カップ    | 2002 |   500
 カップ    | 2003 |   520
 カップ    | 2004 |   630
```

```
カップ    | 2005 |   630
スプーン  | 2002 |   500
スプーン  | 2003 |   500
スプーン  | 2004 |   525
スプーン  | 2005 |   525
ナイフ    | 2002 |   600
ナイフ    | 2003 |   550
ナイフ    | 2004 |   577
ナイフ    | 2005 |   420
```


条件分岐問題の基礎ね。year列の値を分岐の条件に使う、と。ワイリー、あなたならどう解く？


任せてください！ これならわかります。レコードに条件を指定するわけだから、WHERE句を使えばいいんですよ。あとはそれをUNIONでつなげれば……はい、できました！（**リスト0-11**）この解のポイントはですね、UNIONの代わりにUNION ALLを使うことでソートを回避して性能改善も図っていることです。条件が排他的だから問題ないわけです[注3]。

リスト0-11　**ワイリーの解答（UNIONを使う）**

```
SELECT item_name, year, price_tax_ex AS price
  FROM ItemPrice
 WHERE year <= 2003
UNION ALL
SELECT item_name, year, price_tax_in AS price
  FROM ItemPrice
 WHERE year >= 2004;
```


イタタタ……。


足の小指でもぶつけました？


いや、そうじゃなくて、あなたの解を見てアタマ痛くなったの！ もう、先が思いやられるわ……。

注3　ワイリーの言っていることは間違いではありません。たしかにUNION ALLはソートをスキップしますが、今の問題はそこではありません。

ヘレンがなぜ頭痛を覚えてしまったのか、詳しく見ていきましょう。ワイリーの解は、機能的には問題ありません。正しい結果を得られるクエリになっています。問題は、一言で言うと冗長であることです。ほとんど同じ中身のクエリを両方実行しているからです。これは、SQLを無駄に長くして読みにくくするだけでなく、パフォーマンス上も無駄です。


お前、大学で実行計画(アクセスプラン)の読み方は習ったか？


はあ、基本的なところは。


よし、ではこのクエリの実行計画を表示してみろ。PostgreSQLとOracleで取れ[注4]。


少々お待ちを、ええと、実行計画を表示するコマンドは……Oracleだと SQL*Plus で set autotrace traceonly コマンドのあとにクエリ、PostgreSQL は psql で explain のあとにクエリか……はい、実行計画取れました(**図0-9、図0-10**)。

図0-9　**ワイリーのクエリの実行計画(PostgreSQL)**

```
                          QUERY PLAN
------------------------------------------------------------------------
 Append  (cost=0.00..36.72 rows=394 width=90)
   ->  Seq Scan on itemprice  (cost=0.00..17.38 rows=197 width=90)
         Filter: (year <= 2003)
   ->  Seq Scan on itemprice itemprice_1
         (cost=0.00..17.38 rows=197 width=90)
         Filter: (year >= 2004)
```

図0-10　**ワイリーのクエリの実行計画(Oracle)**

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    13 |   338 |     4   (0)| 00:00:01 |
|   1 |  UNION-ALL         |           |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| ITEMPRICE |     7 |   182 |     2   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| ITEMPRICE |     6 |   156 |     2   (0)| 00:00:01 |
```

注4　本書では今後実行計画を見る際にPostgreSQLとOracleのものを使いますが、ほかのDBMSでもほぼ同様の実行計画になります。

```
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("YEAR"<=2003)
   3 - filter("YEAR">=2004)
```

PostgreSQLでは「Seq Scan」(シーケンシャル・スキャン)、Oracleでは「TABLE ACCESS FULL」というテーブル全体へのスキャンが発生していることが見て取れます。ワイリーの解はItemsテーブルに対して2度のアクセスを実行しているのです。これは大きな無駄です。シーケンシャルスキャンのコストはデータ量に線形に伸びていくので、テーブルサイズが大きくなればなるほど線形でパフォーマンスも悪化していきます[注5]。

UNIONはたしかに便利な道具です。簡単にレコード集合をマージできるため、ともするとこれを条件分岐のためのツールとして使いたい誘惑に駆られます。しかし、これは危険思想です。ワイリーのように、SELECT文全体をUNIONで連ねて冗長なコードを記述したくなる誘惑を**冗長性症候群**と呼びます(第2章でもっと詳しく見ます)。

それでは、SQLにおける正しい条件分岐の書き方がどうなるか、ヘレンにお手本を見せてもらいましょう。

いい？ SQLを使ううえで、条件分岐をWHERE句で行うのは素人のやることよ。プロはSELECT句で分岐させるの(リスト0-12)。

リスト0-12　**ヘレンの解答(CASE式を使う)**

```
SELECT item_name, year,
       CASE WHEN year <= 2003 THEN price_tax_ex
            WHEN year >= 2004 THEN price_tax_in
            ELSE NULL END AS price
  FROM ItemPrice;
```

ああ、ここでもCASE式なんだ。思いつかなかったなあ。

簡単な判定方法としては、もし「この問題を手続き型言語で解いた

注5　実際のDBMSではキャッシュが働くのでもう少し話が複雑ですが、原則としてです。

ら？」と考えたとき、if文を使う個所があれば、それをSQLに翻訳したらCASE式を使う、と思うことね。まあ慣れれば即座に判断が付くようになるわ。そうね。100例も症例を見れば。


うーん、道は遠そうだ。


さて、ヘレンの解の実行計画を見ておこう。なぜCASE式が優れているかがよくわかる(**図0-11**、**図0-12**)。

図0-11　**ヘレンの解の実行計画(PostgreSQL)**

```
                        QUERY PLAN
------------------------------------------------------------
 Seq Scan on itemprice  (cost=0.00..18.85 rows=590 width=90)
```

図0-12　**ヘレンの解の実行計画(Oracle)**

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |    12 |   312 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| ITEMPRICE |    12 |   312 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```


え、たったこれだけなんですか？


そう、ItemPriceテーブルへのアクセス一回こっきり。シンプル・イズ・ベストでしょう。


SELECT句でCASE式の計算をやってるからもっと複雑な実行計画になるかと思ってました。


この程度の計算、最近のCPUをもってすれば大した負荷じゃないわ。それよりテーブルアクセスを削減できる効果のほうが圧倒的メリットよ。何しろデータベースではいかにしてストレージのI/Oを減らすかが勝負だからね。

WHERE句でCASE式——条件式の列を切り替える

カルテ4 先ほどのItemPriceテーブルを使って、2003年までは外税の価格、2004年以降は内税の価格を使ってそれぞれ600円以上の商品を選択したい。

年によって調べる列をprice_tax_ex(価格_外税)とprice_tax_in(価格_内税)で切り替えるということですね。レコードに対する条件指定だからWHERE句を使うのはわかるんですけど、レコードによって条件に使う列が変わるわけですよね。たぶんこれでいけるような……(**リスト0-13、図0-13**)。

リスト0-13 **ワイリーの解(WHERE句でCASE式)**

```
SELECT item_name, year
  FROM ItemPrice
 WHERE 600 <= CASE WHEN year <= 2003 THEN price_tax_ex
                   WHEN year >= 2004 THEN price_tax_in
                   ELSE NULL END;
```

図0-13 **実行結果**

```
 item_name | year
-----------+------
 カップ    | 2004
 カップ    | 2005
 ナイフ    | 2002
```

あら、いいじゃない。CASE式の使い方、だいぶスマートになってきたわよ。

えへへ。CASE式って大きいからこれが**全体で一つの値に定まる**って感覚がまだ持てないですけど。

たしかにCASEからENDまでが長いからこれがスカラ値に収束するというのが最初は不思議な気がするものよ。そのうち慣れるわ。

GROUP BY句でCASE式の列を参照する
――アドホックな集計キー


CASE式の強力さがわかる問題をもう一つ出してやろう。

カルテ5 Cityテーブル(リスト0-7)から都市が存在する地域によって人口を集計したい。たとえば、New YorkとNew OrleansはEast Coast、San FranciscoとLos AngelsはWest Coastにまとめて人口を集計したい。


うーん、地域を示す列がCityテーブルにあればそれをGROUP BY句に指定して簡単に集計できるのだけどなあ……どうやるんだろう。


考え方はそのとおりよ。列がないならね、**作ってあげればいいの(リスト0-14、図0-14)**。

リスト0-14　**集計単位を変換して人口の合計を求める**

```
SELECT CASE WHEN city IN ('New York', 'New Orleans')     THEN 'East Coast'
            WHEN city IN ('San Francisco', 'Los Angels') THEN 'West Coast'
            ELSE NULL END AS region,
       SUM(population) AS sum_pop
  FROM City
 GROUP BY region;
```

図0-14　**実行結果**

```
   region   | sum_pop
------------+---------
 West Coast | 4655000
 East Coast | 8837000
```


これはすごい。集約キーの列がないからCASE式で強引に作ってるんですね(region列)。CASE式の中でIN述語を使うことで柔軟な条件分岐ができるんですね。これは便利だなあ。アドホックに集計したいときにどんな集約キーでも作れる。

CASE式のWHEN句には、式が書けるからさまざまな述語や演算子が利用できるわ。INのほかにもLIKE、BETWEEN、EXISTSなど各種述語を使うことができる。その柔軟さもCASE式の魅力の一つね。あと、このクエリで注意が必要なのは、SELECT句のCASE式で定義したregion列をGROUP BY句で参照しているのだけど、この構文が使えるのはOracle、PostgreSQLとMySQLだけで、ほかのDBMSではGROUP BY句にregion列を指定するとエラーになるわ(**図0-15**)[注6]。

図0-15　**エラーメッセージ(SQL Server)**

```
Msg 207, Level 16, State 1, Line 6
列名 'region' が無効です。
```

これは、SQL文の評価の順序として本来はGROUP BY句のほうがSELECT句より先に来るから、その時点ではまだregion列が存在していないとみなされるためよ。そういう場合は、ちょっと面倒だけどSELECT句のCASE式と同じCASE式をGROUP BY句にも書いてあげることで対応できるわ(**リスト0-15**)。

リスト0-15　**OracleとPostgreSQLとMySQL以外での書き方**

```
SELECT CASE WHEN city IN ('New York', 'New Orleans')     THEN 'East Coast'
            WHEN city IN ('San Francisco', 'Los Angels') THEN 'West Coast'
            ELSE NULL END AS region,
       SUM(population) AS sum_pop
  FROM City
 GROUP BY CASE WHEN city IN ('New York', 'New Orleans')     THEN 'East Coast'
               WHEN city IN ('San Francisco', 'Los Angels') THEN 'West Coast'
               ELSE NULL END;
```

二度同じCASE式を書かねばならないのはちょっと冗長ですね。

そうね。早くほかのDBMSも対応してくれるといいのだけど。なくても冗長に書けばなんとかなっちゃう機能だからベンダーとしても対応の優先度が低いのでしょうね。わりと放っておかれがちな機能よ。

注6　Oracleは長らくこの構文をサポートしていませんでしたが、Oracle Database 23aiからGROUP BY句でのエイリアス列名を指定できるようになりました。


なぜCASE式が重要か、お前にもわかってきたか。一言で言えば、CASE式はSQL文において**ユーザー定義関数**を記述できるのだ。この柔軟さを使いこなすことによってSQLプログラミングのレベルは飛躍的に向上する！ まさにアメイジング・グレイスな機能なのだ！


CASE式、たしかに便利だなあ。ちょっとすごさがわかってきました。

ORDER BY句でCASE式――任意の順番でソート


おっと、CASE式の柔軟性がよくわかる症例が来たぞ。これもなかなかおもしろい。

カルテ6 Cityテーブル（リスト0-7）から**図0-16**のような順序でレコードを取得したい。

図0-16　**任意の順序で結果を得る**

```
       city
------------------
 New Orleans
 San Francisco
 New York
 Los Angels
```


この順序には辞書順（ABC順）のような明確なルールがあるわけではない。アドホックに決められた無秩序な順番だ。


出力するレコードの順序を制御するにはORDER BY句を使うんですよね。でも明確なルールのない順番でソートするってどうやるんでしょうか。


ここでも考え方は同じ。明確なルールがなければ、作ってしまえばいいの。CASE式を使ってね（**リスト0-16**）。

リスト0-16　**ヘレンの解：ORDER BY句でCASE式を使う**

```
SELECT city
  FROM City
 ORDER BY CASE WHEN city = 'New Orleans'   THEN 1
               WHEN city = 'San Francisco' THEN 2
               WHEN city = 'New York'      THEN 3
               WHEN city = 'Los Angels'    THEN 4
          ELSE NULL END;
```

こうやって整数型に置換してやれば、ORDER BY句のソート順はデフォルトで昇順だから、そのままソートされるわ。もし降順でソートしたいならDESCキーワードを付ければOKよ。

なるほどなあ。CASE式はこれ全体で**一つの列**みたいなものなんですね。

列もCASE式も最終的には評価されて**一つのスカラ値**に定まるという点で同じだからね。CASE式が文（statement）ではなく式（expression）であるというのは、SQLにとってとても本質的なことなのよ。

UPDATE文でもCASE式——値をくるっと入れ替える

カルテ7　実はCityテーブル（リスト0-7）のNew YorkとLos Angelsの人口データが逆だった。正しくなるように更新したい（**リスト0-17**、**図0-17**）。

リスト0-17　**値の入れ替え：患者のUPDATE文**

```
UPDATE City
   SET population = 3840000
 WHERE city = 'New York';

UPDATE City
   SET population = 8460000
 WHERE city = 'Los Angels';
```

図0-17　実行結果

```
             city              | population
-------------------------------+------------
 New York                      |    3840000
 Los Angels                    |    8460000
 San Francisco                 |     815000
 New Orleans                   |     377000
```


結果も合ってるし、この患者の何が問題なんですか？


UPDATE文を2回実行しているのが高コストね。今は主キーのインデックスを使えているからパフォーマンスは良いほうだけど、常にそう都合良く行を絞り込めるとは限らないわ。こういう場合は一つのUPDATE文にまとめてあげるの（**リスト0-18**）。

リスト0-18　ヘレンの解：UPDATE文でCASE式を使う

```
UPDATE City
   SET population = CASE WHEN city = 'New York'   THEN 3840000
                         WHEN city = 'Los Angels' THEN 8460000
                         ELSE population END
 WHERE city IN ('New York', 'Los Angels');
```


へえ！ UPDATE文のSET句でもCASE式が使えるんですね。これは便利だ。くるっと2行の値をいっぺんに入れ替えるんですね。CASE式って本当にどこでも使えるんだなあ。


スカラサブクエリを使えば、動的に人口の値を取ってくるように一般化することも可能よ（**リスト0-19**）。

リスト0-19　スカラサブクエリで一般化したUPDATE文（MySQLではエラーになる）

```
UPDATE City
   SET population = CASE WHEN city = 'New York'   THEN
                             (SELECT population
                                FROM City WHERE city = 'Los Angels')
                         WHEN city = 'Los Angels' THEN
                             (SELECT population
                                FROM City WHERE city = 'New York')
                         ELSE population END
 WHERE city IN ('New York', 'Los Angels');
```


たしかに、これならいちいち各都市の人口が何人か調べなくても更新できますね。


ただ、この解だとどうしてもテーブルへのアクセスが増えちゃうのが欠点なんだけどね。


ではこのクエリの実行計画を見てみるとしよう。PostgreSQLとOracleで取れ。


少々お待ちを……はい、実行計画取れました(**図0-18**、**図0-19**)。

図0-18　**スカラサブクエリの解：実行計画(PostgreSQL)**

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Update on city  (cost=2.10..3.16 rows=0 width=0)
   InitPlan 1 (returns $0)
     ->  Seq Scan on city city_1  (cost=0.00..1.05 rows=1 width=4)
           Filter: (city = 'New York'::bpchar)
   InitPlan 2 (returns $1)
     ->  Seq Scan on city city_2  (cost=0.00..1.05 rows=1 width=4)
           Filter: (city = 'Los Angels'::bpchar)
     ->  Seq Scan on city  (cost=0.00..1.06 rows=2 width=10)
           Filter: (city = ANY ('{"New York","Los Angels"}'::bpchar[]))
```

図0-19　**スカラサブクエリの解：実行計画(Oracle)**

```
-------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT             |         |     2 |    74 |     6  (34)| 00:00:01 |
|   1 |  UPDATE                      | CITY    |       |       |            |          |
|*  2 |   TABLE ACCESS FULL          | CITY    |     2 |    74 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS BY INDEX ROWID| CITY    |     1 |    37 |     1   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | PK_CITY |     1 |       |     1   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| CITY    |     1 |    37 |     1   (0)| 00:00:01 |
|*  6 |    INDEX UNIQUE SCAN         | PK_CITY |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("CITY"='Los Angels' OR "CITY"='New York')
   4 - access("CITY"='New York')
   6 - access("CITY"='Los Angels')
```

なるほど、テーブルへのスキャンが3回発生していますね。利便性と引き換えにパフォーマンスが犠牲になるわけだ。ところで、Oracleの実行計画に出ているINDEX UNIQUE SCANって何ですか？

主キーの一意制約で作られたインデックスに対する1行に絞り込める場合のスキャンで、とても高速よ。WHERE city = '<都市名>'の条件があるから可能になっているの。インデックスの恩恵を受けられるのは心強いわね。

実行計画の読み方

実行計画は、データベースがSQL文を実行する際に立てるテーブルやインデックスに対するアクセスパスを記述したもので、SQL文を実行する際にはDBMSは必ず実行計画を立てます。一つのSQLに対して可能な実行計画が複数ある場合もあり、テーブルの件数や値の分散の度合いなど複数の条件を考慮して(ほぼ)最適なものが選択されるようになっています(ときどきそうでないケースもあり、その場合は実行計画を人間が制御してやるのですが)。このような実行計画を立てるDBMSのモジュールを**オプティマイザ**と呼びます。

実行計画には非常に多くの情報が出力されるため、最初に見たときは面食らってしまうかもしれませんが、重要なポイントを押さえておけば読み方は難しくありません。

まず、ツリー形式で表示されますが、ネストが一番深い箇所から浅い箇所に向かって順に実行されます。同じレベルにある処理は上から順に実行されます。

そしてどのようなオブジェクトを対象にどんな処理が行われているかが示されていますが(OracleであればOperation列とName列)、ここが一番重要で、残りの**Cost**や**Bytes**などの数値列はとりあえず深く考えなくても大丈夫です。

本書では今後、ほぼすべてのSQL文について実行計画を見ながらSQL文の良し悪しを評価していきます。これは、実行計画によってSQL文のパフォーマンスが決定されるからで、SQLの良し悪しを評価するには必ず実行計画のレベルで見なければなりません。ただ、おおむねSQLの複雑さと実行計画の複雑さは比例しており、SQLをなるべくシンプルでエレガントに書くことで、実行計画も良いものが作られるようになっています。したがって本書の目的は、「**エレガントなSQLを書く**」ことでもあります。それがエレガントな実行計画にもつながるからです。

魔法のツール、ウィンドウ関数

（手術室の電話が鳴る。救急隊員からの受け入れ要請のようだ。）

はい……ええ、大丈夫です。すぐに対応できます。

（受話器を置きながら）急患だそうです。あと10分で救急車が到着します。

よし準備しろ。ウォーミングアップでだいぶ体もあったまってきたぞ！

累計とウィンドウ関数

カルテ8 アイスクリーム店の店舗ごとの日々の売り上げを記録するSalesIcecreamテーブル（**リスト0-20**）がある。このテーブルから店舗ごとの売り上げについて現在までの累計を求めたい。

リスト0-20 **アイスクリーム売り上げテーブル**

```
CREATE TABLE SalesIcecream
(shop_id   CHAR(4) NOT NULL,
 sale_date DATE NOT NULL,
 sales_amt INTEGER NOT NULL,
   CONSTRAINT pk_SalesIcecream PRIMARY KEY(shop_id, sale_date) );
```

SalesIcecream:アイスクリーム売り上げテーブル　shop_id:店舗ID　sale_date:売上日
sales_amt:売上金額

アイスクリーム売り上げテーブル

SalesIcecream

shop_id (店舗 ID)	sale_date (売上日)	sales_amt (売上金額)
A	2024-06-01	67,800
A	2024-06-02	87,000
A	2024-06-05	11,300
A	2024-06-10	9,800
A	2024-06-15	9,800
B	2024-06-02	178,000
B	2024-06-15	18,800
B	2024-06-17	19,850
B	2024-06-20	23,800
B	2024-06-21	18,800
C	2024-06-01	12,500


ふむ。売り上げの分析を行おうというわけだな。それで患者のクエリは。


はい、こちらに(**リスト0-21**、**図0-20**)。

リスト0-21　**患者のクエリ：相関サブクエリで累計を求める**

```
SELECT shop_id, sale_date, sales_amt,
       (SELECT SUM(sales_amt)
          FROM SalesIcecream SI1
         WHERE SI1.shop_id = SI2.shop_id
           AND SI1.sale_date <= SI2.sale_date) AS cumlative_amt
  FROM SalesIcecream SI2;
```

図0-20　**患者の実行結果**

```
 shop_id | sale_date  | sales_amt | cumlative_amt
---------+------------+-----------+---------------
 A       | 2024-06-01 |     67800 |         67800  ← (67800)
 A       | 2024-06-02 |     87000 |        154800  ← (67800 + 87000)
 A       | 2024-06-05 |     11300 |        166100  ← (67800 + 87000 + 11300)
 A       | 2024-06-10 |      9800 |        175900  ← (67800 + 87000 + 11300 + 9800)
 A       | 2024-06-15 |      9800 |        185700  ← (67800 + 87000 + 11300 + 9800 + 9800)
```

```
                                                    相関サブクエリによってここで累計がリセットされる
B       | 2024-06-02 |    178000 |       178000  ← (178000)
B       | 2024-06-15 |     18800 |       196800  ← (178000 + 18800)
B       | 2024-06-17 |     19850 |       216650  ← (178000 + 18800 + 19850)
B       | 2024-06-20 |     23800 |       240450  ← (178000 + 18800 + 19850 + 23800)
B       | 2024-06-21 |     18800 |       259250  ← (178000 + 18800 + 19850 + 23800 + 18800)
                                                    相関サブクエリによってここで累計がリセットされる
C       | 2024-06-01 |     12500 |        12500  ← (12500)
```

まったく、醜いクエリだ。見るに堪えんよ。こんなクエリは淘汰（とうた）されてしまえばいいのだ！

ああ、相関サブクエリ……にスカラサブクエリも兼ねてるのかな。大学で習いました。行間比較に使うんですよね。このクエリそんなに悪いですか？ オーソドックスな書き方に見えますけど。

ふん、その教科書は古いな。捨ててしまえ。モダンSQLではもう**相関サブクエリはお払い箱**だ。理由はわかるか？

読みにくいからですか？

それもある。だが最大の理由は、相関サブクエリのパフォーマンスが悪いからだ。実行計画を見てみよう（**図0-21**、**図0-22**）。

図0-21　**患者の実行計画（PostgreSQL）**

```
                              QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on salesicecream si2  (cost=0.00..14.06 rows=11 width=21)
   SubPlan 1
     ->  Aggregate  (cost=1.17..1.18 rows=1 width=8)
           ->  Seq Scan on salesicecream si1
                 (cost=0.00..1.17 rows=1 width=4)
                 Filter: ((sale_date <= si2.sale_date)
                      AND (shop_id = si2.shop_id))
```

図0-22　**患者の実行計画（Oracle）**

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |               |     8 |   136 |    12   (0)| 00:00:01 |
```

```
|   1 |  SORT AGGREGATE    |               |     1 |    17 |            |          |
|*  2 |   TABLE ACCESS FULL| SALESICECREAM |     1 |    17 |     2   (0)| 00:00:01 |
|   3 |  TABLE ACCESS FULL | SALESICECREAM |     8 |   136 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("SI1"."SALE_DATE"<=:B1 AND "SI1"."SHOP_ID"=:B2)
```


この実行計画を見て気付くことはある？


どちらもSalesIcecreamへのアクセスが2回発生していますね。PostgreSQLなら**Seq Scan**(シーケンシャル・スキャン)、Oracleなら**TABLE ACCESS FULL**が2度現れています。


そう、このクエリのダメなところは、テーブルへのアクセスが2度発生することで無駄なストレージへのI/Oが発生していることよ。SQLコーディングにおいては、いかにしてストレージへのI/Oを減らすかがパフォーマンス上重要なの。**「ストレージに触る者は不幸になる」**という格言があるぐらいよ(※著者が作った格言なので一般的ではありません)。SSDが普及してだいぶマシになったとはいえ、それでもストレージはメモリと比べると、レイテンシがとても高い部類に入るからね[注7]。

PARTITION BY句とORDER BY句の使い方


でもサブクエリを使う以上、テーブルアクセスが2回現れるのは避けられないのではないですか？


そうだ。だからサブクエリを使わないことが解になる。正しい解法

注7 ストレージにデータを取りにいくから遅くなるのなら、そもそもデータをメモリに持ってしまえばよいという発想で作られたのがインメモリデータベースです。SAP社のHANAやOracle社のTimesTen In-Memory Databaseなどの製品があります。またチューニングにおいても、Oracleのようにテーブルやインデックスをメモリ上に固定するという手段(keep buffer)が用意されているDBMSもあります。ただ、現在一般的に使われているメモリは揮発性があるためどうしても永続層としてのストレージを必要とします。

はこうだ(**リスト0-22**)。

リスト0-22　**ロバートの解：ウィンドウ関数(得られる結果は相関サブクエリと同じ)**

```
SELECT shop_id, sale_date, sales_amt,
       SUM(sales_amt) OVER (PARTITION BY shop_id
                            ORDER BY sale_date) AS cumlative_amt
  FROM SalesIcecream;
```


えっと、これウィンドウ関数……でしたっけ。まだ大学で習ったばかりなんですけど。たしかにコードが短くなりますね。PARTITION BY句で計算対象の行集合を区切って、ORDER BY句で行を順序付けしてるんだ[注8]。


ウィンドウ関数の良いところは、コードが簡潔でエレガントになるだけではない。実行計画もまたシンプルになり、パフォーマンスが向上する。実行計画を見てみろ。


はい、ただいま(**図0-23**、**図0-24**)。

図0-23　**ロバートの解の実行計画(PostgreSQL)**

```
                          QUERY PLAN
------------------------------------------------------------------------
 WindowAgg  (cost=1.30..1.52 rows=11 width=21)
   ->  Sort  (cost=1.30..1.33 rows=11 width=13)
         Sort Key: shop_id, sale_date
         ->  Seq Scan on salesicecream
               (cost=0.00..1.11 rows=11 width=13)
```

図0-24　**ロバートの解の実行計画(Oracle)**

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |               |     8 |   136 |     3  (34)| 00:00:01 |
|   1 |  WINDOW SORT       |               |     8 |   136 |     3  (34)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| SALESICECREAM |     8 |   136 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

注8　PARTITION BY句はGROUP BY句と似ていると思った人もいるかもしれません。実際両者はよく似ていますが、GROUP BY句が行の集約まで行うのに対して、PARTITION句はグループ化は行いますが集約は行いません。そのため、結果のレコードの行数が減らないのです。


うわあシンプル……テーブルアクセスが1回に減って、ウィンドウ関数のソートだけになりましたね。


そうだ。これが**SQL最強の武器**ウィンドウ関数だ。よく覚えておけ。SQL:2003からフルレベルで標準に入ったから、今ではどのDBMSでも使える優れモノだ。

ウィンドウとは何か


ところでウィンドウって何なんでしょう。別にこのクエリにウィンドウって出てこないと思うのですけど。


ふむ、素朴だが重要な疑問だ。実際のところ、ウィンドウってのはいったい何だろうな？ その答えは、次のクエリを見るとわかるだろう。先ほどのアイスクリーム屋のクエリは次のようにも書くことができる(**リスト0-23**)。

リスト0-23　**正式なウィンドウ関数の書き方(結果は先ほどと同じ)**

```
SELECT shop_id, sale_date, sales_amt,
       SUM(sales_amt) OVER CUMLATIVE AS cumlative_amt
  FROM SalesIcecream
  WINDOW CUMLATIVE AS (PARTITION BY shop_id
                           ORDER BY sale_date);
```


あれ、WINDOWというキーワードが登場している。ということは、ここがウィンドウの定義ということですか。


そうだ。本来のウィンドウ関数の書き方はこっちのほうが正式なのだ。我々がよく使うウィンドウ関数は、**無名ウィンドウ**を使った簡略版だ。しかし簡略版のほうが普及してしまったため、本来の書き方のほうが珍しくなってしまった。この書き方なら一目瞭然だが、ウィンドウというのはPARTITION BY句で区切られたりORDER BY句で順序付けられたりするテーブルのサブセットからなるレコード集合のことだ。SELECT句で何度も同じウィンドウにアクセスするような場合は、有名ウィンドウで定義を一ヵ所にまとめるほう

がすっきり書ける。そうでないケースでは無名ウィンドウを使うことのほうが多い。


うーんなるほど。定義はよくわかりました。ところでウィンドウってどういう意味なんですか？「窓」って意味ではないですよね。


ウィンドウは窓という意味が一般的だけど、この場合のウィンドウは「時系列に順序付けられたデータ」という意味ね。もともとウィンドウには、「バッチウィンドウ」とか「メンテナンスウィンドウ」みたいに特定の時間の「範囲」を意味する使い方があるのだけど、それに近い使われ方をしているわ。実際、ウィンドウ関数も時系列データの分析に使うことがとても多いわ。


さて、ウィンドウ関数の使い方をもう１問練習しておくか。次の症例を治療してみろ。今度は少し難しいぞ。お前の実力を見てやる。

フレーム句の使い方

カルテ9 先ほどのアイスクリームの売り上げテーブルから、現在の行から２行前までを計算範囲とする移動平均を取得したい。答えは小数点第１位で四捨五入して整数で求める（**図0-25**）。

図0-25　アイスクリーム売り上げテーブル（再掲）

SalesIcecream

shop_id（店舗 ID）	sale_date（売上日）	sales_amt（売上金額）
A	2024-06-01	67,800
A	2024-06-02	87,000
A	2024-06-05	11,300
A	2024-06-10	9,800
A	2024-06-15	9,800
B	2024-06-02	178,000
B	2024-06-15	18,800
B	2024-06-17	19,850
B	2024-06-20	23,800
B	2024-06-21	18,800
C	2024-06-01	12,500


移動平均、僕知ってます。株やってるんで。平均をとる期間がどんどんずれていくんですよね。テクニカル分析では移動平均線は必須の指標ですよ。


青二才のくせにいっちょまえに資産形成か。まあ知識があるのはいいことだ。そうだ。現在行を起点として日付を過去に2行分さかのぼって平均を計算する。


さっきの累計は特に期間を限定せずに集計したけど、今度は行集合のサブセットをどう定義するかがポイントね。


すいません、全然わかんないですけど。


堂々と白旗挙げるな！ もうちょっと悩むふりぐらいしろ（こいつハズレかもな……）。


ふう……次のクエリを見て（この子ハズレかも……）（**リスト0-24**、**図0-26**）。

リスト0-24　**ウィンドウ関数で移動平均を求める**

```
SELECT shop_id, sale_date, sales_amt,
       ROUND(AVG(sales_amt) OVER
         (PARTITION BY shop_id
              ORDER BY sale_date
              ROWS BETWEEN 2 PRECEDING AND CURRENT ROW),0) AS moving_avg
  FROM SalesIcecream;
```

図0-26　**実行結果**

```
 shop_id | sale_date  | sales_amt | moving_avg
---------+------------+-----------+-----------
 A       | 2024-06-01 |     67800 |      67800    ← (67800) / 1
 A       | 2024-06-02 |     87000 |      77400    ← (67800 + 87000) / 2
 A       | 2024-06-05 |     11300 |      55367    ← (67800 + 87000 + 11300) / 3
 A       | 2024-06-10 |      9800 |      36033    ← (87000 + 11300 + 9800) / 3
 A       | 2024-06-15 |      9800 |      10300    ← (11300 + 9800 + 9800) / 3
-------------------------------------------------- PARTITION BY句によるリセット
 B       | 2024-06-02 |    178000 |     178000    ← (178000) / 1
 B       | 2024-06-15 |     18800 |      98400    ← (178000 + 18800) / 2
 B       | 2024-06-17 |     19850 |      72217    ← (178000 + 18800 + 19850) / 3
 B       | 2024-06-20 |     23800 |      20817    ← (18800 + 19850 + 23800) / 3
 B       | 2024-06-21 |     18800 |      20817    ← (19850 + 23800 + 18800) / 3
-------------------------------------------------- PARTITION BY句によるリセット
 C       | 2024-06-01 |     12500 |      12500    ← (12500) / 1
```

平均だからAVG関数を使うのはわかるんですけど、ROWS BETWEEN 2 PRECEDING AND CURRENT ROWって何ですか。

そこがこのクエリの肝よ。**フレーム句**といって、ウィンドウの中でサブセットを定義するための機能よ。読んで字のごとく、「2行前から現在行まで」の3行を指定しているわ。

移動平均は通常過去にレコードをさかのぼるが、もし未来にレコードを進めたいならPRECEDINGの反対でFOLLOWINGというキーワードを使う。フレーム句が指定されていない場合はデフォルトでROWS UNBOUNDED PRECEDINGとみなされる。制限なしにレコードを前にさかのぼるということだな。

なるほど、累計のときの動作ですね。計算対象の行集合が3行に満たない場合は自動的に除数を調節して平均の計算してくれるのが気が利いてますね。


少しコードを変えれば、そういう行数が足りないケースはNULLで出力することもできるわ。ちょうどいいから今日の宿題にしましょう。


ウゲッ。


今はROWSを使ったが、データの値を条件にしたければ代わりにRANGEというキーワードを使うこともできる。実に行き届いた機能じゃないか。


よし、次が最後のテストだ。この問題が解けるようなら救命室に置いてやる。


うー緊張するなあ。

カルテ10 **リスト0-25**のような学生の体重を管理するテーブルがある。このクラスの学生の中で平均体重よりも重い学生を選択したい。平均体重の小数点以下は四捨五入とする。

リスト0-25　**体重テーブル**

```
CREATE TABLE Weights
(student_id CHAR(4) NOT NULL PRIMARY KEY,
 weight INTEGER NOT NULL);
```

Weights:体重テーブル　student_id:学生ID　weight:体重

Weights

student_id (学生 ID)	weight (体重)
A	55
B	70
C	65
D	120
E	83
F	63


まず、クラスの平均を求めないと始まらないから、次のクエリで求めます(**リスト0-26**、**図0-27**)。

リスト0-26　**平均を求めるクエリ**

```
SELECT ROUND(AVG(weight), 0) AS avg_weight
  FROM Weights;
```

図0-27　**実行結果**

```
 avg_weight
------------
         76
```


平均体重が76ということは、求める結果は学生ＥとＤですね。


そうね。あとはこれを各学生の体重とどう比較するかね。


素直にやるとこうなると思うんだけど……(**リスト0-27**、**図0-28**)。

リスト0-27　**クラスの平均と学生の体重を比較する**

```
SELECT *
  FROM Weights
 WHERE weight > (SELECT ROUND(AVG(weight), 0) AS avg_weight
                   FROM Weights);
```

図0-28　実行結果

```
 student_id | weight
------------+--------
 D          |    120
 E          |     83
```


ふむ。まあ答えが合っているから半分点数をやろう。50点だ。


正しい答えはどうなるんでしょう。これ以上単純な解は思いつかないんですけど。


まずこのクエリの実行計画を取ってみろ。


はい、explainとautotraceで……取れました(図0-29、図0-30)。

図0-29　ワイリーの解の実行計画(PostgreSQL)

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on weights  (cost=29.64..63.19 rows=523 width=24)
   Filter: ((weight)::numeric > $0)
   InitPlan 1 (returns $0)
     ->  Aggregate  (cost=29.63..29.64 rows=1 width=32)
           ->  Seq Scan on weights weights_1  (cost=0.00..25.70 rows=1570 width=4)
```

図0-30　ワイリーの解の実行計画(Oracle)

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |         |     1 |    19 |     4   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL  | WEIGHTS |     1 |    19 |     2   (0)| 00:00:01 |
|   2 |   SORT AGGREGATE    |         |     1 |    13 |            |          |
|   3 |    TABLE ACCESS FULL| WEIGHTS |     6 |    78 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```


実行計画を見て、どこがダメかわかる？


やっぱりテーブルアクセスが2回発生しているところですか？


そう。SQLのパフォーマンスは一にI/O、二にI/O、三四がなくて五にI/Oよ。とにかくストレージへのI/Oを減らすことに全力を傾

けるの。**もうこれ以上削れないってくらいに削るの。**


正しい解はこうなる(**リスト0-28、図0-31**)。

リスト0-28　**正しい解：ウィンドウ関数を使う**

```
SELECT student_id, weight, avg_weight
  FROM (SELECT student_id, weight,
               ROUND(AVG(weight) OVER(), 0) AS avg_weight
          FROM Weights) TMP
  WHERE weight > avg_weight;
```

図0-31　**実行結果**

```
 student_id | weight | avg_weight
------------+--------+------------
 D          |    120 |         76
 E          |     83 |         76
```


OVER句の中に何も書かないって許されるんだ！


PARTITION BY句もORDER BY句もオプションだから、**書かなくても構文上問題ないわ。**今回は特にパーティションを区切る必要はないし、平均を出すのに行の順序も関係ないから、OVER句は空っぽで大丈夫よ。


はー、これでもれっきとしたウィンドウになるんだ。ちょっとびっくり。


実行計画をお前のクエリと比較してみろ。


はい、ただいま(**図0-32、図0-33**)。

図0-32　**ヘレンの解の実行計画(PostgreSQL)**

```
                            QUERY PLAN
------------------------------------------------------------------------
 Subquery Scan on tmp  (cost=0.00..72.80 rows=523 width=56)
   Filter: ((tmp.weight)::numeric > tmp.avg_weight)
   ->  WindowAgg  (cost=0.00..49.25 rows=1570 width=56)
         ->  Seq Scan on weights  (cost=0.00..25.70 rows=1570 width=24)
```

図0-33　ヘレンの解の実行計画(Oracle)

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |         |     6 |   192 |     2   (0)| 00:00:01 |
|*  1 |  VIEW               |         |     6 |   192 |     2   (0)| 00:00:01 |
|   2 |   WINDOW BUFFER     |         |     6 |   114 |     2   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| WEIGHTS |     6 |   114 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

テーブルアクセスが１回に減っていますね。すごい、こんなに実行計画が単純化されるんだ。

ウィンドウ関数はその強力無比の万能さから**魔法の関数**と呼ばれることもあるのだけど、その理由がわかるでしょ[注9]。使いこなすともっとすごいことができるんだから。これからたくさん症例を見ることになるけど、必ず使うからよく覚えておいてね。

あと、ウィンドウ関数を使って地味にうれしいのが平均体重(avg_weight)も結果に出力できることだな。これもウィンドウ関数がレコードの集約を行わずヒラで結果を得るから可能なことだ。

うーんすごい関数だ。SQLって僕が思っていた以上に奥が深いんだな……。

(15:00、休憩室。３人がコーヒーを飲んでいる。)

ウィンドウ関数、本当に行き届いた機能だなあ。CASE式とウィンドウ関数があればほかのプログラミング言語にもひけをとらないコーディングができそうな気がしてきました。なんかちょっと僕の習ってきたSQLとは別物のクエリを見ましたよ。感動です。

うむ、この２つはSQLの誇る双璧だからな。それはこの２つの機能が手続き型言語における条件分岐とループに相当するからだ。条件

注9　ウィンドウ関数がサブクエリを消去する効果があることは、早い段階から知られていました。下記論文を参照。
Calisto Zuzarte, Hamid Pirahesh, Wenbin Ma, Qi Cheng, Linqi Liu, Kwai Wong, "WinMagic: subquery elimination using window aggregation", 2003
https://dl.acm.org/doi/10.1145/872757.872840

分岐とループは手続き型言語にとってもかなめだろう？ これからも毎回オペで使うから、この救命室で働くならよく覚えておけ。でないと救える患者も救えないぞ。


イエッサー！


（返事はいいんだけどな……まあとりあえずとってみて様子を見るとするか）


よし、お前を救命室で採用してやる。朝は8時に来て9時から診療を開始できるようカルテを準備しておけ。夜勤もあるから覚悟しろ。

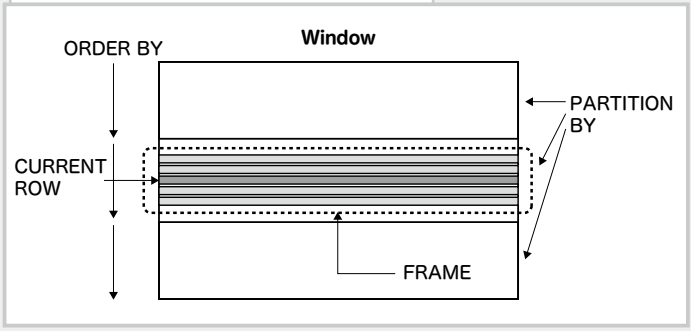

はい！ よろしくお願いします。

まとめ

- CASE式とウィンドウ関数がモダンSQLの二大技術。これを使いこなせるようになることが本書の主目的の一つ
- CASE式は式であるがゆえにどこにでも書け、柔軟に条件分岐を指定できる。そのため一種のアドホックなユーザー定義関数のように機能する
- ウィンドウ関数はSQLでループ相当の機能を実現する強力な技術
- ウィンドウ関数はPARTITION BY句で区切られた行集合に対してORDER BY句で順序付けを行いさまざまな集計を行うことができる。どちらの句もオプションなので書かなくてもエラーにはならない
- ウィンドウの行集合の中でさらにサブセットを定義したい場合はフレーム句を利用する。ウィンドウ関数を一枚で図示すると**図0-34**のようなイメージになる[注a]

注a この図は下記論文に少し著者が変更を加えて引用。
Viktor Leis, Alfons Kemper, Kan Kundhikanjana, Thomas Neumann, "Efficient Processing of Window Functions in Analytical SQL Queries"
https://www.vldb.org/pvldb/vol8/p1058-leis.pdf

図0-34　ウィンドウ関数を一枚の図で表現する



演習問題

解答374ページ

演習0-1

City テーブル（リスト0-7）のNew YorkとLos Angelsの人口を入れ替える手段としては、都市名を入れ替えるという方法もあります。これを実現するUPDATE文を考えてください。

ヒント：一つのUPDATE文で一気に処理してください。

演習0-2

カルテ9で求めた移動平均について、集計対象行が3行に満たない場合は無効としてNULLを出力するようにクエリを改変してください。

ヒント：CASE式とウィンドウ関数の合わせ技です。