==========

18. サブクエリ：WHERE句内のサブクエリ

はじめに

これまでの章では、テーブルの結合（JOIN）を使って複数のテーブルからデータを取得する方法を学びました。しかし、データの取得や絞り込みには、もう一つの強力な方法があります。それが「サブクエリ（副問合せ）」です。

サブクエリとは、SQLクエリの中に埋め込まれた別のSQLクエリのことです。たとえば次のような場合に便利です：

- 「平均点より高い点数の成績だけを取得したい」
- 「特定の講座を受講している学生だけを検索したい」
- 「出席率が80%以上の学生の情報を知りたい」

この章では、サブクエリの基本概念を理解し、特にWHERE句内でのサブクエリの使い方について学びます。

サブクエリとは

サブクエリ（または副問合せ）とは、別のSQL文の中に含まれるSQL文のことです。「クエリの中のクエリ」と考えることができます。

> **用語解説**：
>
> - **サブクエリ（副問合せ）**：SQLクエリの中に埋め込まれた別のSQLクエリで、外側のクエリ（外部クエリ）に値や条件を提供します。
> - **外部クエリ**：サブクエリを含む、より大きなクエリのことです。

サブクエリの特徴

サブクエリには、以下のような特徴があります：

1. **括弧で囲む**：サブクエリは常に括弧`()`で囲む必要があります。
2. **内側から実行**：通常、サブクエリは外部クエリより先に実行されます。
3. **返す値**：サブクエリが返す値によって、いくつかのタイプに分けられます：
    - **スカラーサブクエリ**：単一の値（1行1列）を返す
    - **行サブクエリ**：単一の行（複数列可）を返す
    - **表サブクエリ**：複数の行と列を返す（結果セットのようなもの）
4. **使用場所**：SQL文の様々な場所で使用できます：
    - WHERE句内（この章のトピック）
    - FROM句内（テーブルのように扱う）
    - SELECT句内（計算値として）
    - HAVING句内（集計後のフィルタリング）
    - JOIN句の条件として

サブクエリとJOINの違い

サブクエリとJOINは両方とも複数のテーブルからデータを関連付ける方法ですが、アプローチが異なります：

サブクエリ	JOIN
クエリの中に別のクエリを埋め込む	複数のテーブルを直接連結する
結果を段階的に絞り込む	結果を一度に結合して表示する
複雑なフィルタリングや計算に適している	複数テーブルからの情報を一覧表示するのに適している
読みやすいクエリになることがある	簡潔なクエリになることがある
パフォーマンスが良い場合と悪い場合がある	通常は効率的だが、大きなテーブル結合では重くなることがある

どちらを使うかは、解決したい問題や求める結果の形式によって異なります。多くの場合、同じ結果を得るためにサブクエリとJOINの両方の方法が考えられます。

WHERE句内のサブクエリ

WHERE句内でのサブクエリは、条件の一部として別のクエリの結果を使用する方法です。これは特に、動的に条件を決定したい場合に便利です。

基本構文：比較演算子とサブクエリ

```
SELECT カラム名
FROM テーブル名
WHERE カラム名 比較演算子 (SELECT カラム名 FROM テーブル名 WHERE 条件);
```

比較演算子には、`=`, `<>`, `>`, `<`, `>=`, `<=`などが使えます。

例1：スカラーサブクエリ（単一値）

平均点よりも高い成績を取得するクエリを考えてみましょう：

```
SELECT student_id, course_id, grade_type, score
FROM grades
WHERE score > (SELECT AVG(score) FROM grades)
ORDER BY score DESC;
```

このクエリでは：

1. サブクエリ `(SELECT AVG(score) FROM grades)` がまず実行され、すべての成績の平均点を計算します。
2. 外部クエリでは、その平均点より高いスコアを持つレコードだけを取得します。

実行結果：

student_id	course_id	grade_type	score
311	1	中間テスト	95.0
320	1	中間テスト	93.5
302	1	中間テスト	92.0
308	1	中間テスト	91.0
…	…	…	…

上記のクエリでサブクエリが返す値は単一の値（例えば78.5など）です。このように単一の値を返すサブクエリをスカラーサブクエリと呼びます。

例2：特定教師の担当科目

教師ID=101（寺内鞍）が担当する講座を取得するクエリ：

```
SELECT course_id, course_name
FROM courses
WHERE teacher_id = (SELECT teacher_id FROM teachers WHERE teacher_name = '寺内
鞍');
```

このクエリでは：

1. サブクエリ `(SELECT teacher_id FROM teachers WHERE teacher_name = '寺内鞍')` が寺内鞍先生のIDを取得します（結果は101）。
2. 外部クエリでは、そのIDに一致する講座を取得します。

実行結果：

course_id	course_name
1	ITのための基礎知識
3	Cプログラミング演習
29	コードリファクタリングとクリーンコード

例3：複数値を返すサブクエリ（IN演算子）

特定の講座を受講している学生を取得するクエリを考えてみましょう：

```
SELECT student_id, student_name
FROM students
WHERE student_id IN (
    SELECT student_id
    FROM student_courses
    WHERE course_id = '1'
```

```
)
ORDER BY student_id;
```

このクエリでは：

1. サブクエリは講座ID=1を受講している学生IDのリストを返します。
2. 外部クエリでは、そのリストに含まれる学生IDを持つ学生レコードだけを取得します。

実行結果：

student_id	student_name
301	黒沢春馬
302	新垣愛留
303	柴崎春花
306	河田咲奈
…	…

ここでは、サブクエリが複数の値（学生IDのリスト）を返し、外部クエリではIN演算子を使って「そのリストのいずれかに一致する」という条件を表現しています。

IN、NOT IN、EXISTS、NOT EXISTSとの組み合わせ

サブクエリの結果が複数の行を返す場合、以下の演算子と組み合わせて使用します：

1. IN / NOT IN

「～のいずれかに一致する」または「～のいずれにも一致しない」という条件を表現します。

> **用語解説**：
>
> - **IN**：値がリストの中のいずれかの値と一致するか確認します。
> - **NOT IN**：値がリストの中のどの値とも一致しないか確認します。

例4：IN演算子を使ったサブクエリ

2025年5月20日に授業が予定されている講座を検索：

```
SELECT course_id, course_name
FROM courses
WHERE course_id IN (
    SELECT DISTINCT course_id
    FROM course_schedule
    WHERE schedule_date = '2025-05-20'
)
ORDER BY course_id;
```

例5：NOT IN演算子を使ったサブクエリ

2025年5月に授業が予定されていない講座を検索：

```
SELECT course_id, course_name
FROM courses
WHERE course_id NOT IN (
    SELECT DISTINCT course_id
    FROM course_schedule
    WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31'
)
ORDER BY course_id;
```

2. EXISTS / NOT EXISTS

レコードの存在チェックを行います。結果の値そのものは重要ではなく、「存在するかどうか」だけが条件になります。

> **用語解説**：
>
> - **EXISTS**：サブクエリが少なくとも1行の結果を返すかどうかをチェックします。
> - **NOT EXISTS**：サブクエリが結果を1行も返さないかどうかをチェックします。

例6：EXISTS演算子を使ったサブクエリ

出席記録がある学生を検索：

```
SELECT student_id, student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM attendance a
    WHERE a.student_id = s.student_id
)
ORDER BY student_id;
```

このクエリでは、各学生に対して出席記録があるかどうかをチェックしています。サブクエリの`SELECT 1`は、結果の値は重要ではなく、単にレコードが存在するかどうかだけを調べるためのものです。

例7：NOT EXISTS演算子を使ったサブクエリ

まだ一度も講座を受講していない学生を検索：

```
SELECT student_id, student_name
FROM students s
WHERE NOT EXISTS (
    SELECT 1
```

```
    FROM student_courses sc
    WHERE sc.student_id = s.student_id
)
ORDER BY student_id;
```

ALL、ANY、SOMEとの組み合わせ

サブクエリが複数の値を返す場合、以下の修飾子と比較演算子を組み合わせて使うこともできます：

1. ALL

「すべての値と比較して条件を満たす」という意味です。

> **用語解説**：
>
> - **ALL**：サブクエリの結果のすべての値と比較して条件を満たすかどうかをチェックします。

例8：ALL修飾子を使ったサブクエリ

すべての成績の平均よりも高い点数を取った学生を検索：

```
SELECT DISTINCT s.student_id, s.student_name
FROM students s
JOIN grades g ON s.student_id = g.student_id
WHERE g.score > ALL (
    SELECT AVG(score)
    FROM grades
    GROUP BY course_id
)
ORDER BY s.student_id;
```

2. ANY / SOME

「いずれかの値と比較して条件を満たす」という意味です（ANY と SOME は同じ意味）。

> **用語解説**：
>
> - **ANY/SOME**：サブクエリの結果のいずれかの値と比較して条件を満たすかどうかをチェックします。

例9：ANY修飾子を使ったサブクエリ

少なくとも1つの授業で85点以上を取得している学生を検索：

```
SELECT DISTINCT s.student_id, s.student_name
FROM students s
WHERE s.student_id = ANY (
    SELECT student_id
    FROM grades
```

```
    WHERE score >= 85
)
ORDER BY s.student_id;
```

相関サブクエリの基本

通常のサブクエリは、外部クエリとは独立して実行されます。一方、相関サブクエリは外部クエリの現在処理中の行を参照します。これにより、各行ごとに異なる条件での評価が可能になります。

> **用語解説**：
>
> - **相関サブクエリ**：外部クエリの現在の行を参照するサブクエリで、外部クエリと「相関関係」にあるサブクエリです。

例10：相関サブクエリの基本例

各学生の平均点より高い成績だけを取得するクエリ：

```
SELECT g.student_id, g.course_id, g.grade_type, g.score
FROM grades g
WHERE g.score > (
    SELECT AVG(score)
    FROM grades
    WHERE student_id = g.student_id
)
ORDER BY g.student_id, g.score DESC;
```

このクエリでは：

1. 外部クエリでgrades表から1行ずつ処理します。
2. サブクエリでは、現在処理中の学生IDの平均点を計算します（`WHERE student_id = g.student_id`の部分が相関しています）。
3. その学生の平均点より高い成績だけを結果に含めます。

外部クエリの各行に対して、サブクエリが実行されるため、処理は次のようになります：

- 学生301の場合：学生301の平均点を計算し、それより高い学生301の成績を返す
- 学生302の場合：学生302の平均点を計算し、それより高い学生302の成績を返す
- ...以下同様

サブクエリのパフォーマンスと注意点

サブクエリを使用する際の主な注意点は以下の通りです：

1. **パフォーマンス**：複雑なサブクエリや相関サブクエリは、実行に時間がかかることがあります。特に大きなテーブルで相関サブクエリを使う場合は注意が必要です。

2. **NULL値の扱い**：NOT IN演算子とサブクエリを組み合わせる場合、サブクエリの結果にNULL値が含まれると予期しない結果になることがあります。NULL値の処理には注意しましょう。

3. **代替手段の検討**：多くの場合、サブクエリはJOINや他の方法でも同じ結果を得られます。パフォーマンスを考慮して、最適な方法を選択しましょう。

4. **可読性**：サブクエリはSQLを理解しやすくする場合もありますが、過度に複雑なネストされたサブクエリは可読性を低下させます。

練習問題

問題18-1

成績（grades）テーブルを使って、平均点より高い点数を取った成績レコードを取得するSQLを書いてください。結果には学生ID、講座ID、評価タイプ、点数を含め、点数の高い順にソートしてください。

問題18-2

学生（students）テーブルと受講（student_courses）テーブルを使って、「クラウドコンピューティング」（course_id = 9）の講座を受講している学生の名前を取得するSQLを書いてください。サブクエリを使用してください。

問題18-3

教師（teachers）テーブル、講座（courses）テーブルを使って、担当講座が3つ以上ある教師の名前を取得するSQLを書いてください。サブクエリとIN演算子を使用してください。

問題18-4

講座（courses）テーブル、学生コース（student_courses）テーブルを使って、受講者が一人もいない講座を取得するSQLを書いてください。NOT EXISTS演算子を使用してください。

問題18-5

学生（students）テーブル、成績（grades）テーブルを使って、全科目の平均点が85点以上の学生を取得するSQLを書いてください。相関サブクエリを使用してください。

問題18-6

教師（teachers）テーブル、講座（courses）テーブル、授業カレンダー（course_schedule）テーブルを使って、2025年5月に授業を行っていない教師を取得するSQLを書いてください。サブクエリとNOT IN演算子を使用してください。

解答

解答18-1

```
SELECT student_id, course_id, grade_type, score
FROM grades
WHERE score > (SELECT AVG(score) FROM grades)
ORDER BY score DESC;
```

解答18-2

```
SELECT student_id, student_name
FROM students
WHERE student_id IN (
    SELECT student_id
    FROM student_courses
    WHERE course_id = '9'
)
ORDER BY student_id;
```

解答18-3

```
SELECT teacher_id, teacher_name
FROM teachers
WHERE teacher_id IN (
    SELECT teacher_id
    FROM courses
    GROUP BY teacher_id
    HAVING COUNT(course_id) >= 3
)
ORDER BY teacher_id;
```

解答18-4

```
SELECT course_id, course_name
FROM courses c
WHERE NOT EXISTS (
    SELECT 1
    FROM student_courses sc
    WHERE sc.course_id = c.course_id
)
ORDER BY course_id;
```

解答18-5

```
SELECT s.student_id, s.student_name
FROM students s
WHERE 85 <= (
    SELECT AVG(score)
    FROM grades g
    WHERE g.student_id = s.student_id
)
ORDER BY s.student_id;
```

解答18-6

```sql
SELECT t.teacher_id, t.teacher_name
FROM teachers t
WHERE t.teacher_id NOT IN (
    SELECT DISTINCT cs.teacher_id
    FROM course_schedule cs
    WHERE cs.schedule_date BETWEEN '2025-05-01' AND '2025-05-31'
)
ORDER BY t.teacher_id;
```

まとめ

この章では、サブクエリの基本概念とWHERE句内でのサブクエリの使い方について学びました：

1. **サブクエリの基本概念**：
    - クエリ内に埋め込まれた別のクエリ
    - 括弧で囲む
    - 通常は内側から実行される
2. **サブクエリの種類**：
    - スカラーサブクエリ（単一値）
    - 複数値を返すサブクエリ
    - 相関サブクエリ（外部クエリを参照）
3. **WHERE句での使用方法**：
    - 比較演算子（=, <>, >, <, >=, <=）との組み合わせ
    - IN / NOT INでの複数値との比較
    - EXISTS / NOT EXISTSでの存在チェック
    - ALL / ANY / SOMEでの条件修飾
4. **サブクエリとJOINの使い分け**：
    - 用途に応じた適切な方法の選択
    - パフォーマンスと可読性の考慮

サブクエリは、動的な条件や複雑な絞り込みを実現するための強力なツールです。次の章では、FROM句内でのサブクエリ（導出テーブル）について学び、さらに高度なクエリ技術を習得していきます。

---

19. サブクエリ応用：SELECT句、FROM句でのサブクエリ

はじめに

前章では、サブクエリの基本概念とWHERE句内でのサブクエリの使い方について学びました。サブクエリは、WHERE句だけでなく、SQL文の様々な部分で使用することができます。特に重要なのは、SELECT句とFROM句でのサブクエリです。

以下のような場合に役立ちます：

- 「各学生の点数と全体の平均点を同時に表示したい」（SELECT句）
- 「複雑な集計結果を元に、さらに計算や絞り込みを行いたい」（FROM句）
- 「テーブル自体を動的に生成して使用したい」（FROM句）

この章では、SELECT句とFROM句におけるサブクエリの活用方法とその応用について学びます。

SELECT句内のサブクエリ

SELECT句内のサブクエリは、クエリの結果セットに計算列を追加する方法の一つです。通常、スカラーサブクエリ（単一の値を返すサブクエリ）が使用されます。

> **用語解説**：
>
> - **スカラーサブクエリ**：単一の値（1行1列）のみを返すサブクエリのことです。
> - **計算列**：既存のデータから計算されて生成される列のことです。

基本構文

```
SELECT
    カラム1,
    カラム2,
    (SELECT 集計関数 FROM テーブル名 WHERE 条件) AS 別名
FROM テーブル名
WHERE 条件;
```

例1：全体の平均点を各成績と一緒に表示

```
SELECT
    student_id,
    course_id,
    grade_type,
    score,
    (SELECT AVG(score) FROM grades) AS 全体平均
FROM grades
WHERE grade_type = '中間テスト'
ORDER BY score DESC;
```

このクエリでは、各成績レコードに「全体平均」という列を追加しています。サブクエリ `(SELECT AVG(score) FROM grades)` は全成績の平均値を計算します。

実行結果：

student_id	course_id	grade_type	score	全体平均
311	1	中間テスト	95.0	78.5
320	1	中間テスト	93.5	78.5
302	1	中間テスト	92.0	78.5
...	...	...	...	...

例2：学生ごとの平均点を表示（相関サブクエリ）

```
SELECT
    s.student_id,
    s.student_name,
    (SELECT AVG(score) FROM grades g WHERE g.student_id = s.student_id) AS 平均点
FROM students s
WHERE s.student_id BETWEEN 301 AND 310
ORDER BY 平均点 DESC;
```

このクエリでは、相関サブクエリを使用して各学生の平均点を計算しています。サブクエリは外部クエリの現在の行（学生）に依存しています。

実行結果：

student_id	student_name	平均点
311	鈴木健太	89.8
302	新垣愛留	86.5
308	永田悦子	85.9
301	黒沢春馬	82.3
...	...	...

例3：受講している講座数を表示

```
SELECT
    s.student_id,
    s.student_name,
    (SELECT COUNT(*) FROM student_courses sc WHERE sc.student_id = s.student_id)
AS 受講講座数
FROM students s
WHERE s.student_id BETWEEN 301 AND 310
ORDER BY 受講講座数 DESC, s.student_id;
```

このクエリでは、各学生が受講している講座の数を相関サブクエリを使って計算しています。

実行結果：

student_id	student_name	受講講座数
301	黒沢春馬	8
309	相沢吉夫	7
306	河田咲奈	6
310	吉川伽羅	6
...	...	...

FROM句内のサブクエリ（導出テーブル）

FROM句内のサブクエリは、クエリの実行中に一時的に生成されるテーブル（導出テーブルやインラインビューとも呼ばれる）として機能します。これにより、複雑な集計や絞り込みを段階的に行うことができます。

> **用語解説**：
>
> - **導出テーブル（Derived Table）**：FROM句内のサブクエリによって生成される一時的なテーブルのことです。
> - **インラインビュー（Inline View）**：FROM句内のサブクエリの別名で、特にOracle製品でよく使われる用語です。

基本構文

```
SELECT カラム1, カラム2, ...
FROM (SELECT カラム1, カラム2, ... FROM テーブル名 WHERE 条件) AS 別名
WHERE 条件;
```

例4：各講座の平均点と全体平均との差を計算

```
SELECT
    avg_scores.course_id,
    c.course_name,
    avg_scores.平均点,
    avg_scores.平均点 - (SELECT AVG(score) FROM grades) AS 全体平均との差
FROM (
    SELECT course_id, AVG(score) AS 平均点
    FROM grades
    GROUP BY course_id
) AS avg_scores
JOIN courses c ON avg_scores.course_id = c.course_id
ORDER BY avg_scores.平均点 DESC;
```

このクエリでは：

1. サブクエリ（導出テーブル）で各講座の平均点を計算
2. 外部クエリでは、その平均点と全体平均との差を計算
3. 結果を平均点の高い順に並べる

実行結果：

course_id	course_name	平均点	全体平均との差
1	ITのための基礎知識	86.21	7.71
2	UNIX入門	83.79	5.29
5	データベース設計と実装	82.33	3.83
…	…	…	…

例5：成績上位の学生を抽出

```
SELECT
    top_students.student_id,
    s.student_name,
    top_students.平均点
FROM (
    SELECT student_id, AVG(score) AS 平均点
    FROM grades
    GROUP BY student_id
    HAVING AVG(score) > 85
) AS top_students
JOIN students s ON top_students.student_id = s.student_id
ORDER BY top_students.平均点 DESC;
```

このクエリでは：

1. サブクエリで平均点が85点を超える学生を抽出
2. 外部クエリでその学生の名前を取得
3. 平均点の高い順に並べる

実行結果：

student_id	student_name	平均点
311	鈴木健太	89.8
302	新垣愛留	86.5
308	永田悦子	85.9
…	…	…

例6：複数の集計結果を組み合わせる

```
SELECT
    attendance_stats.student_id,
    s.student_name,
    attendance_stats.出席回数,
    attendance_stats.欠席回数,
    attendance_stats.遅刻回数,
    attendance_stats.総授業数,
    ROUND(attendance_stats.出席回数 * 100.0 / attendance_stats.総授業数, 1) AS 出席
率
FROM (
    SELECT
        student_id,
        SUM(CASE WHEN status = 'present' THEN 1 ELSE 0 END) AS 出席回数,
        SUM(CASE WHEN status = 'absent' THEN 1 ELSE 0 END) AS 欠席回数,
        SUM(CASE WHEN status = 'late' THEN 1 ELSE 0 END) AS 遅刻回数,
        COUNT(*) AS 総授業数
    FROM attendance
    GROUP BY student_id
) AS attendance_stats
JOIN students s ON attendance_stats.student_id = s.student_id
ORDER BY 出席率 DESC;
```

このクエリでは：

1. サブクエリ内で複雑な集計（出席状況の分類と集計）を行い
2. 外部クエリではその結果を使って出席率を計算しています

このように、複雑な集計を段階的に行うことで、クエリの可読性を高めることができます。

SELECT句とFROM句の両方でサブクエリを使用

より複雑な分析では、SELECT句とFROM句の両方でサブクエリを使用することもあります。

例7：各講座の平均点と全体との比較

```
SELECT
    course_stats.course_id,
    c.course_name,
    course_stats.受講者数,
    course_stats.平均点,
    (SELECT AVG(score) FROM grades) AS 全体平均,
    course_stats.平均点 - (SELECT AVG(score) FROM grades) AS 平均点差,
    CASE
        WHEN course_stats.平均点 > (SELECT AVG(score) FROM grades) THEN '↑'
        WHEN course_stats.平均点 < (SELECT AVG(score) FROM grades) THEN '↓'
        ELSE '→'
    END AS 比較
FROM (
    SELECT
        course_id,
```

```
        COUNT(DISTINCT student_id) AS 受講者数,
        AVG(score) AS 平均点
    FROM grades
    GROUP BY course_id
) AS course_stats
JOIN courses c ON course_stats.course_id = c.course_id
ORDER BY course_stats.平均点 DESC;
```

このクエリでは：

1. FROM句のサブクエリで各講座の受講者数と平均点を計算
2. SELECT句のサブクエリで全体平均を計算
3. CASE式で平均点と全体平均を比較して矢印記号を表示

実行結果：

course_id	course_name	受講者数	平均点	全体平均	平均点差	比較
1	ITのための基礎知識	12	86.21	78.5	7.71	↑
2	UNIX入門	8	83.79	78.5	5.29	↑
5	データベース設計と実装	7	82.33	78.5	3.83	↑
…	…	…	…	…	…	…

サブクエリのネスト（入れ子）

サブクエリはネスト（入れ子）することもできます。つまり、サブクエリの中に別のサブクエリを含めることができます。

例8：平均点が全体の上位25%に入る学生を検索

```
SELECT student_id, student_name
FROM students
WHERE student_id IN (
    SELECT student_id
    FROM (
        SELECT
            student_id,
            AVG(score) AS avg_score,
            PERCENT_RANK() OVER (ORDER BY AVG(score)) AS percentile
        FROM grades
        GROUP BY student_id
    ) AS student_percentiles
    WHERE percentile >= 0.75
)
ORDER BY student_id;
```

このクエリでは：

1. 最も内側のサブクエリで各学生の平均点とパーセンタイルを計算
2. 中間のサブクエリでパーセンタイルが75%以上の学生IDを抽出
3. 外部クエリでその学生の情報を取得

サブクエリとJOINの使い分け

前章でも触れましたが、多くの場合、サブクエリとJOINは互いに代替可能です。一般的な傾向として：

サブクエリが適している場合：

- 段階的に結果を絞り込みたい場合
- 一時的な集計結果を使って更なる計算や絞り込みをしたい場合
- クエリの各部分を明確に分離したい場合
- 相関サブクエリを使って行ごとの計算や比較をしたい場合

JOINが適している場合：

- 複数のテーブルから同時にデータを取得したい場合
- 結果セットで複数のテーブルのカラムを表示したい場合
- 大きなデータセットで処理速度を重視する場合

共通テーブル式（CTE）との比較

MySQL 8.0以降では、FROM句のサブクエリの代わりに「共通テーブル式（CTE）」を使用することもできます。CTE（WITH句）は可読性が高く、同じ導出テーブルを複数回参照する場合に特に便利です。CTEについては後の章で詳しく学びます。

例9：CTEを使った同等のクエリ（参考）

```
WITH course_stats AS (
    SELECT
        course_id,
        COUNT(DISTINCT student_id) AS 受講者数,
        AVG(score) AS 平均点
    FROM grades
    GROUP BY course_id
)
SELECT
    course_stats.course_id,
    c.course_name,
    course_stats.受講者数,
    course_stats.平均点,
    (SELECT AVG(score) FROM grades) AS 全体平均,
    course_stats.平均点 - (SELECT AVG(score) FROM grades) AS 平均点差
FROM course_stats
JOIN courses c ON course_stats.course_id = c.course_id
ORDER BY course_stats.平均点 DESC;
```

パフォーマンスの考慮点

サブクエリを使用する際のパフォーマンスに関する主な考慮点は以下の通りです：

1. **相関サブクエリの影響**：相関サブクエリは外部クエリの各行に対して実行されるため、大きなテーブルでは処理が遅くなる可能性があります。
2. **実行計画の確認**：複雑なサブクエリを使用する場合は、データベースがどのようにクエリを実行するかを確認しましょう（EXPLAIN文などを使用）。
3. **代替手段の検討**：特にパフォーマンスが重要な場合は、JOIN、インデックス、一時テーブル、ビューなどの代替手段も検討しましょう。
4. **再利用可能性**：同じサブクエリを複数回使用する場合は、CTEや一時テーブルを使用することで処理を一度だけにすることができます。

練習問題

問題19-1

SELECT句内のサブクエリを使用して、各学生の名前、学生ID、およびその学生が受講している講座の数を表示するSQLを書いてください。学生ID=301から305までの学生だけを対象とし、結果を受講講座数の多い順にソートしてください。

問題19-2

FROM句内のサブクエリを使用して、成績の平均点が80点以上の講座の情報（講座ID、講座名、平均点）を取得するSQLを書いてください。結果を平均点の高い順にソートしてください。

問題19-3

SELECT句とFROM句の両方でサブクエリを使用して、各学生の出席率（出席回数÷全授業回数×100）とクラス全体の平均出席率を比較するSQLを書いてください。結果には学生ID、学生名、出席率、平均出席率、および出席率と平均出席率の差を含めてください。

問題19-4

FROM句内のサブクエリを使用して、各教師が担当している講座の数と、その教師が担当するすべての講座の平均受講者数を計算するSQLを書いてください。結果を担当講座数の多い順にソートしてください。

問題19-5

FROM句内のサブクエリとJOINを組み合わせて、各学生の中間テストとレポート1の点数を横並びで比較するSQLを書いてください。結果には学生ID、学生名、中間テスト点数、レポート1点数、およびその差（中間テスト - レポート1）を含めてください。

問題19-6

複数のサブクエリをネストして、以下の情報を含むレポートを作成するSQLを書いてください：各講座について、講座名、担当教師名、平均点、最高点を取った学生の名前、そして授業の予定回数を表示します。結果を講座IDの順にソートしてください。

解答

解答19-1

```
SELECT
    student_id,
    student_name,
    (SELECT COUNT(*) FROM student_courses sc WHERE sc.student_id = s.student_id)
AS 受講講座数
FROM students s
WHERE student_id BETWEEN 301 AND 305
ORDER BY 受講講座数 DESC, student_id;
```

解答19-2

```
SELECT
    course_avg.course_id,
    c.course_name,
    course_avg.平均点
FROM (
    SELECT
        course_id,
        AVG(score) AS 平均点
    FROM grades
    GROUP BY course_id
    HAVING AVG(score) >= 80
) AS course_avg
JOIN courses c ON course_avg.course_id = c.course_id
ORDER BY course_avg.平均点 DESC;
```

解答19-3

```
SELECT
    attendance_stats.student_id,
    s.student_name,
    attendance_stats.出席率,
    (SELECT
         AVG(CASE WHEN status = 'present' THEN 100.0 ELSE 0 END)
     FROM attendance) AS 平均出席率,
    attendance_stats.出席率 - (SELECT
                                AVG(CASE WHEN status = 'present' THEN 100.0 ELSE 0
END)
                            FROM attendance) AS 出席率差
FROM (
    SELECT
        student_id,
        AVG(CASE WHEN status = 'present' THEN 100.0 ELSE 0 END) AS 出席率
    FROM attendance
    GROUP BY student_id
) AS attendance_stats
```

```
JOIN students s ON attendance_stats.student_id = s.student_id
ORDER BY 出席率差 DESC;
```

解答19-4

```
SELECT
    teacher_stats.teacher_id,
    t.teacher_name,
    teacher_stats.担当講座数,
    teacher_stats.平均受講者数
FROM (
    SELECT
        c.teacher_id,
        COUNT(c.course_id) AS 担当講座数,
        AVG(student_count.受講者数) AS 平均受講者数
    FROM courses c
    LEFT JOIN (
        SELECT
            course_id,
            COUNT(student_id) AS 受講者数
        FROM student_courses
        GROUP BY course_id
    ) AS student_count ON c.course_id = student_count.course_id
    GROUP BY c.teacher_id
) AS teacher_stats
JOIN teachers t ON teacher_stats.teacher_id = t.teacher_id
ORDER BY teacher_stats.担当講座数 DESC;
```

解答19-5

```
SELECT
    s.student_id,
    s.student_name,
    grades_comp.中間テスト,
    grades_comp.レポート1,
    grades_comp.中間テスト - grades_comp.レポート1 AS 点数差
FROM students s
JOIN (
    SELECT
        student_id,
        MAX(CASE WHEN grade_type = '中間テスト' THEN score ELSE NULL END) AS 中間テス
ト,
        MAX(CASE WHEN grade_type = 'レポート1' THEN score ELSE NULL END) AS レポート1
    FROM grades
    WHERE grade_type IN ('中間テスト', 'レポート1')
    GROUP BY student_id
) AS grades_comp ON s.student_id = grades_comp.student_id
WHERE grades_comp.中間テスト IS NOT NULL AND grades_comp.レポート1 IS NOT NULL
ORDER BY 点数差 DESC;
```

解答19-6

```
SELECT
    c.course_id,
    c.course_name,
    t.teacher_name AS 担当教師,
    course_stats.平均点,
    (SELECT s.student_name
     FROM grades g
     JOIN students s ON g.student_id = s.student_id
     WHERE g.course_id = c.course_id
     ORDER BY g.score DESC
     LIMIT 1) AS 最高得点学生,
    (SELECT COUNT(*)
     FROM course_schedule cs
     WHERE cs.course_id = c.course_id) AS 授業予定回数
FROM courses c
JOIN teachers t ON c.teacher_id = t.teacher_id
LEFT JOIN (
    SELECT
        course_id,
        AVG(score) AS 平均点
    FROM grades
    GROUP BY course_id
) AS course_stats ON c.course_id = course_stats.course_id
ORDER BY c.course_id;
```

まとめ

この章では、SELECT句とFROM句におけるサブクエリの応用的な使い方について学びました：

1. **SELECT句内のサブクエリ**：
    - 計算列を追加するためのスカラーサブクエリ
    - 相関サブクエリを使った行ごとの計算
    - 各行に対する集計値や比較値の表示
2. **FROM句内のサブクエリ（導出テーブル）**：
    - 一時的なテーブルとして機能するサブクエリ
    - 複雑な集計や絞り込みを段階的に行う方法
    - 導出テーブルを使った二次加工
3. **SELECT句とFROM句の両方でのサブクエリ**：
    - 複雑な分析のための組み合わせ
    - 複数の集計レベルでの比較
4. **サブクエリのネスト**：

- サブクエリ内に別のサブクエリを含める方法
- 段階的な絞り込みの実現

5. **サブクエリとJOINの使い分け**：

- それぞれの適した用途
- パフォーマンスへの影響

サブクエリはSQL機能の中でも特に強力なツールの一つであり、複雑な分析や条件付き集計などを実現するための重要なテクニックです。用途に応じてJOINなどの他の手法と使い分けながら、効率的かつ読みやすいクエリを作成することが重要です。

次の章では、「相関サブクエリ：外部クエリと連動するサブクエリ」について詳しく学び、行ごとの比較や条件付き操作をさらに深く理解していきます。

---

20. 相関サブクエリ：外部クエリと連動するサブクエリ

はじめに

前章までで、サブクエリの基本概念とWHERE句・SELECT句・FROM句でのサブクエリの使い方について学びました。サブクエリの中でも特に重要で強力なのが「相関サブクエリ」です。

相関サブクエリとは、外部クエリの値を参照するサブクエリのことです。例えば以下のようなケースで活用できます：

- 「各学生の成績が、その学生の平均点よりも高いものだけを抽出したい」
- 「各講座において、その講座の平均点より高い点数を取った学生を見つけたい」
- 「各教師が担当する講座の中で、最も受講者が多い講座を特定したい」

通常のサブクエリは独立して処理されますが、相関サブクエリは外部クエリの各行に対して実行されるため、より柔軟な条件指定が可能になります。この章では、相関サブクエリの基本概念と具体的な活用方法について詳しく学びます。

相関サブクエリとは

相関サブクエリとは、外部クエリの現在処理中の行の値を参照するサブクエリのことです。サブクエリが外部クエリの列を参照するため、「相関関係(correlation)」があると言われています。

> **用語解説**：
>
> - **相関サブクエリ**：外部クエリの値を参照する（外部クエリに依存する）サブクエリのことです。
> - **外部クエリ**：サブクエリを含む、より大きなクエリのことです。
> - **外部参照**：サブクエリ内から外部クエリのカラムを参照することを指します。

通常のサブクエリと相関サブクエリの違い

1. **通常のサブクエリ**：

- 外部クエリとは独立して実行される
- 外部クエリが実行される前に一度だけ評価される
- 外部クエリの列を参照しない

2. **相関サブクエリ**：
   - 外部クエリの各行に対して実行される
   - 外部クエリの現在の行に依存する
   - 外部クエリの列を参照する

相関サブクエリの基本構文

相関サブクエリの基本構文は以下の通りです：

```
SELECT カラム1, カラム2, ...
FROM テーブル1 外部別名
WHERE カラム 演算子 (
    SELECT 集計関数(カラム)
    FROM テーブル2
    WHERE テーブル2.カラム = 外部別名.カラム
);
```

ここで重要なのは、サブクエリ内のWHERE句が外部クエリのテーブル別名を参照している点です。この参照により、外部クエリの各行に対してサブクエリが実行されます。

相関サブクエリの実践例

例1：学生の平均点より高い成績だけを抽出

```
SELECT g1.student_id, g1.course_id, g1.grade_type, g1.score
FROM grades g1
WHERE g1.score > (
    SELECT AVG(g2.score)
    FROM grades g2
    WHERE g2.student_id = g1.student_id
)
ORDER BY g1.student_id, g1.score DESC;
```

このクエリでは：

1. 外部クエリでgrades表の各行を処理します。
2. サブクエリでは、現在処理中の行の学生ID（`g1.student_id`）を使って、その学生の平均点を計算します。
3. その学生の平均点より高い成績だけを結果に含めます。

実行結果（一部）：

student_id	course_id	grade_type	score
301	2	中間テスト	88.0
301	9	中間テスト	87.5
301	2	レポート1	85.0
302	1	中間テスト	92.0
302	7	中間テスト	91.0
…	…	…	…

例2：講座ごとの平均点より高い点数を取った学生を見つける

```
SELECT c.course_name, s.student_name, g1.score
FROM grades g1
JOIN courses c ON g1.course_id = c.course_id
JOIN students s ON g1.student_id = s.student_id
WHERE g1.grade_type = '中間テスト'
AND g1.score > (
    SELECT AVG(g2.score)
    FROM grades g2
    WHERE g2.course_id = g1.course_id
    AND g2.grade_type = '中間テスト'
)
ORDER BY c.course_name, g1.score DESC;
```

このクエリでは：

1. 外部クエリでは中間テストの成績を処理します。
2. サブクエリでは、現在処理中の行の講座ID（`g1.course_id`）を使って、その講座の中間テストの平均点を計算します。
3. 講座の平均点より高い点数を取った学生だけを結果に含めます。

実行結果：

course_name	student_name	score
AI・機械学習入門	新垣愛留	91.0
AI・機械学習入門	中村彩香	89.5
ITのための基礎知識	鈴木健太	95.0
ITのための基礎知識	松本さくら	93.5
ITのための基礎知識	新垣愛留	92.0
…	…	…

例3：各教師が担当する講座の中で最も受講者が多い講座

```
SELECT t.teacher_id, t.teacher_name, c.course_name,
       (SELECT COUNT(*) FROM student_courses sc WHERE sc.course_id = c.course_id)
AS 受講者数
FROM teachers t
JOIN courses c ON t.teacher_id = c.teacher_id
WHERE (
    SELECT COUNT(*)
    FROM student_courses sc
    WHERE sc.course_id = c.course_id
) = (
    SELECT MAX(enrollment_count)
    FROM (
        SELECT c2.course_id, COUNT(*) AS enrollment_count
        FROM courses c2
        JOIN student_courses sc ON c2.course_id = sc.course_id
        WHERE c2.teacher_id = t.teacher_id
        GROUP BY c2.course_id
    ) AS course_enrollments
)
ORDER BY t.teacher_id;
```

このクエリでは：

1. 外部クエリでteachersとcoursesテーブルを結合します。
2. 最初のサブクエリは、現在の講座の受講者数を計算します。
3. 二番目のサブクエリは、相関サブクエリとネスト（入れ子）を組み合わせて、教師（`t.teacher_id`）が担当する講座の中での最大受講者数を求めます。
4. それが一致する講座だけを結果に含めます。

実行結果：

teacher_id	teacher_name	course_name	受講者数
101	寺内鞍	ITのための基礎知識	12
102	田尻朋美	サーバーサイドプログラミング	9
103	藤本理恵	Webアプリケーション開発	11
…	…	…	…

このように、相関サブクエリを使うことで「各〜について」という条件を実現できます。

EXISTS演算子と相関サブクエリ

相関サブクエリはEXISTS演算子と組み合わせると特に強力です。EXISTSは、サブクエリが少なくとも1行結果を返すかどうかだけをチェックします。

> **用語解説：**
>
> - **EXISTS**：サブクエリが少なくとも1行の結果を返すかどうかをチェックする演算子です。

> - **NOT EXISTS**：サブクエリが結果を1行も返さないかどうかをチェックする演算子です。

例4：EXISTS演算子を使った相関サブクエリ

中間テストとレポート1の両方を提出している学生を検索：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
    AND g.grade_type = '中間テスト'
)
AND EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
    AND g.grade_type = 'レポート1'
)
ORDER BY s.student_id;
```

このクエリでは、各学生について「中間テストがある」「レポート1がある」という二つの条件を相関サブクエリとEXISTSを使ってチェックしています。

例5：NOT EXISTS演算子を使った相関サブクエリ

いずれかの授業で欠席している学生を検索：

```
SELECT DISTINCT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN course_schedule cs ON sc.course_id = cs.course_id
WHERE NOT EXISTS (
    SELECT 1
    FROM attendance a
    WHERE a.student_id = s.student_id
    AND a.schedule_id = cs.schedule_id
    AND a.status = 'present'
)
AND cs.schedule_date <= CURRENT_DATE
ORDER BY s.student_id;
```

このクエリでは、各学生が受講しているはずの授業について、出席記録がないか「欠席」状態になっているレコードを検索しています。

相関サブクエリの処理の流れ

相関サブクエリがどのように処理されるかを理解するために、簡単な例で詳しく見てみましょう：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
    AND g.score > 90
);
```

このクエリの処理の流れは次のようになります：

1. 外部クエリがstudentsテーブルの最初の行を取得
2. 現在の学生ID（例えば301）に対して、サブクエリが実行される
    - `WHERE g.student_id = 301 AND g.score > 90`のレコードを検索
    - 該当レコードがあればEXISTSはtrueを返す
3. EXISTSがtrueなら、その学生が結果に含まれる
4. 外部クエリが次の行に進み、サブクエリが再び実行される
5. すべての行に対してこれを繰り返す

このように、相関サブクエリは外部クエリの各行に対して実行されるため、外部クエリの行数が多いと処理時間が長くなる可能性があります。

相関サブクエリとJOINの比較

相関サブクエリとJOINは多くの場合、同じ結果を得るための別のアプローチとして使えます。どちらを選ぶかは、クエリの目的、データ量、可読性などによって異なります。

例6：相関サブクエリとJOINの比較

相関サブクエリを使った例：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
    AND g.score > 90
);
```

同等のJOINを使った例：

```
SELECT DISTINCT s.student_id, s.student_name
FROM students s
JOIN grades g ON s.student_id = g.student_id
WHERE g.score > 90;
```

使い分けの基準：

1. **相関サブクエリが適している場合**：
    - 「存在する/しない」という条件チェックが主目的
    - 結果セットにサブクエリのテーブルからのデータが不要
    - 複雑な条件や集計結果との比較
    - 各行ごとに異なる条件での評価が必要
2. **JOINが適している場合**：
    - 両方のテーブルから情報を表示したい
    - 大量データの処理でパフォーマンスが重要
    - 複数のテーブルからのデータを組み合わせる
    - クエリの可読性を重視する

UPDATE文での相関サブクエリ

相関サブクエリはSELECT文だけでなく、UPDATE文でも利用できます。これにより、あるテーブルの値に基づいて別のテーブルを更新することが可能になります。

例7：UPDATE文での相関サブクエリ

例えば、学生の出席状況に基づいて成績に出席点を加算する場合：

```
UPDATE grades g
SET g.score = g.score + 5
WHERE g.grade_type = '最終評価'
AND EXISTS (
    SELECT 1
    FROM attendance a
    JOIN course_schedule cs ON a.schedule_id = cs.schedule_id
    WHERE a.student_id = g.student_id
    AND cs.course_id = g.course_id
    AND a.status = 'present'
    GROUP BY a.student_id, cs.course_id
    HAVING COUNT(*) / COUNT(DISTINCT cs.schedule_id) >= 0.9  -- 90%以上の出席率
);
```

このクエリでは、出席率が90%以上の学生の最終評価に5点加算しています。

相関サブクエリの注意点とパフォーマンス

相関サブクエリは強力ですが、以下の点に注意が必要です：

1. **パフォーマンス**：
    - 外部クエリの各行に対してサブクエリが実行されるため、データ量が多いとパフォーマンスが低下する可能性がある

- 特に深くネストした相関サブクエリでは注意が必要

2. **インデックス**：
   - 相関サブクエリのパフォーマンスを向上させるには、参照されるカラムにインデックスを設定することが重要
   - 特に、サブクエリ内のWHERE句で使用されるカラムには適切なインデックスを作成する

3. **代替手段の検討**：
   - JOINや一時テーブルなど、同じ結果を得るための別の方法も検討する
   - 特に大量データを扱う場合は、実行計画を比較して最適な方法を選択する

4. **可読性と保守性**：
   - 相関サブクエリは複雑になりがちなので、適切なコメントや命名規則を使って可読性を高める
   - 非常に複雑なロジックは、複数のステップに分解することも検討する

練習問題

問題20-1

grades（成績）テーブルを使って、各学生の平均点より10点以上高い成績を取得するSQLを書いてください。結果には学生ID、講座ID、評価タイプ、点数、および学生の平均点との差を「点差」という列名で表示してください。

問題20-2

courses（講座）テーブルとstudent_courses（受講）テーブルを使って、教師ごとに担当する講座の中で最も受講者が多い講座を取得するSQLを書いてください。結果には教師ID、教師名、講座名、受講者数を含めてください。相関サブクエリを使用してください。

問題20-3

students（学生）テーブルとattendance（出席）テーブルを使って、すべての授業に出席している学生（status = 'present'）を取得するSQLを書いてください。EXISTS演算子と相関サブクエリを使用してください。

問題20-4

course_schedule（授業カレンダー）テーブルとteachers（教師）テーブルを使って、担当している講座のうち一つでも欠席（status = 'absent'）の学生がいる授業を担当している教師を取得するSQLを書いてください。相関サブクエリとEXISTS演算子を使用してください。

問題20-5

grades（成績）テーブルとcourses（講座）テーブルを使って、講座ごとに平均点が全体の平均点よりも高い講座を取得するSQLを書いてください。結果には講座ID、講座名、講座平均点、全体平均点、差（講座平均点 - 全体平均点）を含めてください。相関サブクエリを使用してください。

問題20-6

students（学生）テーブル、student_courses（受講）テーブル、grades（成績）テーブルを使って、受講しているすべての講座で合格点（70点以上）を取っている学生を取得するSQLを書いてください。NOT EXISTS演算子と相関サブクエリを使用してください。

解答

解答20-1

```
SELECT
    g1.student_id,
    g1.course_id,
    g1.grade_type,
    g1.score,
    g1.score - (
        SELECT AVG(g2.score)
        FROM grades g2
        WHERE g2.student_id = g1.student_id
    ) AS 点差
FROM grades g1
WHERE g1.score >= (
    SELECT AVG(g2.score) + 10
    FROM grades g2
    WHERE g2.student_id = g1.student_id
)
ORDER BY 点差 DESC;
```

解答20-2

```
SELECT
    t.teacher_id,
    t.teacher_name,
    c.course_name,
    (SELECT COUNT(*) FROM student_courses sc WHERE sc.course_id = c.course_id) AS
受講者数
FROM teachers t
JOIN courses c ON t.teacher_id = c.teacher_id
WHERE (
    SELECT COUNT(*)
    FROM student_courses sc
    WHERE sc.course_id = c.course_id
) = (
    SELECT MAX(student_count)
    FROM (
        SELECT c2.course_id, COUNT(sc.student_id) AS student_count
        FROM courses c2
        LEFT JOIN student_courses sc ON c2.course_id = sc.course_id
        WHERE c2.teacher_id = t.teacher_id
        GROUP BY c2.course_id
    ) AS max_students
```

```
)
ORDER BY t.teacher_id, 受講者数 DESC;
```

解答20-3

```
SELECT s.student_id, s.student_name
FROM students s
WHERE NOT EXISTS (
    SELECT 1
    FROM course_schedule cs
    JOIN student_courses sc ON cs.course_id = sc.course_id
    WHERE sc.student_id = s.student_id
    AND NOT EXISTS (
        SELECT 1
        FROM attendance a
        WHERE a.schedule_id = cs.schedule_id
        AND a.student_id = s.student_id
        AND a.status = 'present'
    )
)
AND EXISTS (
    SELECT 1 FROM student_courses sc WHERE sc.student_id = s.student_id
)
ORDER BY s.student_id;
```

解答20-4

```
SELECT DISTINCT t.teacher_id, t.teacher_name
FROM teachers t
WHERE EXISTS (
    SELECT 1
    FROM course_schedule cs
    WHERE cs.teacher_id = t.teacher_id
    AND EXISTS (
        SELECT 1
        FROM attendance a
        WHERE a.schedule_id = cs.schedule_id
        AND a.status = 'absent'
    )
)
ORDER BY t.teacher_id;
```

解答20-5

```
SELECT
    c.course_id,
```

```
    c.course_name,
    (SELECT AVG(g.score) FROM grades g WHERE g.course_id = c.course_id) AS 講座平
均点,
    (SELECT AVG(score) FROM grades) AS 全体平均点,
    (SELECT AVG(g.score) FROM grades g WHERE g.course_id = c.course_id) -
    (SELECT AVG(score) FROM grades) AS 差
FROM courses c
WHERE (
    SELECT AVG(g.score)
    FROM grades g
    WHERE g.course_id = c.course_id
) > (
    SELECT AVG(score)
    FROM grades
)
ORDER BY 差 DESC;
```

解答20-6

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1 FROM student_courses sc WHERE sc.student_id = s.student_id
)
AND NOT EXISTS (
    SELECT 1
    FROM student_courses sc
    WHERE sc.student_id = s.student_id
    AND EXISTS (
        SELECT 1
        FROM grades g
        WHERE g.student_id = s.student_id
        AND g.course_id = sc.course_id
        AND g.score < 70
    )
)
ORDER BY s.student_id;
```

まとめ

この章では、相関サブクエリについて詳しく学びました：

1. **相関サブクエリの概念**：
    - 外部クエリの値を参照するサブクエリ
    - 外部クエリの各行に対して実行される
    - 「各〜について」という条件を実現できる強力な機能
2. **相関サブクエリの構文と使用例**：

- 基本構文と動作原理
- 学生の平均点より高い成績の抽出
- 講座ごとの平均点との比較
- 教師ごとの最大受講者数の講座の特定

3. **EXISTS演算子との組み合わせ**：
   - レコードの存在チェックに効果的なEXISTS/NOT EXISTS
   - 複雑な条件を持つデータの抽出方法
4. **処理の流れとパフォーマンス**：
   - 相関サブクエリの実行順序と動作の理解
   - パフォーマンスへの影響と最適化方法
5. **相関サブクエリとJOINの比較**：
   - 同じ結果を得るための異なるアプローチ
   - 使い分けの基準と考慮点
6. **UPDATE文での活用**：
   - データ更新における相関サブクエリの応用

相関サブクエリは、複雑な条件や「各〜について」という条件を実現するための強力なツールです。適切に使用することで、単純なJOINでは実現しにくい複雑なデータ抽出や条件付きの操作が可能になります。ただし、パフォーマンスへの影響を考慮し、適切な場面で使用することが重要です。

次の章では、「EXISTS句とサブクエリ：存在チェックの高度な使い方」について学び、EXISTSとサブクエリをさらに深く理解していきます。

---

21. 集合演算：UNION、INTERSECT、EXCEPT

はじめに

これまでの章では、サブクエリや相関サブクエリを使用して複雑なデータ検索を行う方法を学びました。SQLにはもう一つの強力な技術があります。それが「集合演算」です。

集合演算とは、複数のSELECT文の結果を結合したり、比較したりする操作のことです。例えば以下のようなケースで活用できます：

- 「複数の検索条件の結果を一つにまとめたい」
- 「二つの検索結果の共通部分だけを抽出したい」
- 「ある条件の結果から別の条件の結果を除外したい」

この章では、主要な集合演算（UNION、INTERSECT、EXCEPT）の基本概念と実践的な使用方法について学びます。

集合演算とは

集合演算とは、複数のクエリ結果を数学的な「集合」として扱い、それらを組み合わせる操作のことです。
主な集合演算には以下の3種類があります：

> **用語解説：**
>
> - **集合演算**：複数のSELECT文の結果を集合として扱い、合成する演算のことです。
> - **UNION（和集合）**：二つのクエリ結果を結合し、重複を排除した結果を返します。
> - **INTERSECT（積集合）**：二つのクエリ結果の共通部分のみを返します。
> - **EXCEPT（差集合）**：一つ目のクエリ結果から二つ目のクエリ結果に含まれるレコードを除外した結果を返します。

集合演算の基本規則

集合演算を使用する際には、以下の基本規則に従う必要があります：

1. **カラム数の一致**：集合演算で結合する各SELECT文は、同じ数のカラムを持つ必要があります。
2. **データ型の互換性**：対応するカラムは互換性のあるデータ型である必要があります。
3. **カラム名**：最終的な結果のカラム名は、最初のSELECT文で指定されたものが使用されます。
4. **ORDER BY**：集合演算の結果全体に対して一番最後に一度だけ指定できます。
5. **NULL値**：集合演算においてNULL値も通常の値として扱われます。

UNION（和集合）

UNIONは、二つ以上のクエリ結果を結合し、重複を排除した結果を返します。

> **用語解説：**
>
> - **UNION**：複数のSELECT文の結果を結合し、重複を排除する演算子です。
> - **UNION ALL**：複数のSELECT文の結果を結合し、重複を排除せずにすべての行を返す演算子です。

基本構文

```
SELECT カラム1, カラム2, ...
FROM テーブル1
WHERE 条件1

UNION

SELECT カラム1, カラム2, ...
FROM テーブル2
WHERE 条件2;
```

例1：UNIONの基本

ITかAI関連の講座を受講している学生の一覧を取得するクエリ：

```
-- ITのための基礎知識を受講している学生
SELECT s.student_id, s.student_name, '1' AS course_id, 'ITのための基礎知識' AS
```

```
course_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '1'

UNION

-- AI・機械学習入門を受講している学生
SELECT s.student_id, s.student_name, '7' AS course_id, 'AI・機械学習入門' AS
course_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '7'

ORDER BY student_id;
```

実行結果（一部）：

student_id	student_name	course_id	course_name
301	黒沢春馬	1	ITのための基礎知識
302	新垣愛留	1	ITのための基礎知識
302	新垣愛留	7	AI・機械学習入門
303	柴崎春花	1	ITのための基礎知識
305	河口菜恵子	7	AI・機械学習入門
…	…	…	…

このクエリでは、講座ID=1（ITのための基礎知識）または講座ID=7（AI・機械学習入門）を受講している学生をUNIONで結合しています。ある学生が両方の講座を受講している場合（例：302の新垣愛留）、その学生は両方の講座で結果に含まれます。

例2：UNION ALLを使用した重複を許容する例

```
-- 出席率が高い学生（90%以上）
SELECT s.student_id, s.student_name, '高出席率' AS category
FROM students s
JOIN attendance a ON s.student_id = a.student_id
GROUP BY s.student_id, s.student_name
HAVING AVG(CASE WHEN a.status = 'present' THEN 100 ELSE 0 END) >= 90

UNION ALL

-- 成績が優秀な学生（85点以上）
SELECT s.student_id, s.student_name, '高成績' AS category
FROM students s
JOIN grades g ON s.student_id = g.student_id
GROUP BY s.student_id, s.student_name
```

```
HAVING AVG(g.score) >= 85

ORDER BY category, student_id;
```

このクエリでは、出席率の高い学生と成績が優秀な学生を `UNION ALL` で結合しています。`UNION ALL` は重複を排除しないため、ある学生が両方の条件を満たす場合、結果に2回含まれます（出席率が高く、かつ成績も優秀な学生）。

実行結果（一部）：

student_id	student_name	category
301	黒沢春馬	高出席率
302	新垣愛留	高出席率
307	織田柚夏	高出席率
302	新垣愛留	高成績
308	永田悦子	高成績
311	鈴木健太	高成績
…	…	…

UNIONとUNION ALLの違い

- **UNION**：重複する行を排除します（重複チェックのためにソートが行われるため、パフォーマンスへの影響があります）。
- **UNION ALL**：重複する行もすべて保持します（ソートが不要なため、通常UNIONより高速です）。

重複を排除する必要がなければ、パフォーマンスの観点から`UNION ALL`を使用する方が好ましいことが多いです。

INTERSECT（積集合）

INTERSECTは、二つのクエリ結果の共通部分（両方のクエリ結果に含まれる行）のみを返します。

> **用語解説**：
>
> - **INTERSECT**：二つのSELECT文の結果の共通部分のみを返す演算子です。

基本構文

```
SELECT カラム1, カラム2, ...
FROM テーブル1
WHERE 条件1

INTERSECT

SELECT カラム1, カラム2, ...
```

```
FROM テーブル2
WHERE 条件2;
```

例3：INTERSECTの基本

ITの基礎知識とAI・機械学習入門の両方を受講している学生を検索：

```
-- ITのための基礎知識を受講している学生
SELECT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '1'

INTERSECT

-- AI･機械学習入門を受講している学生
SELECT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '7'

ORDER BY student_id;
```

実行結果：

student_id	student_name
302	新垣愛留
310	吉川伽羅
315	遠藤勇気
…	…

このクエリでは、講座ID=1と講座ID=7の両方を受講している学生のみが結果に含まれます。

例4：複数条件の共通部分

中間テストとレポート1の両方で85点以上を取得した学生を検索：

```
-- 中間テストで85点以上の学生
SELECT s.student_id, s.student_name
FROM students s
JOIN grades g ON s.student_id = g.student_id
WHERE g.grade_type = '中間テスト' AND g.score >= 85

INTERSECT

-- レポート1で85点以上の学生
```

```
SELECT s.student_id, s.student_name
FROM students s
JOIN grades g ON s.student_id = g.student_id
WHERE g.grade_type = 'レポート1' AND g.score >= 85

ORDER BY student_id;
```

実行結果：

student_id	student_name
302	新垣愛留
308	永田悦子
311	鈴木健太
...	...

EXCEPT（差集合）

EXCEPTは、最初のクエリ結果から2番目のクエリ結果に含まれる行を除外した結果を返します。

> **用語解説**：
>
> - **EXCEPT**：一つ目のSELECT文の結果から二つ目のSELECT文の結果に含まれる行を除外する演算子です。一部のデータベース（例：Oracle）では、MINUS演算子が同じ目的で使用されます。

基本構文

```
SELECT カラム1, カラム2, ...
FROM テーブル1
WHERE 条件1

EXCEPT

SELECT カラム1, カラム2, ...
FROM テーブル2
WHERE 条件2;
```

例5：EXCEPTの基本

ITのための基礎知識は受講しているが、AI・機械学習入門は受講していない学生を検索：

```
-- ITのための基礎知識を受講している学生
SELECT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '1'
```

```
EXCEPT

-- AI・機械学習入門を受講している学生
SELECT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '7'

ORDER BY student_id;
```

実行結果：

student_id	student_name
301	黒沢春馬
303	柴崎春花
306	河田咲奈
…	…

このクエリでは、講座ID=1を受講しているが、講座ID=7は受講していない学生だけが結果に含まれます。

例6：除外条件を使った検索

出席記録はあるが、成績記録がない学生を検索：

```
-- 出席記録がある学生
SELECT DISTINCT s.student_id, s.student_name
FROM students s
JOIN attendance a ON s.student_id = a.student_id

EXCEPT

-- 成績記録がある学生
SELECT DISTINCT s.student_id, s.student_name
FROM students s
JOIN grades g ON s.student_id = g.student_id

ORDER BY student_id;
```

実行結果：

student_id	student_name
312	佐々木優斗
317	長谷川結衣
321	西山太一

student_id	student_name
...	...

集合演算子の組み合わせ

複数の集合演算子を組み合わせて使用することもできます。その場合、括弧（）を使用して演算の優先順位を明確にしましょう。

例7：集合演算子の組み合わせ

IT関連の講座か、クラウド関連の講座を受講していて、かつプログラミング関連の講座は受講していない学生を検索：

```
-- IT関連の講座を受講している学生
(SELECT DISTINCT s.student_id, s.student_name
 FROM students s
 JOIN student_courses sc ON s.student_id = sc.student_id
 WHERE sc.course_id IN ('1', '7')  -- ITの基礎知識またはAI・機械学習入門

 UNION

 -- クラウド関連の講座を受講している学生
 SELECT DISTINCT s.student_id, s.student_name
 FROM students s
 JOIN student_courses sc ON s.student_id = sc.student_id
 WHERE sc.course_id IN ('9', '16'))  -- クラウドコンピューティングまたはクラウドネイティブアーキテクチャ

EXCEPT

-- プログラミング関連の講座を受講している学生
SELECT DISTINCT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id IN ('3', '4', '8');  -- Cプログラミングまたはウェブアプリまたはモバイルアプリ
```

このクエリでは：

1. まず、IT関連またはクラウド関連の講座を受講している学生を`UNION`で結合
2. 次に、プログラミング関連の講座を受講している学生を`EXCEPT`で除外

集合演算を使用する状況

集合演算は以下のような状況で特に役立ちます：

1. **複数の条件を満たすレコードの検索**：
    - 複数の条件すべてを満たすレコード（`INTERSECT`）

- いずれかの条件を満たすレコード（UNION）
- 特定の条件は満たすが別の条件は満たさないレコード（EXCEPT）

2. **異なるテーブルからの類似データの結合**：
   - 構造は同じだが別々のテーブルにあるデータを結合する（例：アーカイブと現行データ）
3. **複雑なレポート作成**：
   - 複数のカテゴリを含むレポート作成
   - 異なる条件のデータを一つのレポートにまとめる
4. **差分分析**：
   - 二つのデータセット間の違いを特定する（例：前月と今月の比較）

集合演算とサブクエリやJOINの比較

集合演算に代わる方法として、サブクエリやJOINを使用することも可能な場合があります。以下に、それぞれの方法の比較を示します：

例8：集合演算とサブクエリの比較

集合演算を使う場合（INTERSECT）：

```
-- 中間テストで85点以上の学生
SELECT s.student_id, s.student_name
FROM students s
JOIN grades g ON s.student_id = g.student_id
WHERE g.grade_type = '中間テスト' AND g.score >= 85

INTERSECT

-- レポート1で85点以上の学生
SELECT s.student_id, s.student_name
FROM students s
JOIN grades g ON s.student_id = g.student_id
WHERE g.grade_type = 'レポート1' AND g.score >= 85;
```

サブクエリを使う場合（IN）：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE s.student_id IN (
    SELECT g1.student_id
    FROM grades g1
    WHERE g1.grade_type = '中間テスト' AND g1.score >= 85
)
AND s.student_id IN (
    SELECT g2.student_id
```

```
    FROM grades g2
    WHERE g2.grade_type = 'レポート1' AND g2.score >= 85
);
```

JOINを使う場合：

```
SELECT DISTINCT s.student_id, s.student_name
FROM students s
JOIN grades g1 ON s.student_id = g1.student_id
JOIN grades g2 ON s.student_id = g2.student_id
WHERE g1.grade_type = '中間テスト' AND g1.score >= 85
AND g2.grade_type = 'レポート1' AND g2.score >= 85;
```

使い分けの基準：

1. **集合演算が適している場合**：
   - クエリが概念的に「和集合」「積集合」「差集合」として考えやすい場合
   - 複数の異なるクエリ結果を組み合わせる場合
   - クエリが複雑で、分割して考えた方が理解しやすい場合
2. **サブクエリが適している場合**：
   - 一方のクエリ結果がフィルタとして使用される場合
   - EXISTS/NOT EXISTSでの存在チェックが必要な場合
   - 相関サブクエリで行ごとの条件チェックが必要な場合
3. **JOINが適している場合**：
   - 複数のテーブルからの情報を組み合わせて表示する場合
   - 結合条件が明確で、パフォーマンスが重要な場合
   - 追加の集計や条件が必要な場合

集合演算のパフォーマンスと注意点

集合演算を使用する際の主な注意点は以下の通りです：

1. **ソーティングコスト**：
   - `UNION`、`INTERSECT`、`EXCEPT`は重複排除のためのソートが必要なため、大きなデータセットでは処理が遅くなる可能性があります。
   - 重複排除が不要な場合は`UNION ALL`を使用すると効率的です。
2. **メモリ消費**：
   - 大きなデータセットを結合する場合、メモリ消費が大きくなることがあります。
   - 特に、`INTERSECT`と`EXCEPT`は両方のクエリ結果を一時的に保存する必要があります。
3. **クエリの最適化**：

- 大きなデータセットの集合演算では、各SELECT文を最適化して、必要最小限のデータだけを処理するようにしましょう。
- 可能な限り、集合演算の前に条件でフィルタリングして結果セットを小さくしましょう。

4. **代替手段の検討**：

- 同じ結果を得るためのJOINや条件式など、より効率的な方法がないか検討しましょう。
- 特にパフォーマンスが重要な場合は、実行計画を比較して最適な方法を選択しましょう。

練習問題

問題21-1

UNION演算子を使用して、「ITのための基礎知識」（course_id = 1）と「データベース設計と実装」（course_id = 5）のいずれかを受講している学生の一覧を取得するSQLを書いてください。結果には学生ID、学生名、講座名を含めてください。

問題21-2

INTERSECT演算子を使用して、「Webアプリケーション開発」（course_id = 4）と「モバイルアプリ開発」（course_id = 8）の両方を受講している学生の一覧を取得するSQLを書いてください。

問題21-3

EXCEPT演算子を使用して、「プロジェクト管理手法」（course_id = 10）は受講しているが、「UNIX入門」（course_id = 2）は受講していない学生の一覧を取得するSQLを書いてください。

問題21-4

UNION ALL演算子を使用して、出席率（status = 'present'の割合）が85%以上の学生と、平均点が85点以上の学生をそれぞれ「高出席」「高成績」のカテゴリー別に一覧表示し、同じ学生が両方の条件を満たす場合は両方のカテゴリーに表示されるようにするSQLを書いてください。

問題21-5

UNION、INTERSECTおよびEXCEPT演算子を組み合わせて、次の条件を満たす学生の一覧を取得するSQLを書いてください：

- 「データベース設計と実装」または「データ分析と可視化」を受講している
- 「プロジェクト管理手法」は受講していない
- 中間テストの成績が80点以上である

問題21-6

講座の組み合わせのうち、同じ学生が受講している頻度の高い組み合わせトップ5を見つけるために、INTERSECT演算子とUNION演算子を組み合わせたSQLを書いてください。結果には講座ID、講座名、受講者数を含めてください。

解答

解答21-1

```sql
-- ITのための基礎知識を受講している学生
SELECT s.student_id, s.student_name, c.course_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN courses c ON sc.course_id = c.course_id
WHERE c.course_id = '1'

UNION

-- データベース設計と実装を受講している学生
SELECT s.student_id, s.student_name, c.course_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN courses c ON sc.course_id = c.course_id
WHERE c.course_id = '5'

ORDER BY student_id;
```

解答21-2

```sql
-- Webアプリケーション開発を受講している学生
SELECT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '4'

INTERSECT

-- モバイルアプリ開発を受講している学生
SELECT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '8'

ORDER BY student_id;
```

解答21-3

```sql
-- プロジェクト管理手法を受講している学生
SELECT s.student_id, s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '10'

EXCEPT

-- UNIX入門を受講している学生
SELECT s.student_id, s.student_name
```

```
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '2'

ORDER BY student_id;
```

解答21-4

```
-- 出席率が85%以上の学生
SELECT s.student_id, s.student_name, '高出席' AS category
FROM students s
JOIN attendance a ON s.student_id = a.student_id
GROUP BY s.student_id, s.student_name
HAVING AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END) >= 85

UNION ALL

-- 平均点が85点以上の学生
SELECT s.student_id, s.student_name, '高成績' AS category
FROM students s
JOIN grades g ON s.student_id = g.student_id
GROUP BY s.student_id, s.student_name
HAVING AVG(g.score) >= 85

ORDER BY student_id, category;
```

解答21-5

```
-- データベース設計と実装またはデータ分析と可視化を受講している学生
(SELECT DISTINCT s.student_id, s.student_name
 FROM students s
 JOIN student_courses sc ON s.student_id = sc.student_id
 WHERE sc.course_id IN ('5', '11'))

EXCEPT

-- プロジェクト管理手法を受講している学生
(SELECT s.student_id, s.student_name
 FROM students s
 JOIN student_courses sc ON s.student_id = sc.student_id
 WHERE sc.course_id = '10')

INTERSECT

-- 中間テストの成績が80点以上の学生
(SELECT s.student_id, s.student_name
 FROM students s
 JOIN grades g ON s.student_id = g.student_id
 WHERE g.grade_type = '中間テスト' AND g.score >= 80)
```

```
ORDER BY student_id;
```

解答21-6

```
WITH course_pairs AS (
    -- 講座ペアごとの受講学生数を計算
    SELECT
        c1.course_id AS course_id1,
        c1.course_name AS course_name1,
        c2.course_id AS course_id2,
        c2.course_name AS course_name2,
        COUNT(*) AS common_students
    FROM courses c1
    JOIN student_courses sc1 ON c1.course_id = sc1.course_id
    JOIN student_courses sc2 ON sc1.student_id = sc2.student_id
    JOIN courses c2 ON sc2.course_id = c2.course_id
    WHERE c1.course_id < c2.course_id  -- 重複を避ける
    GROUP BY c1.course_id, c1.course_name, c2.course_id, c2.course_name
)
-- 上位5件を取得
SELECT
    course_id1,
    course_name1,
    course_id2,
    course_name2,
    common_students AS 共通受講者数
FROM course_pairs
ORDER BY common_students DESC
LIMIT 5;
```

注：この解答の例は、共通テーブル式（WITH句）を使用しています。これは次の章で詳しく学習する内容ですが、ここでは効率的な解答のために先行して使用しています。

まとめ

この章では、SQL集合演算について詳しく学びました：

1. **集合演算の基本概念**：
    - 複数のクエリ結果を集合として扱う演算
    - 集合演算を使用する際の基本規則（カラム数の一致、データ型の互換性など）
2. **UNION（和集合）**：
    - 複数のクエリ結果を結合し、重複を排除する方法
    - UNION ALLで重複を保持する方法
    - パフォーマンスの違いと使い分け
3. **INTERSECT（積集合）**：

    - 複数のクエリ結果の共通部分を抽出する方法
    - 複数条件をすべて満たすレコードの検索

4. **EXCEPT（差集合）**：

    - 一方のクエリ結果から他方のクエリ結果を除外する方法
    - 特定の条件を満たすが別の条件は満たさないレコードの検索

5. **集合演算子の組み合わせ**：

    - 複数の集合演算子を組み合わせた複雑な条件の表現
    - 括弧を使った演算優先順位の制御

6. **集合演算とサブクエリやJOINの比較**：

    - 同じ結果を得るための異なるアプローチ
    - 使い分けの基準

集合演算は、複雑なレポート作成や条件付きデータ分析において強力なツールとなります。適切に使用することで、複雑な条件を持つデータの抽出や異なるデータセットの比較が効率的に行えるようになります。

次の章では、「EXISTS演算子：存在確認のクエリ」について学び、レコードの存在確認に特化したSQLテクニックを深く理解していきます。

---

22. EXISTS演算子：存在確認のクエリ

はじめに

これまでの章で、サブクエリ、相関サブクエリ、集合演算について学び、その中でEXISTS演算子にも触れてきました。EXISTS演算子は、レコードの存在確認に特化した非常に強力で頻繁に使用されるSQL機能です。

EXISTS演算子は以下のような場面で特に威力を発揮します：

- 「特定の条件を満たすレコードが存在する場合のみ」という条件付き検索
- 「関連テーブルに対応するデータがある/ない」という存在チェック
- 「複数の条件をすべて満たす」または「いずれかの条件を満たす」という複雑な条件設定
- 大きなデータセットでの効率的な存在確認

この章では、EXISTS演算子の詳細な使い方、パフォーマンス特性、そして実践的な活用方法について深く学びます。

EXISTS演算子とは

EXISTS演算子は、サブクエリが少なくとも1行の結果を返すかどうかをチェックする演算子です。結果の値そのものは重要ではなく、「存在するかどうか」だけが評価されます。

> **用語解説**：
>
> - **EXISTS**：サブクエリが少なくとも1行の結果を返すかどうかをチェックする演算子です。
> - **NOT EXISTS**：サブクエリが結果を1行も返さないかどうかをチェックする演算子です。

> - **存在確認**：データが存在するかどうかを確認することで、値そのものではなく存在の有無だけを調べる操作です。

EXISTSの特徴

1. **真偽値の返却**：EXISTSは常にTRUE（真）またはFALSE（偽）を返します。
2. **効率的な処理**：サブクエリが1行でも条件に一致すれば、それ以上の検索を停止します（ショートサーキット評価）。
3. **NULLに影響されない**：サブクエリの結果にNULL値が含まれていても正常に動作します。
4. **相関サブクエリとの相性**：外部クエリとの相関関係を持つサブクエリと組み合わせて使用されることが多いです。

EXISTS演算子の基本構文

```
SELECT カラム1, カラム2, ...
FROM テーブル1
WHERE EXISTS (
    SELECT 1  -- または任意のカラム
    FROM テーブル2
    WHERE 条件
);
```

EXISTSのサブクエリでは、`SELECT 1`や`SELECT *`のように書くのが一般的です。実際の値は使用されないため、どのような値を SELECT しても結果は同じです。

EXISTS演算子の基本例

例1：EXISTSの基本的な使用

成績記録がある学生のみを取得：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
)
ORDER BY s.student_id;
```

このクエリでは：

1. 外部クエリでstudentsテーブルから各学生を処理します。
2. 各学生に対して、サブクエリでその学生の成績記録があるかチェックします。
3. 成績記録が存在する学生のみが結果に含まれます。

実行結果：

student_id	student_name
301	黒沢春馬
302	新垣愛留
303	柴崎春花
306	河田咲奈
...	...

例2：NOT EXISTSの使用

成績記録がない学生を取得：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE NOT EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
)
ORDER BY s.student_id;
```

NOT EXISTSは、サブクエリが結果を1行も返さない場合にTRUEを返します。

実行結果：

student_id	student_name
304	森下風凛
305	河口菜恵子
309	相沢吉夫
312	佐々木優斗
...	...

EXISTS vs IN：違いとパフォーマンス

EXISTSとINは、似たような結果を得ることができる場合がありますが、重要な違いがあります。

例3：EXISTSとINの比較

EXISTSを使用した場合：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
```

```
    SELECT 1
    FROM student_courses sc
    WHERE sc.student_id = s.student_id
    AND sc.course_id = '1'
);
```

INを使用した場合：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE s.student_id IN (
    SELECT sc.student_id
    FROM student_courses sc
    WHERE sc.course_id = '1'
);
```

EXISTSとINの主な違い

項目	EXISTS	IN
NULL値の扱い	NULL値に影響されない	サブクエリにNULLが含まれると予期しない結果になることがある
パフォーマンス	1行見つかると検索停止（効率的）	場合によってはすべての結果を評価する必要がある
相関サブクエリ	外部クエリとの相関関係を自然に表現できる	相関関係の表現が複雑になる場合がある
重複の扱い	重複を気にする必要がない	サブクエリの重複が結果に影響することがある

NULL値の問題例

INでの問題例：

```
-- NOT INでNULL値が含まれる場合の問題
SELECT s.student_id, s.student_name
FROM students s
WHERE s.student_id NOT IN (
    SELECT sc.student_id
    FROM student_courses sc
    WHERE sc.course_id = '999'  -- 存在しない講座ID
);
-- 上記クエリは期待通りに動作しない可能性があります（NULLが含まれる場合）
```

NOT EXISTSを使用した場合：

```
-- NOT EXISTSは安全に動作します
SELECT s.student_id, s.student_name
FROM students s
WHERE NOT EXISTS (
    SELECT 1
    FROM student_courses sc
    WHERE sc.student_id = s.student_id
    AND sc.course_id = '999'
);
```

複雑な存在確認パターン

例4：複数条件の存在確認

中間テストとレポート1の両方を提出している学生を検索：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM grades g1
    WHERE g1.student_id = s.student_id
    AND g1.grade_type = '中間テスト'
)
AND EXISTS (
    SELECT 1
    FROM grades g2
    WHERE g2.student_id = s.student_id
    AND g2.grade_type = 'レポート1'
)
ORDER BY s.student_id;
```

このクエリでは、各学生について「中間テストがある」AND「レポート1がある」という2つの条件をそれぞれ別のEXISTS句でチェックしています。

例5：条件付き存在確認

85点以上の成績を少なくとも1つ持つ学生を検索：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
    AND g.score >= 85
)
ORDER BY s.student_id;
```

例6：複雑な相関条件

各講座において平均点以上を取った学生を検索：

```
SELECT DISTINCT s.student_id, s.student_name, c.course_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN courses c ON sc.course_id = c.course_id
WHERE EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
    AND g.course_id = sc.course_id
    AND g.score >= (
        SELECT AVG(score)
        FROM grades g2
        WHERE g2.course_id = g.course_id
    )
)
ORDER BY s.student_id, c.course_name;
```

このクエリでは、EXISTSサブクエリの中にさらにサブクエリが含まれており、各講座の平均点と比較しています。

NOT EXISTSの高度な活用

NOT EXISTSは、「～が存在しない」という条件を表現するために使用され、データの欠損や未達成条件の特定に非常に有効です。

例7：完全な条件の否定

すべての講座で80点以上を取っている学生（80点未満の成績が存在しない学生）を検索：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    -- 少なくとも1つの成績記録がある
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
)
AND NOT EXISTS (
    -- 80点未満の成績が存在しない
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
    AND g.score < 80
```

```
)
ORDER BY s.student_id;
```

例8：関連データの完全性チェック

受講しているすべての講座に出席している学生を検索：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    -- 受講している講座がある
    SELECT 1
    FROM student_courses sc
    WHERE sc.student_id = s.student_id
)
AND NOT EXISTS (
    -- 欠席した授業が存在しない
    SELECT 1
    FROM student_courses sc
    JOIN course_schedule cs ON sc.course_id = cs.course_id
    WHERE sc.student_id = s.student_id
    AND NOT EXISTS (
        SELECT 1
        FROM attendance a
        WHERE a.student_id = sc.student_id
        AND a.schedule_id = cs.schedule_id
        AND a.status = 'present'
    )
)
ORDER BY s.student_id;
```

EXISTS演算子を使った実践的なクエリパターン

例9：階層的な存在確認

特定の教師が担当する講座を受講し、かつその講座で良い成績を収めている学生を検索：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM student_courses sc
    JOIN courses c ON sc.course_id = c.course_id
    JOIN teachers t ON c.teacher_id = t.teacher_id
    WHERE sc.student_id = s.student_id
    AND t.teacher_name = '寺内鞍'
    AND EXISTS (
        SELECT 1
        FROM grades g
```

```
        WHERE g.student_id = s.student_id
        AND g.course_id = sc.course_id
        AND g.score >= 85
    )
)
ORDER BY s.student_id;
```

例10：時系列での存在確認

最近の授業（直近30日間）で欠席したことがない学生を検索：

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    -- 直近30日間に授業がある
    SELECT 1
    FROM student_courses sc
    JOIN course_schedule cs ON sc.course_id = cs.course_id
    WHERE sc.student_id = s.student_id
    AND cs.schedule_date >= CURRENT_DATE - INTERVAL 30 DAY
)
AND NOT EXISTS (
    -- 直近30日間で欠席していない
    SELECT 1
    FROM student_courses sc
    JOIN course_schedule cs ON sc.course_id = cs.course_id
    LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id
                          AND a.student_id = s.student_id
    WHERE sc.student_id = s.student_id
    AND cs.schedule_date >= CURRENT_DATE - INTERVAL 30 DAY
    AND (a.status IS NULL OR a.status != 'present')
)
ORDER BY s.student_id;
```

EXISTS演算子のパフォーマンス最適化

EXISTSを効率的に使用するためのポイントを説明します。

1. インデックスの活用

EXISTS句で使用される結合条件にはインデックスを設定することが重要です：

```
-- 効率的なEXISTSクエリの例
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id  -- student_idにインデックスが必要
```

```
    AND g.score >= 90
);
```

2. 適切な条件の配置

より選択性の高い条件を先に配置することで、処理を効率化できます：

```
-- 効率的な条件の配置
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM grades g
    WHERE g.score >= 95  -- 選択性の高い条件を先に
    AND g.student_id = s.student_id  -- 結合条件を後に
);
```

3. 不要な結合の回避

EXISTSではサブクエリの結果値が使用されないため、必要最小限の結合にとどめます：

```
-- 効率的なEXISTSクエリ
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM student_courses sc
    WHERE sc.student_id = s.student_id
    AND sc.course_id = '1'
    -- 不要な結合（coursesテーブルなど）は行わない
);
```

EXISTS演算子と他の手法の使い分け

1. EXISTS vs JOIN

EXISTSが適している場合：

- 存在確認だけが目的
- 関連テーブルからのデータが不要
- 1対多の関係で重複を避けたい

```
-- EXISTS：存在確認のみ
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
```

```
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
);
```

JOINが適している場合：

- 関連テーブルからのデータも取得したい
- パフォーマンスが重要
- 集計などの追加処理が必要

```
-- JOIN：関連データも取得
SELECT DISTINCT s.student_id, s.student_name, g.score
FROM students s
JOIN grades g ON s.student_id = g.student_id;
```

2. EXISTS vs IN

EXISTSが適している場合：

- サブクエリにNULL値が含まれる可能性がある
- 相関サブクエリを使用する
- 複雑な条件がある

INが適している場合：

- 単純な値のリストとの照合
- サブクエリが非相関で簡潔
- NULLが含まれない確実な場合

練習問題

問題22-1

EXISTS演算子を使用して、レポート1の成績が90点以上の学生が在籍している講座を取得するSQLを書いてください。結果には講座ID、講座名、担当教師名を含めてください。

問題22-2

NOT EXISTS演算子を使用して、2025年5月20日以降の授業にまだ一度も欠席（status = 'absent'）していない学生を取得するSQLを書いてください。ただし、授業に出席した記録がある学生のみを対象とします。

問題22-3

EXISTS演算子を使用して、担当しているすべての講座で平均受講者数が10人以上の教師を取得するSQLを書いてください。NOT EXISTSも組み合わせて使用してください。

問題22-4

EXISTS演算子を使用して、プログラミング関連の講座（course_id = 3, 4, 8のいずれか）を受講し、かつそのすべての講座で80点以上を取得している学生を取得するSQLを書いてください。

問題22-5

EXISTS演算子を使用して、同じ教師が担当する複数の講座を受講している学生を取得するSQLを書いてください。結果には学生ID、学生名、教師名を含めてください。

問題22-6

NOT EXISTS演算子を使用して、受講している講座があるにも関わらず、成績記録が一切ない学生を特定するSQLを書いてください。このような学生は成績入力漏れの可能性があります。

解答

解答22-1

```
SELECT c.course_id, c.course_name, t.teacher_name
FROM courses c
JOIN teachers t ON c.teacher_id = t.teacher_id
WHERE EXISTS (
    SELECT 1
    FROM grades g
    JOIN students s ON g.student_id = s.student_id
    JOIN student_courses sc ON s.student_id = sc.student_id
    WHERE sc.course_id = c.course_id
    AND g.course_id = c.course_id
    AND g.grade_type = 'レポート1'
    AND g.score >= 90
)
ORDER BY c.course_id;
```

解答22-2

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    -- 出席記録がある学生
    SELECT 1
    FROM attendance a
    WHERE a.student_id = s.student_id
)
AND NOT EXISTS (
    -- 2025年5月20日以降に欠席していない
    SELECT 1
    FROM attendance a
    JOIN course_schedule cs ON a.schedule_id = cs.schedule_id
    WHERE a.student_id = s.student_id
    AND cs.schedule_date >= '2025-05-20'
```

```
    AND a.status = 'absent'
)
ORDER BY s.student_id;
```

解答22-3

```
SELECT t.teacher_id, t.teacher_name
FROM teachers t
WHERE EXISTS (
    -- 担当講座がある
    SELECT 1
    FROM courses c
    WHERE c.teacher_id = t.teacher_id
)
AND NOT EXISTS (
    -- 受講者数が10人未満の講座が存在しない
    SELECT 1
    FROM courses c
    WHERE c.teacher_id = t.teacher_id
    AND (
        SELECT COUNT(*)
        FROM student_courses sc
        WHERE sc.course_id = c.course_id
    ) < 10
)
ORDER BY t.teacher_id;
```

解答22-4

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    -- プログラミング関連講座を受講している
    SELECT 1
    FROM student_courses sc
    WHERE sc.student_id = s.student_id
    AND sc.course_id IN ('3', '4', '8')
)
AND NOT EXISTS (
    -- 受講しているプログラミング関連講座で80点未満の成績が存在しない
    SELECT 1
    FROM student_courses sc
    JOIN grades g ON sc.student_id = g.student_id AND sc.course_id = g.course_id
    WHERE sc.student_id = s.student_id
    AND sc.course_id IN ('3', '4', '8')
    AND g.score < 80
)
ORDER BY s.student_id;
```

解答22-5

```
SELECT DISTINCT s.student_id, s.student_name, t.teacher_name
FROM students s
JOIN student_courses sc1 ON s.student_id = sc1.student_id
JOIN courses c1 ON sc1.course_id = c1.course_id
JOIN teachers t ON c1.teacher_id = t.teacher_id
WHERE EXISTS (
    -- 同じ教師が担当する別の講座も受講している
    SELECT 1
    FROM student_courses sc2
    JOIN courses c2 ON sc2.course_id = c2.course_id
    WHERE sc2.student_id = s.student_id
    AND c2.teacher_id = t.teacher_id
    AND sc2.course_id != sc1.course_id
)
ORDER BY s.student_id, t.teacher_name;
```

解答22-6

```
SELECT s.student_id, s.student_name
FROM students s
WHERE EXISTS (
    -- 受講している講座がある
    SELECT 1
    FROM student_courses sc
    WHERE sc.student_id = s.student_id
)
AND NOT EXISTS (
    -- 成績記録が存在しない
    SELECT 1
    FROM grades g
    WHERE g.student_id = s.student_id
)
ORDER BY s.student_id;
```

まとめ

この章では、EXISTS演算子について詳しく学びました：

1. **EXISTS演算子の基本概念**：
    - レコードの存在確認に特化した演算子
    - TRUE/FALSEの真偽値を返し、値そのものは評価しない
    - ショートサーキット評価による効率的な処理
2. **EXISTS vs IN の違い**：
    - NULL値の扱いにおける安全性の違い

- パフォーマンス特性の違い
- 相関サブクエリとの親和性

3. **NOT EXISTSの活用**：

- 否定条件の表現
- データの欠損や未達成条件の特定
- 完全性チェックの実装

4. **複雑な存在確認パターン**：

- 複数条件の組み合わせ
- 階層的な存在確認
- 条件付き存在確認

5. **パフォーマンス最適化**：

- インデックスの重要性
- 効率的な条件配置
- 不要な結合の回避

6. **他の手法との使い分け**：

- EXISTS vs JOIN の適切な選択
- EXISTS vs IN の使い分け基準

EXISTS演算子は、データの存在確認において非常に強力で安全な方法を提供します。特に、NULL値の扱いやパフォーマンスの観点から、多くの場面でINよりも適切な選択となります。適切に使用することで、複雑な条件でのデータ検索や関連性の確認が効率的に行えるようになります。

次の章では、「CASE式：条件分岐による値の変換」について学び、SQLでの条件付きロジックの実装方法を深く理解していきます。

---

23. CASE式：条件分岐による値の変換

はじめに

これまでの章で、サブクエリや EXISTS演算子など、SQLの高度な検索技術について学んできました。しかし、実際のデータ分析では、取得したデータを条件に応じて変換・分類する必要があることが多くあります。

例えば以下のような場合です：

- 「点数を文字等級（A、B、C、D、F）に変換したい」
- 「出席状況を『良好』『要注意』『問題あり』に分類したい」
- 「時間帯によって授業を『午前』『午後』『夜間』に分類したい」
- 「条件に応じて異なる計算を行いたい」

このような条件分岐による値の変換を実現するのが「CASE式」です。CASE式は、SQLにおけるプログラミング言語の「if-then-else」文に相当する機能で、条件に応じて異なる値を返すことができます。

この章では、CASE式の基本概念から高度な活用方法まで、実践的な例を通して詳しく学びます。

CASE式とは

CASE式は、複数の条件を評価し、条件に応じて異なる値を返すSQL構文です。プログラミング言語の条件分岐（if-then-else）と同様の機能を提供します。

> **用語解説**：
>
> - **CASE式**：条件に応じて異なる値を返すSQL構文で、条件分岐を実現します。
> - **単純CASE式**：特定のカラムの値と複数の値を比較するCASE式です。
> - **検索CASE式**：複雑な条件式を評価できるCASE式です。
> - **WHEN句**：「～の場合」という条件を指定する部分です。
> - **THEN句**：条件が真の場合に返される値を指定する部分です。
> - **ELSE句**：すべての条件が偽の場合に返される値を指定する部分です（省略可能）。

CASE式の基本構文

CASE式には2つの形式があります：

1. 単純CASE式（Simple CASE）

```
CASE カラム名
    WHEN 値1 THEN 結果1
    WHEN 値2 THEN 結果2
    WHEN 値3 THEN 結果3
    ELSE デフォルト値
END
```

2. 検索CASE式（Searched CASE）

```
CASE
    WHEN 条件1 THEN 結果1
    WHEN 条件2 THEN 結果2
    WHEN 条件3 THEN 結果3
    ELSE デフォルト値
END
```

検索CASE式の方がより柔軟で一般的に使用されます。

CASE式の基本例

例1：成績の等級変換（検索CASE式）

点数を文字等級に変換してみましょう：

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.score AS 点数,
    CASE
        WHEN g.score >= 90 THEN 'A'
        WHEN g.score >= 80 THEN 'B'
        WHEN g.score >= 70 THEN 'C'
        WHEN g.score >= 60 THEN 'D'
        ELSE 'F'
    END AS 等級
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE g.grade_type = '中間テスト'
ORDER BY g.score DESC;
```

実行結果：

学生名	講座名	点数	等級
鈴木健太	ITのための基礎知識	95.0	A
松本さくら	ITのための基礎知識	93.5	A
新垣愛留	ITのための基礎知識	92.0	A
永田悦子	ITのための基礎知識	91.0	A
河田咲奈	ITのための基礎知識	88.0	B
黒沢春馬	ITのための基礎知識	85.5	B
…	…	…	…

例2：出席状況の分類（単純CASE式）

出席状況を日本語に変換：

```
SELECT
    s.student_name AS 学生名,
    cs.schedule_date AS 日付,
    CASE a.status
        WHEN 'present' THEN '出席'
        WHEN 'absent' THEN '欠席'
        WHEN 'late' THEN '遅刻'
        ELSE '不明'
    END AS 出席状況
FROM attendance a
JOIN students s ON a.student_id = s.student_id
JOIN course_schedule cs ON a.schedule_id = cs.schedule_id
```

```
WHERE cs.schedule_date = '2025-05-20'
ORDER BY s.student_name;
```

実行結果：

学生名	日付	出席状況
黒沢春馬	2025-05-20	出席
新垣愛留	2025-05-20	遅刻
柴崎春花	2025-05-20	出席
森下風凛	2025-05-20	欠席
…	…	…

SELECT句でのCASE式の活用

例3：複雑な条件による分類

学生の学習状況を総合的に評価：

```
SELECT
    s.student_id,
    s.student_name AS 学生名,
    AVG(g.score) AS 平均点,
    COUNT(DISTINCT sc.course_id) AS 受講講座数,
    AVG(CASE WHEN a.status = 'present' THEN 1.0 ELSE 0.0 END) * 100 AS 出席率,
    CASE
        WHEN AVG(g.score) >= 85 AND
             AVG(CASE WHEN a.status = 'present' THEN 1.0 ELSE 0.0 END) >= 0.9 THEN
'優秀'
        WHEN AVG(g.score) >= 75 AND
             AVG(CASE WHEN a.status = 'present' THEN 1.0 ELSE 0.0 END) >= 0.8 THEN
'良好'
        WHEN AVG(g.score) >= 65 OR
             AVG(CASE WHEN a.status = 'present' THEN 1.0 ELSE 0.0 END) >= 0.7 THEN
'普通'
        ELSE '要指導'
    END AS 総合評価
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id
LEFT JOIN student_courses sc ON s.student_id = sc.student_id
LEFT JOIN attendance a ON s.student_id = a.student_id
WHERE s.student_id BETWEEN 301 AND 310
GROUP BY s.student_id, s.student_name
ORDER BY AVG(g.score) DESC;
```

実行結果：

student_id	学生名	平均点	受講講座数	出席率	総合評価
311	鈴木健太	89.8	6	92.5	優秀
302	新垣愛留	86.5	7	88.9	良好
308	永田悦子	85.9	5	95.0	優秀
301	黒沢春馬	82.3	8	85.7	良好
...	...	...	...	...	...

例4：時間帯による授業の分類

```
SELECT
    c.course_name AS 講座名,
    cp.start_time AS 開始時間,
    CASE
        WHEN cp.start_time < '12:00:00' THEN '午前'
        WHEN cp.start_time < '17:00:00' THEN '午後'
        ELSE '夜間'
    END AS 時間帯,
    COUNT(*) AS 授業回数
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
JOIN class_periods cp ON cs.period_id = cp.period_id
GROUP BY c.course_name, cp.start_time
ORDER BY c.course_name, cp.start_time;
```

WHERE句でのCASE式

CASE式はWHERE句でも使用できます。これにより、複雑な条件を分かりやすく表現できます。

例5：条件付きフィルタリング

成績のタイプに応じて異なる合格基準を適用：

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.grade_type AS 評価タイプ,
    g.score AS 点数,
    CASE g.grade_type
        WHEN '中間テスト' THEN 70
        WHEN 'レポート1' THEN 60
        WHEN '最終評価' THEN 75
        ELSE 65
    END AS 合格基準
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN courses c ON g.course_id = c.course_id
```

```
WHERE g.score >= CASE g.grade_type
                    WHEN '中間テスト' THEN 70
                    WHEN 'レポート1' THEN 60
                    WHEN '最終評価' THEN 75
                    ELSE 65
                END
ORDER BY s.student_name, c.course_name;
```

ORDER BY句でのCASE式

ORDER BY句でCASE式を使用することで、カスタムソート順を実現できます。

例6：カスタムソート順

出席状況を優先度順でソート：

```
SELECT
    s.student_name AS 学生名,
    cs.schedule_date AS 日付,
    CASE a.status
        WHEN 'present' THEN '出席'
        WHEN 'late' THEN '遅刻'
        WHEN 'absent' THEN '欠席'
        ELSE '不明'
    END AS 出席状況
FROM attendance a
JOIN students s ON a.student_id = s.student_id
JOIN course_schedule cs ON a.schedule_id = cs.schedule_id
WHERE cs.schedule_date = '2025-05-20'
ORDER BY
    CASE a.status
        WHEN 'absent' THEN 1   -- 欠席を最初に
        WHEN 'late' THEN 2     -- 遅刻を次に
        WHEN 'present' THEN 3  -- 出席を最後に
        ELSE 4
    END,
    s.student_name;
```

例7：成績タイプのカスタムソート

```
SELECT
    s.student_name AS 学生名,
    g.grade_type AS 評価タイプ,
    g.score AS 点数
FROM grades g
JOIN students s ON g.student_id = s.student_id
WHERE s.student_id = 301
ORDER BY
    CASE g.grade_type
```

```
        WHEN '中間テスト' THEN 1
        WHEN 'レポート1' THEN 2
        WHEN '課題1' THEN 3
        WHEN '最終評価' THEN 4
        ELSE 5
    END;
```

集計関数とCASE式の組み合わせ

CASE式を集計関数と組み合わせることで、条件付き集計が可能になります。

例8：条件付きカウント

各講座の出席状況別人数を集計：

```
SELECT
    c.course_name AS 講座名,
    cs.schedule_date AS 日付,
    COUNT(*) AS 受講者数,
    SUM(CASE WHEN a.status = 'present' THEN 1 ELSE 0 END) AS 出席者数,
    SUM(CASE WHEN a.status = 'late' THEN 1 ELSE 0 END) AS 遅刻者数,
    SUM(CASE WHEN a.status = 'absent' THEN 1 ELSE 0 END) AS 欠席者数,
    ROUND(SUM(CASE WHEN a.status = 'present' THEN 1 ELSE 0 END) * 100.0 /
COUNT(*), 1) AS 出席率
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id
WHERE cs.schedule_date = '2025-05-20'
GROUP BY c.course_name, cs.schedule_date
ORDER BY 出席率 DESC;
```

実行結果：

講座名	日付	受講者数	出席者数	遅刻者数	欠席者数	出席率
データベース設計と実装	2025-05-20	8	7	1	0	87.5
ITのための基礎知識	2025-05-20	12	9	2	1	75.0
AI・機械学習入門	2025-05-20	10	7	1	2	70.0
…	…	…	…	…	…	…

例9：成績レベル別の統計

```
SELECT
    c.course_name AS 講座名,
    COUNT(*) AS 総受験者数,
    SUM(CASE WHEN g.score >= 90 THEN 1 ELSE 0 END) AS A評価数,
    SUM(CASE WHEN g.score >= 80 AND g.score < 90 THEN 1 ELSE 0 END) AS B評価数,
```

```
    SUM(CASE WHEN g.score >= 70 AND g.score < 80 THEN 1 ELSE 0 END) AS C評価数,
    SUM(CASE WHEN g.score >= 60 AND g.score < 70 THEN 1 ELSE 0 END) AS D評価数,
    SUM(CASE WHEN g.score < 60 THEN 1 ELSE 0 END) AS F評価数,
    ROUND(AVG(g.score), 1) AS 平均点
FROM grades g
JOIN courses c ON g.course_id = c.course_id
WHERE g.grade_type = '中間テスト'
GROUP BY c.course_name
ORDER BY 平均点 DESC;
```

実行結果：

講座名	総受験者数	A評価数	B評価数	C評価数	D評価数	F評価数	平均点
ITのための基礎知識	12	4	5	2	1	0	86.2
データベース設計と実装	7	2	3	2	0	0	82.3
AI・機械学習入門	8	2	2	3	1	0	78.9
…	…	…	…	…	…	…	…

CASE式のネスト（入れ子）

CASE式の中に別のCASE式を含めることで、より複雑な条件分岐を表現できます。

例10：複雑な成績評価システム

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.score AS 点数,
    CASE g.grade_type
        WHEN '中間テスト' THEN
            CASE
                WHEN g.score >= 95 THEN 'A+'
                WHEN g.score >= 90 THEN 'A'
                WHEN g.score >= 85 THEN 'A-'
                WHEN g.score >= 80 THEN 'B+'
                WHEN g.score >= 75 THEN 'B'
                WHEN g.score >= 70 THEN 'B-'
                WHEN g.score >= 65 THEN 'C+'
                WHEN g.score >= 60 THEN 'C'
                ELSE 'F'
            END
        WHEN 'レポート1' THEN
            CASE
                WHEN g.score >= 90 THEN 'A'
                WHEN g.score >= 80 THEN 'B'
                WHEN g.score >= 70 THEN 'C'
```

```
                WHEN g.score >= 60 THEN 'D'
                ELSE 'F'
            END
        ELSE
            CASE
                WHEN g.score >= 85 THEN 'A'
                WHEN g.score >= 75 THEN 'B'
                WHEN g.score >= 65 THEN 'C'
                WHEN g.score >= 55 THEN 'D'
                ELSE 'F'
            END
    END AS 等級
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN courses c ON g.course_id = c.course_id
ORDER BY s.student_name, c.course_name, g.grade_type;
```

NULL値の処理

CASE式では、NULL値を適切に処理することが重要です。

例11：NULL値を含む条件分岐

```
SELECT
    s.student_name AS 学生名,
    sc.course_id AS 講座ID,
    CASE
        WHEN g.score IS NULL THEN '未提出'
        WHEN g.score >= 80 THEN '合格'
        WHEN g.score >= 60 THEN '条件付き合格'
        ELSE '不合格'
    END AS 結果,
    COALESCE(g.score, 0) AS 点数
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
LEFT JOIN grades g ON s.student_id = g.student_id
                  AND sc.course_id = g.course_id
                  AND g.grade_type = '中間テスト'
WHERE s.student_id BETWEEN 301 AND 305
ORDER BY s.student_name, sc.course_id;
```

CASE式の実践的な応用例

例12：学習進捗ダッシュボード

```
SELECT
    s.student_id,
    s.student_name AS 学生名,
    COUNT(DISTINCT sc.course_id) AS 受講講座数,
```

```
    CASE
        WHEN COUNT(DISTINCT sc.course_id) >= 8 THEN '多い'
        WHEN COUNT(DISTINCT sc.course_id) >= 5 THEN '適正'
        WHEN COUNT(DISTINCT sc.course_id) >= 3 THEN '少ない'
        ELSE '非常に少ない'
    END AS 履修状況,
    ROUND(AVG(CASE WHEN g.score IS NOT NULL THEN g.score END), 1) AS 平均点,
    CASE
        WHEN AVG(CASE WHEN g.score IS NOT NULL THEN g.score END) IS NULL THEN '未
評価'
        WHEN AVG(CASE WHEN g.score IS NOT NULL THEN g.score END) >= 85 THEN '優秀'
        WHEN AVG(CASE WHEN g.score IS NOT NULL THEN g.score END) >= 75 THEN '良好'
        WHEN AVG(CASE WHEN g.score IS NOT NULL THEN g.score END) >= 65 THEN '普通'
        ELSE '要改善'
    END AS 成績評価,
    ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS 出席率,
    CASE
        WHEN AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END) >= 90 THEN
'良好'
        WHEN AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END) >= 80 THEN
'普通'
        WHEN AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END) >= 70 THEN
'要注意'
        ELSE '問題あり'
    END AS 出席評価
FROM students s
LEFT JOIN student_courses sc ON s.student_id = sc.student_id
LEFT JOIN grades g ON s.student_id = g.student_id
LEFT JOIN attendance a ON s.student_id = a.student_id
WHERE s.student_id BETWEEN 301 AND 310
GROUP BY s.student_id, s.student_name
ORDER BY s.student_id;
```

CASE式のパフォーマンス考慮点

CASE式を効率的に使用するためのポイント：

1. **条件の順序**：最も頻繁に該当する条件を最初に配置することで、評価回数を減らせます。
2. **複雑な条件の分解**：非常に複雑なCASE式は、複数のステップに分解したり、一時テーブルを使用したりすることを検討しましょう。
3. **インデックスの活用**：CASE式で使用するカラムにインデックスがある場合、WHERE句でそのカラムを直接使用することも検討しましょう。

パフォーマンス改善の例：

```
-- 効率的なCASE式の例（頻度の高い条件を先に）
SELECT
    student_name,
    CASE
```

```
        WHEN score >= 70 THEN '合格'       -- 最も頻度が高い
        WHEN score >= 60 THEN '条件付き'    -- 次に頻度が高い
        WHEN score IS NULL THEN '未受験'    -- まれなケース
        ELSE '不合格'                      -- 最も少ない
    END AS 結果
FROM students s
JOIN grades g ON s.student_id = g.student_id;
```

練習問題

問題23-1

CASE式を使用して、講座の受講者数を「多い」（15人以上）、「普通」（10〜14人）、「少ない」（5〜9人）、「非常に少ない」（4人以下）に分類するSQLを書いてください。結果には講座ID、講座名、受講者数、分類を含めてください。

問題23-2

CASE式を使用して、各学生の出席率を計算し、「優秀」（90%以上）、「良好」（80%以上）、「普通」（70%以上）、「要改善」（70%未満）に分類するSQLを書いてください。出席率も合わせて表示してください。

問題23-3

CASE式を使用して、教師の担当負荷を「重い」（4講座以上）、「適正」（2〜3講座）、「軽い」（1講座）、「なし」（担当なし）に分類するSQLを書いてください。結果には教師ID、教師名、担当講座数、負荷分類を含めてください。

問題23-4

CASE式と集計関数を組み合わせて、各講座について成績分布を分析するSQLを書いてください。「90点以上」「80-89点」「70-79点」「60-69点」「60点未満」「未提出」の各カテゴリの人数を表示してください。

問題23-5

CASE式を使用して、学生の学習パフォーマンスを総合評価するSQLを書いてください。平均点と出席率の両方を考慮し、以下の基準で評価してください：

- 平均点85点以上かつ出席率90%以上：「S」
- 平均点75点以上かつ出席率80%以上：「A」
- 平均点65点以上かつ出席率70%以上：「B」
- それ以外：「C」

問題23-6

CASE式を使用して、時間割を見やすく整理するSQLを書いてください。授業時間を「1限目」「2限目」...として表示し、教室を建物別（1号館、2号館など）に分類して、曜日別に整理してください。2025年5月21日の授業スケジュールを対象とします。

解答

解答23-1

```
SELECT
    c.course_id,
    c.course_name AS 講座名,
    COUNT(sc.student_id) AS 受講者数,
    CASE
        WHEN COUNT(sc.student_id) >= 15 THEN '多い'
        WHEN COUNT(sc.student_id) >= 10 THEN '普通'
        WHEN COUNT(sc.student_id) >= 5 THEN '少ない'
        ELSE '非常に少ない'
    END AS 分類
FROM courses c
LEFT JOIN student_courses sc ON c.course_id = sc.course_id
GROUP BY c.course_id, c.course_name
ORDER BY 受講者数 DESC;
```

解答23-2

```
SELECT
    s.student_id,
    s.student_name AS 学生名,
    ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS 出席率,
    CASE
        WHEN AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END) >= 90 THEN
'優秀'
        WHEN AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END) >= 80 THEN
'良好'
        WHEN AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END) >= 70 THEN
'普通'
        ELSE '要改善'
    END AS 出席評価
FROM students s
LEFT JOIN attendance a ON s.student_id = a.student_id
GROUP BY s.student_id, s.student_name
HAVING COUNT(a.student_id) > 0  -- 出席記録がある学生のみ
ORDER BY 出席率 DESC;
```

解答23-3

```
SELECT
    t.teacher_id,
    t.teacher_name AS 教師名,
    COUNT(c.course_id) AS 担当講座数,
    CASE
```

```
        WHEN COUNT(c.course_id) >= 4 THEN '重い'
        WHEN COUNT(c.course_id) >= 2 THEN '適正'
        WHEN COUNT(c.course_id) = 1 THEN '軽い'
        ELSE 'なし'
    END AS 負荷分類
FROM teachers t
LEFT JOIN courses c ON t.teacher_id = c.teacher_id
GROUP BY t.teacher_id, t.teacher_name
ORDER BY 担当講座数 DESC;
```

解答23-4

```
SELECT
    c.course_id,
    c.course_name AS 講座名,
    SUM(CASE WHEN g.score >= 90 THEN 1 ELSE 0 END) AS '90点以上',
    SUM(CASE WHEN g.score >= 80 AND g.score < 90 THEN 1 ELSE 0 END) AS '80-89点',
    SUM(CASE WHEN g.score >= 70 AND g.score < 80 THEN 1 ELSE 0 END) AS '70-79点',
    SUM(CASE WHEN g.score >= 60 AND g.score < 70 THEN 1 ELSE 0 END) AS '60-69点',
    SUM(CASE WHEN g.score < 60 THEN 1 ELSE 0 END) AS '60点未満',
    (SELECT COUNT(*) FROM student_courses sc WHERE sc.course_id = c.course_id) -
    COUNT(g.score) AS 未提出
FROM courses c
LEFT JOIN grades g ON c.course_id = g.course_id AND g.grade_type = '中間テスト'
GROUP BY c.course_id, c.course_name
ORDER BY c.course_id;
```

解答23-5

```
SELECT
    s.student_id,
    s.student_name AS 学生名,
    ROUND(AVG(g.score), 1) AS 平均点,
    ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS 出席率,
    CASE
        WHEN AVG(g.score) >= 85 AND AVG(CASE WHEN a.status = 'present' THEN 100.0
ELSE 0 END) >= 90 THEN 'S'
        WHEN AVG(g.score) >= 75 AND AVG(CASE WHEN a.status = 'present' THEN 100.0
ELSE 0 END) >= 80 THEN 'A'
        WHEN AVG(g.score) >= 65 AND AVG(CASE WHEN a.status = 'present' THEN 100.0
ELSE 0 END) >= 70 THEN 'B'
        ELSE 'C'
    END AS 総合評価
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id
LEFT JOIN attendance a ON s.student_id = a.student_id
GROUP BY s.student_id, s.student_name
HAVING COUNT(DISTINCT g.grade_id) > 0 AND COUNT(DISTINCT a.schedule_id) > 0
ORDER BY 総合評価, 平均点 DESC;
```

解答23-6

```
SELECT
    CASE cp.period_id
        WHEN 1 THEN '1限目'
        WHEN 2 THEN '2限目'
        WHEN 3 THEN '3限目'
        WHEN 4 THEN '4限目'
        WHEN 5 THEN '5限目'
        ELSE CONCAT(cp.period_id, '限目')
    END AS 時限,
    cp.start_time AS 開始時間,
    cp.end_time AS 終了時間,
    c.course_name AS 講座名,
    cl.classroom_name AS 教室名,
    CASE
        WHEN cl.building LIKE '1号館%' THEN '1号館'
        WHEN cl.building LIKE '2号館%' THEN '2号館'
        WHEN cl.building LIKE '3号館%' THEN '3号館'
        WHEN cl.building LIKE '4号館%' THEN '4号館'
        ELSE cl.building
    END AS 建物,
    t.teacher_name AS 担当教師
FROM course_schedule cs
JOIN class_periods cp ON cs.period_id = cp.period_id
JOIN courses c ON cs.course_id = c.course_id
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
JOIN teachers t ON cs.teacher_id = t.teacher_id
WHERE cs.schedule_date = '2025-05-21'
ORDER BY cp.period_id, cl.building, cl.classroom_name;
```

まとめ

この章では、CASE式について詳しく学びました：

1. **CASE式の基本概念**：
    - 条件分岐による値の変換を実現するSQL構文
    - 単純CASE式と検索CASE式の2つの形式
    - プログラミング言語のif-then-else文に相当する機能
2. **CASE式の基本構文**：
    - WHEN-THEN-ELSE-ENDの基本構造
    - 複数条件の評価順序
    - ELSE句の省略とNULL値の扱い
3. **様々な場面でのCASE式の活用**：

- SELECT句での値の変換と分類
- WHERE句での複雑な条件指定
- ORDER BY句でのカスタムソート順

4. **集計関数との組み合わせ**：

- 条件付きカウントとSUM関数
- カテゴリ別の統計集計
- 複雑な分析レポートの作成

5. **高度なCASE式の使用法**：

- ネストしたCASE式
- NULL値の適切な処理
- 複雑な条件分岐の実装

6. **実践的な応用例**：

- 成績の等級変換システム
- 学習進捗ダッシュボード
- 出席状況の分類と分析

7. **パフォーマンスの考慮点**：

- 条件の順序の最適化
- 複雑なCASE式の分解方法
- インデックスの効果的な活用

CASE式は、データの分類、変換、条件付き処理において非常に強力なツールです。適切に使用することで、複雑なビジネスロジックをSQLで直接実装でき、レポート作成やデータ分析の効率を大幅に向上させることができます。

次の章では、「ウィンドウ関数：OVER句とパーティション」について学び、より高度な分析機能を理解していきます。

---

24. ウィンドウ関数：OVER句とパーティション

はじめに

これまでの章で、CASE式による条件分岐について学びました。SQLには、さらに高度な分析機能として「ウィンドウ関数」があります。ウィンドウ関数は、データ分析やレポート作成において非常に強力で、従来のGROUP BYでは実現が困難な複雑な集計や順位付けを可能にします。

ウィンドウ関数が活躍する場面の例：

- 「各学生の成績順位を求めたい」
- 「講座ごとの成績ランキングを作成したい」
- 「前回のテスト結果と今回の結果を比較したい」
- 「各月の累積出席者数を計算したい」

- 「移動平均を求めたい」

通常の集計関数（SUM、AVG、COUNT等）では、GROUP BYを使用すると元の行数が減ってしまいますが、ウィンドウ関数では元の行構造を保ったまま集計結果を取得できます。

この章では、ウィンドウ関数の基本概念から実践的な活用方法まで、詳しく学んでいきます。

ウィンドウ関数とは

ウィンドウ関数は、指定された「ウィンドウ」（行の範囲）に対して計算を行う関数です。通常の集計関数と異なり、結果セットの行数を変更せず、各行に対して集計値や順位などの情報を追加できます。

> **用語解説**：
>
> - **ウィンドウ関数**：指定された行の範囲（ウィンドウ）に対して計算を行い、元の行構造を保ったまま結果を返す関数です。
> - **OVER句**：ウィンドウ関数でウィンドウの範囲や順序を指定する句です。
> - **パーティション**：データをグループに分割する仕組みで、各グループ内でウィンドウ関数が計算されます。
> - **ウィンドウフレーム**：各行において、計算対象となる行の範囲を指定します。
> - **順位関数**：ROW_NUMBER()、RANK()、DENSE_RANK()など、行に順位を付ける関数です。
> - **分析関数**：LAG()、LEAD()、FIRST_VALUE()、LAST_VALUE()など、行間の比較や分析を行う関数です。

ウィンドウ関数の基本構文

```
関数名() OVER (
    [PARTITION BY カラム1, カラム2, ...]
    [ORDER BY カラム1 [ASC|DESC], カラム2 [ASC|DESC], ...]
    [フレーム指定]
)
```

OVER句の構成要素

1. **PARTITION BY**：データをグループに分割（省略可能）
2. **ORDER BY**：ウィンドウ内での行の順序を指定（省略可能）
3. **フレーム指定**：計算対象の行範囲を指定（省略可能）

順位関数（Ranking Functions）

ROW_NUMBER()

ROW_NUMBER()は、ウィンドウ内で各行に連続した番号を割り当てます。

例1：全体での成績順位

```
SELECT
    s.student_name AS 学生名,
```

```
    c.course_name AS 講座名,
    g.score AS 点数,
    ROW_NUMBER() OVER (ORDER BY g.score DESC) AS 全体順位
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE g.grade_type = '中間テスト'
ORDER BY g.score DESC;
```

実行結果：

学生名	講座名	点数	全体順位
鈴木健太	ITのための基礎知識	95.0	1
松本さくら	ITのための基礎知識	93.5	2
新垣愛留	ITのための基礎知識	92.0	3
永田悦子	ITのための基礎知識	91.0	4
中村彩香	AI・機械学習入門	89.5	5
…	…	…	…

例2：講座別での成績順位（PARTITION BY）

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.score AS 点数,
    ROW_NUMBER() OVER (PARTITION BY c.course_id ORDER BY g.score DESC) AS 講座内順
位
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE g.grade_type = '中間テスト'
ORDER BY c.course_name, g.score DESC;
```

実行結果：

学生名	講座名	点数	講座内順位
新垣愛留	AI・機械学習入門	91.0	1
中村彩香	AI・機械学習入門	89.5	2
吉川伽羅	AI・機械学習入門	82.5	3
…	…	…	…
鈴木健太	ITのための基礎知識	95.0	1

学生名	講座名	点数	講座内順位
松本さくら	ITのための基礎知識	93.5	2
新垣愛留	ITのための基礎知識	92.0	3
…	…	…	…

PARTITION BYを使用することで、各講座内での順位を個別に計算できます。

RANK()とDENSE_RANK()

- **RANK()**：同順位がある場合、次の順位をスキップします
- **DENSE_RANK()**：同順位がある場合でも、次の順位を連続させます

例3：順位関数の比較

```
SELECT
    s.student_name AS 学生名,
    g.score AS 点数,
    ROW_NUMBER() OVER (ORDER BY g.score DESC) AS ROW_NUMBER,
    RANK() OVER (ORDER BY g.score DESC) AS RANK,
    DENSE_RANK() OVER (ORDER BY g.score DESC) AS DENSE_RANK
FROM grades g
JOIN students s ON g.student_id = s.student_id
WHERE g.grade_type = '中間テスト' AND g.course_id = '1'
ORDER BY g.score DESC;
```

実行結果：

学生名	点数	ROW_NUMBER	RANK	DENSE_RANK
鈴木健太	95.0	1	1	1
松本さくら	93.5	2	2	2
新垣愛留	92.0	3	3	3
永田悦子	91.0	4	4	4
河田咲奈	88.0	5	5	5
河田咲奈	88.0	6	5	5
黒沢春馬	85.5	7	7	6
…	…	…	…	…

この例では、河田咲奈が同点の88.0点を取った場合の違いを示しています：

- ROW_NUMBER()：連続した番号（5, 6）
- RANK()：同順位で次をスキップ（5, 5, 7）
- DENSE_RANK()：同順位で次を連続（5, 5, 6）

分析関数（Analytic Functions）

LAG()とLEAD()

LAG()は前の行の値を、LEAD()は次の行の値を取得します。

例4：前回テストとの点数比較

```
SELECT
    s.student_name AS 学生名,
    g.grade_type AS テスト種別,
    g.score AS 今回点数,
    LAG(g.score) OVER (PARTITION BY s.student_id ORDER BY
        CASE g.grade_type
            WHEN '中間テスト' THEN 1
            WHEN 'レポート1' THEN 2
            WHEN '最終評価' THEN 3
        END) AS 前回点数,
    g.score - LAG(g.score) OVER (PARTITION BY s.student_id ORDER BY
        CASE g.grade_type
            WHEN '中間テスト' THEN 1
            WHEN 'レポート1' THEN 2
            WHEN '最終評価' THEN 3
        END) AS 点数変化
FROM grades g
JOIN students s ON g.student_id = s.student_id
WHERE s.student_id = 301 AND g.course_id = '1'
ORDER BY CASE g.grade_type
    WHEN '中間テスト' THEN 1
    WHEN 'レポート1' THEN 2
    WHEN '最終評価' THEN 3
END;
```

実行結果：

学生名	テスト種別	今回点数	前回点数	点数変化
黒沢春馬	中間テスト	85.5	NULL	NULL
黒沢春馬	レポート1	85.0	85.5	-0.5
黒沢春馬	最終評価	87.0	85.0	2.0

例5：月別の出席者数推移

```
SELECT
    DATE_FORMAT(cs.schedule_date, '%Y-%m') AS 年月,
    COUNT(CASE WHEN a.status = 'present' THEN 1 END) AS 今月出席者数,
    LAG(COUNT(CASE WHEN a.status = 'present' THEN 1 END))
        OVER (ORDER BY DATE_FORMAT(cs.schedule_date, '%Y-%m')) AS 前月出席者数,
```

```
    COUNT(CASE WHEN a.status = 'present' THEN 1 END) -
    LAG(COUNT(CASE WHEN a.status = 'present' THEN 1 END))
        OVER (ORDER BY DATE_FORMAT(cs.schedule_date, '%Y-%m')) AS 増減
FROM course_schedule cs
LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id
GROUP BY DATE_FORMAT(cs.schedule_date, '%Y-%m')
ORDER BY 年月;
```

集計ウィンドウ関数

通常の集計関数（SUM、AVG、COUNT等）もウィンドウ関数として使用できます。

例6：累積成績と移動平均

```
SELECT
    s.student_name AS 学生名,
    g.grade_type AS 評価種別,
    g.score AS 点数,
    AVG(g.score) OVER (PARTITION BY s.student_id
                       ORDER BY CASE g.grade_type
                           WHEN '中間テスト' THEN 1
                           WHEN 'レポート1' THEN 2
                           WHEN '最終評価' THEN 3
                       END) AS 累積平均,
    SUM(g.score) OVER (PARTITION BY s.student_id
                       ORDER BY CASE g.grade_type
                           WHEN '中間テスト' THEN 1
                           WHEN 'レポート1' THEN 2
                           WHEN '最終評価' THEN 3
                       END) AS 累積合計
FROM grades g
JOIN students s ON g.student_id = s.student_id
WHERE s.student_id = 302 AND g.course_id = '1'
ORDER BY CASE g.grade_type
    WHEN '中間テスト' THEN 1
    WHEN 'レポート1' THEN 2
    WHEN '最終評価' THEN 3
END;
```

実行結果：

学生名	評価種別	点数	累積平均	累積合計
新垣愛留	中間テスト	92.0	92.0	92.0
新垣愛留	レポート1	88.0	90.0	180.0
新垣愛留	最終評価	90.0	90.0	270.0

例7：各学生の講座別平均との比較

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.score AS 個人点数,
    ROUND(AVG(g.score) OVER (PARTITION BY g.course_id), 1) AS 講座平均,
    ROUND(g.score - AVG(g.score) OVER (PARTITION BY g.course_id), 1) AS 平均との差,
    ROUND(AVG(g.score) OVER (), 1) AS 全体平均
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE g.grade_type = '中間テスト' AND s.student_id BETWEEN 301 AND 305
ORDER BY c.course_name, g.score DESC;
```

フレーム指定（Window Frames）

フレーム指定により、計算対象となる行の範囲をより詳細に制御できます。

基本構文

```
ROWS BETWEEN 開始位置 AND 終了位置
```

フレーム境界の指定方法

- `UNBOUNDED PRECEDING`：ウィンドウの最初
- `CURRENT ROW`：現在の行
- `UNBOUNDED FOLLOWING`：ウィンドウの最後
- `n PRECEDING`：現在行からn行前
- `n FOLLOWING`：現在行からn行後

例8：移動平均の計算

```
SELECT
    cs.schedule_date AS 日付,
    COUNT(CASE WHEN a.status = 'present' THEN 1 END) AS 出席者数,
    ROUND(AVG(COUNT(CASE WHEN a.status = 'present' THEN 1 END))
        OVER (ORDER BY cs.schedule_date
              ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 1) AS 3日移動平均
FROM course_schedule cs
LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id
GROUP BY cs.schedule_date
ORDER BY cs.schedule_date;
```

この例では、過去3日間（2 PRECEDING AND CURRENT ROW）の移動平均を計算しています。

例9：累積出席率の計算

```
SELECT
    s.student_name AS 学生名,
    cs.schedule_date AS 日付,
    CASE WHEN a.status = 'present' THEN 1 ELSE 0 END AS 出席フラグ,
    ROUND(AVG(CASE WHEN a.status = 'present' THEN 1.0 ELSE 0 END)
        OVER (PARTITION BY s.student_id
              ORDER BY cs.schedule_date
              ROWS UNBOUNDED PRECEDING) * 100, 1) AS 累積出席率
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN course_schedule cs ON sc.course_id = cs.course_id
LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id AND s.student_id =
a.student_id
WHERE s.student_id = 301
ORDER BY cs.schedule_date;
```

FIRST_VALUE()とLAST_VALUE()

ウィンドウ内の最初や最後の値を取得できます。

例10：各講座での最高点・最低点との比較

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.score AS 点数,
    FIRST_VALUE(g.score) OVER (PARTITION BY g.course_id
                               ORDER BY g.score DESC
                               ROWS UNBOUNDED PRECEDING) AS 最高点,
    LAST_VALUE(g.score) OVER (PARTITION BY g.course_id
                              ORDER BY g.score DESC
                              ROWS BETWEEN UNBOUNDED PRECEDING
                              AND UNBOUNDED FOLLOWING) AS 最低点,
    g.score - LAST_VALUE(g.score) OVER (PARTITION BY g.course_id
                                        ORDER BY g.score DESC
                                        ROWS BETWEEN UNBOUNDED PRECEDING
                                        AND UNBOUNDED FOLLOWING) AS 最低点との差
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE g.grade_type = '中間テスト' AND g.course_id = '1'
ORDER BY g.score DESC;
```

ウィンドウ関数の実践的な応用例

例11：学生の成績改善状況分析

```
WITH student_progress AS (
    SELECT
        s.student_id,
        s.student_name,
        g.course_id,
        c.course_name,
        g.grade_type,
        g.score,
        ROW_NUMBER() OVER (PARTITION BY s.student_id, g.course_id
                           ORDER BY CASE g.grade_type
                               WHEN '中間テスト' THEN 1
                               WHEN 'レポート1' THEN 2
                               WHEN '最終評価' THEN 3
                           END) AS test_order,
        LAG(g.score) OVER (PARTITION BY s.student_id, g.course_id
                           ORDER BY CASE g.grade_type
                               WHEN '中間テスト' THEN 1
                               WHEN 'レポート1' THEN 2
                               WHEN '最終評価' THEN 3
                           END) AS prev_score
    FROM grades g
    JOIN students s ON g.student_id = s.student_id
    JOIN courses c ON g.course_id = c.course_id
)
SELECT
    student_name AS 学生名,
    course_name AS 講座名,
    grade_type AS 評価種別,
    score AS 今回点数,
    prev_score AS 前回点数,
    score - prev_score AS 点数変化,
    CASE
        WHEN score - prev_score > 10 THEN '大幅改善'
        WHEN score - prev_score > 5 THEN '改善'
        WHEN score - prev_score > -5 THEN '維持'
        WHEN score - prev_score > -10 THEN '低下'
        ELSE '大幅低下'
    END AS 改善状況
FROM student_progress
WHERE prev_score IS NOT NULL
ORDER BY student_name, course_name, test_order;
```

例12：講座の人気度ランキング

```
SELECT
    c.course_id,
    c.course_name AS 講座名,
    t.teacher_name AS 担当教師,
    COUNT(sc.student_id) AS 受講者数,
    RANK() OVER (ORDER BY COUNT(sc.student_id) DESC) AS 人気ランキング,
```

```
    ROUND(AVG(g.score), 1) AS 平均点,
    RANK() OVER (ORDER BY AVG(g.score) DESC) AS 成績ランキング,
    ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS 出席率,
    RANK() OVER (ORDER BY AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0
END) DESC) AS 出席率ランキング
FROM courses c
JOIN teachers t ON c.teacher_id = t.teacher_id
LEFT JOIN student_courses sc ON c.course_id = sc.course_id
LEFT JOIN grades g ON sc.student_id = g.student_id AND sc.course_id = g.course_id
LEFT JOIN attendance a ON sc.student_id = a.student_id
GROUP BY c.course_id, c.course_name, t.teacher_name
ORDER BY 人気ランキング;
```

パフォーマンス考慮点

ウィンドウ関数を効率的に使用するためのポイント：

1. **インデックスの活用**：PARTITION BYやORDER BYで使用するカラムにインデックスを設定
2. **適切なフレーム指定**：必要以上に大きなフレームを指定しない
3. **メモリ使用量**：大きなデータセットでは、ウィンドウ関数がメモリを多く消費する可能性
4. **クエリの最適化**：WHERE句で事前にデータを絞り込む

パフォーマンス改善の例

```
-- 効率的なウィンドウ関数の使用例
SELECT
    s.student_name,
    g.score,
    RANK() OVER (PARTITION BY g.course_id ORDER BY g.score DESC) AS course_rank
FROM grades g
JOIN students s ON g.student_id = s.student_id
WHERE g.grade_type = '中間テスト'  -- 事前にフィルタリング
AND g.course_id IN ('1', '2', '3')  -- 必要な講座のみに限定
ORDER BY g.course_id, course_rank;
```

練習問題

問題24-1

ROW_NUMBER()を使用して、各教師が担当する講座を受講者数の多い順に順位付けするSQLを書いてください。結果には教師名、講座名、受講者数、教師内順位を含めてください。

問題24-2

LAG()関数を使用して、各学生の連続する成績評価（中間テスト→レポート1→最終評価）の点数変化を分析するSQLを書いてください。前回評価からの変化量も計算してください。

問題24-3

DENSE_RANK()を使用して、各講座内での成績上位3位までの学生を抽出するSQLを書いてください。同点の場合は同順位として扱ってください。

問題24-4

移動平均を使用して、過去3回の授業の平均出席率を計算するSQLを書いてください。授業日順に並べ、3日移動平均の出席率を表示してください。

問題24-5

SUM()のウィンドウ関数を使用して、各学生の累積出席回数と累積出席率を計算するSQLを書いてください。日付順に並べて表示してください。

問題24-6

FIRST_VALUE()とLAST_VALUE()を使用して、各講座において最高得点と最低得点を取った学生の情報を、全ての学生の行に追加するSQLを書いてください。中間テストの結果を対象とします。

解答

解答24-1

```
SELECT
    t.teacher_name AS 教師名,
    c.course_name AS 講座名,
    COUNT(sc.student_id) AS 受講者数,
    ROW_NUMBER() OVER (PARTITION BY t.teacher_id
                       ORDER BY COUNT(sc.student_id) DESC) AS 教師内順位
FROM teachers t
JOIN courses c ON t.teacher_id = c.teacher_id
LEFT JOIN student_courses sc ON c.course_id = sc.course_id
GROUP BY t.teacher_id, t.teacher_name, c.course_id, c.course_name
ORDER BY t.teacher_name, 教師内順位;
```

解答24-2

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.grade_type AS 評価種別,
    g.score AS 今回点数,
    LAG(g.score) OVER (PARTITION BY s.student_id, g.course_id
                       ORDER BY CASE g.grade_type
                           WHEN '中間テスト' THEN 1
                           WHEN 'レポート1' THEN 2
                           WHEN '最終評価' THEN 3
                       END) AS 前回点数,
    g.score - LAG(g.score) OVER (PARTITION BY s.student_id, g.course_id
                                 ORDER BY CASE g.grade_type
```

```
                                    WHEN '中間テスト' THEN 1
                                    WHEN 'レポート1' THEN 2
                                    WHEN '最終評価' THEN 3
                                END) AS 点数変化
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN courses c ON g.course_id = c.course_id
ORDER BY s.student_name, c.course_name,
         CASE g.grade_type
             WHEN '中間テスト' THEN 1
             WHEN 'レポート1' THEN 2
             WHEN '最終評価' THEN 3
         END;
```

解答24-3

```
WITH ranked_grades AS (
    SELECT
        s.student_name AS 学生名,
        c.course_name AS 講座名,
        g.score AS 点数,
        DENSE_RANK() OVER (PARTITION BY g.course_id ORDER BY g.score DESC) AS 順位
    FROM grades g
    JOIN students s ON g.student_id = s.student_id
    JOIN courses c ON g.course_id = c.course_id
    WHERE g.grade_type = '中間テスト'
)
SELECT 学生名, 講座名, 点数, 順位
FROM ranked_grades
WHERE 順位 <= 3
ORDER BY 講座名, 順位;
```

解答24-4

```
SELECT
    cs.schedule_date AS 授業日,
    c.course_name AS 講座名,
    COUNT(a.student_id) AS 対象学生数,
    SUM(CASE WHEN a.status = 'present' THEN 1 ELSE 0 END) AS 出席者数,
    ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS 出席率,
    ROUND(AVG(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END))
        OVER (PARTITION BY c.course_id
              ORDER BY cs.schedule_date
              ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 1) AS 3日移動平均出席率
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id
GROUP BY cs.schedule_id, cs.schedule_date, c.course_id, c.course_name
ORDER BY c.course_name, cs.schedule_date;
```

解答24-5

```
SELECT
    s.student_name AS 学生名,
    cs.schedule_date AS 授業日,
    CASE WHEN a.status = 'present' THEN 1 ELSE 0 END AS 今日の出席,
    SUM(CASE WHEN a.status = 'present' THEN 1 ELSE 0 END)
        OVER (PARTITION BY s.student_id
              ORDER BY cs.schedule_date
              ROWS UNBOUNDED PRECEDING) AS 累積出席回数,
    COUNT(*) OVER (PARTITION BY s.student_id
                   ORDER BY cs.schedule_date
                   ROWS UNBOUNDED PRECEDING) AS 累積授業回数,
    ROUND(SUM(CASE WHEN a.status = 'present' THEN 1 ELSE 0 END)
        OVER (PARTITION BY s.student_id
              ORDER BY cs.schedule_date
              ROWS UNBOUNDED PRECEDING) * 100.0 /
        COUNT(*) OVER (PARTITION BY s.student_id
                       ORDER BY cs.schedule_date
                       ROWS UNBOUNDED PRECEDING), 1) AS 累積出席率
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN course_schedule cs ON sc.course_id = cs.course_id
LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id AND s.student_id =
a.student_id
WHERE s.student_id BETWEEN 301 AND 305
ORDER BY s.student_name, cs.schedule_date;
```

解答24-6

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.score AS 点数,
    FIRST_VALUE(s2.student_name) OVER (PARTITION BY g.course_id
                                       ORDER BY g.score DESC
                                       ROWS UNBOUNDED PRECEDING) AS 最高得点者,
    FIRST_VALUE(g.score) OVER (PARTITION BY g.course_id
                               ORDER BY g.score DESC
                               ROWS UNBOUNDED PRECEDING) AS 最高点,
    LAST_VALUE(s2.student_name) OVER (PARTITION BY g.course_id
                                      ORDER BY g.score DESC
                                      ROWS BETWEEN UNBOUNDED PRECEDING
                                      AND UNBOUNDED FOLLOWING) AS 最低得点者,
    LAST_VALUE(g.score) OVER (PARTITION BY g.course_id
                              ORDER BY g.score DESC
                              ROWS BETWEEN UNBOUNDED PRECEDING
                              AND UNBOUNDED FOLLOWING) AS 最低点
```

```
FROM grades g
JOIN students s ON g.student_id = s.student_id
JOIN students s2 ON g.student_id = s2.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE g.grade_type = '中間テスト'
ORDER BY c.course_name, g.score DESC;
```

まとめ

この章では、ウィンドウ関数について詳しく学びました：

1. **ウィンドウ関数の基本概念**：
   - 指定されたウィンドウ（行の範囲）に対して計算を行う関数
   - 元の行構造を保ったまま集計結果を取得
   - OVER句によるウィンドウの定義

2. **OVER句の構成要素**：
   - PARTITION BY：データのグループ化
   - ORDER BY：ウィンドウ内での行の順序
   - フレーム指定：計算対象の行範囲

3. **順位関数**：
   - ROW_NUMBER()：連続した番号の割り当て
   - RANK()：同順位考慮の順位付け（次の順位をスキップ）
   - DENSE_RANK()：同順位考慮の順位付け（次の順位を連続）

4. **分析関数**：
   - LAG()とLEAD()：前後の行の値の取得
   - FIRST_VALUE()とLAST_VALUE()：ウィンドウ内の最初・最後の値

5. **集計ウィンドウ関数**：
   - SUM()、AVG()、COUNT()等の集計関数をウィンドウ関数として使用
   - 累積計算や移動平均の実現

6. **フレーム指定**：
   - ROWS BETWEENによる計算対象範囲の詳細制御
   - 移動平均や累積計算での活用

7. **実践的な応用例**：
   - 成績ランキングの作成
   - 学習進捗の分析
   - 講座人気度の評価
   - 出席率の推移分析

8. **パフォーマンス考慮点**：

- インデックスの重要性
- 適切なフレーム指定
- メモリ使用量への配慮

ウィンドウ関数は、従来のGROUP BYでは実現困難な複雑な分析を可能にする強力なツールです。データ分析、レポート作成、ランキング作成など、様々な場面で活用できる重要なSQL機能です。

次の章では、「共通テーブル式（CTE）：WITH句の活用」について学び、複雑なクエリを構造化して分かりやすく記述する方法を理解していきます。

---

25. 共通テーブル式（CTE）：WITH句の活用

はじめに

これまでの章で、ウィンドウ関数による高度な分析機能について学びました。SQLの最後の重要な機能として「共通テーブル式（CTE：Common Table Expression）」について学びます。CTEは、WITH句を使用して一時的な結果セットを定義し、クエリ内で再利用できる機能です。

CTEが特に威力を発揮する場面：

- 「複雑なサブクエリを分かりやすく構造化したい」
- 「同じサブクエリを複数回使用したい」
- 「階層データを再帰的に処理したい」
- 「複雑な計算を段階的に行いたい」
- 「クエリの可読性と保守性を向上させたい」

従来のサブクエリやFROM句内のサブクエリと比較して、CTEはより読みやすく、再利用可能で、場合によってはパフォーマンスも向上させることができます。

この章では、CTEの基本概念から再帰CTE、実践的な活用方法まで、詳しく学んでいきます。

共通テーブル式（CTE）とは

共通テーブル式（CTE）は、WITH句を使用してクエリ内で一時的な名前付きの結果セットを定義する機能です。定義したCTEは、同じクエリ内で通常のテーブルと同様に参照できます。

> **用語解説**：
>
> - **CTE（Common Table Expression）**：共通テーブル式。WITH句で定義される一時的な名前付き結果セットです。
> - **WITH句**：CTEを定義するためのSQL句で、「～と共に」という意味があります。
> - **非再帰CTE**：自分自身を参照しない通常のCTEです。
> - **再帰CTE**：自分自身を参照して反復処理を行うCTEです。
> - **アンカーメンバー**：再帰CTEにおいて、再帰の開始点となる初期データです。
> - **再帰メンバー**：再帰CTEにおいて、自分自身を参照する部分です。

CTEの基本構文

単一CTE

```
WITH CTE名 AS (
    SELECT文
)
SELECT カラム1, カラム2, ...
FROM CTE名
WHERE 条件;
```

複数CTE

```
WITH
CTE名1 AS (
    SELECT文1
),
CTE名2 AS (
    SELECT文2
)
SELECT カラム1, カラム2, ...
FROM CTE名1
JOIN CTE名2 ON 結合条件;
```

基本的なCTEの例

例1：単純なCTEの使用

成績優秀者を定義して、その詳細情報を取得：

```
WITH excellent_students AS (
    SELECT DISTINCT g.student_id
    FROM grades g
    WHERE g.score >= 90
)
SELECT
    s.student_id,
    s.student_name AS 学生名,
    COUNT(DISTINCT sc.course_id) AS 受講講座数,
    ROUND(AVG(g.score), 1) AS 平均点
FROM excellent_students es
JOIN students s ON es.student_id = s.student_id
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN grades g ON s.student_id = g.student_id
GROUP BY s.student_id, s.student_name
ORDER BY 平均点 DESC;
```

このクエリでは、まず90点以上の成績を取った学生をCTEで定義し、その後でその学生たちの詳細情報を取得しています。

実行結果：

student_id	学生名	受講講座数	平均点
311	鈴木健太	6	89.8
302	新垣愛留	7	86.5
308	永田悦子	5	85.9
320	松本さくら	4	84.2
...	...	...	...

例2：サブクエリとCTEの比較

同じ結果を得るためのサブクエリとCTEの比較：

サブクエリを使用した場合：

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.score AS 点数
FROM students s
JOIN grades g ON s.student_id = g.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE s.student_id IN (
    SELECT g2.student_id
    FROM grades g2
    GROUP BY g2.student_id
    HAVING AVG(g2.score) >= 85
)
AND g.grade_type = '中間テスト'
ORDER BY s.student_name, g.score DESC;
```

CTEを使用した場合：

```
WITH high_performers AS (
    SELECT g.student_id
    FROM grades g
    GROUP BY g.student_id
    HAVING AVG(g.score) >= 85
)
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    g.score AS 点数
FROM high_performers hp
JOIN students s ON hp.student_id = s.student_id
```

```
JOIN grades g ON s.student_id = g.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE g.grade_type = '中間テスト'
ORDER BY s.student_name, g.score DESC;
```

CTEを使用した方が、クエリの意図が明確で読みやすくなります。

複数CTEの活用

複数のCTEを定義することで、複雑な処理を段階的に分解できます。

例3：複数CTEによる総合分析

```
WITH
-- 各学生の平均成績を計算
student_averages AS (
    SELECT
        s.student_id,
        s.student_name,
        ROUND(AVG(g.score), 1) AS avg_score,
        COUNT(DISTINCT g.grade_id) AS total_grades
    FROM students s
    LEFT JOIN grades g ON s.student_id = g.student_id
    GROUP BY s.student_id, s.student_name
),
-- 各学生の出席率を計算
student_attendance AS (
    SELECT
        s.student_id,
        ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS
attendance_rate,
        COUNT(a.schedule_id) AS total_classes
    FROM students s
    LEFT JOIN attendance a ON s.student_id = a.student_id
    GROUP BY s.student_id
),
-- 各学生の受講講座数を計算
student_courses_count AS (
    SELECT
        s.student_id,
        COUNT(DISTINCT sc.course_id) AS course_count
    FROM students s
    LEFT JOIN student_courses sc ON s.student_id = sc.student_id
    GROUP BY s.student_id
)
-- すべての情報を統合
SELECT
    sa.student_id,
    sa.student_name AS 学生名,
    sa.avg_score AS 平均成績,
    sat.attendance_rate AS 出席率,
```

```
    scc.course_count AS 受講講座数,
    CASE
        WHEN sa.avg_score >= 85 AND sat.attendance_rate >= 90 THEN 'S評価'
        WHEN sa.avg_score >= 75 AND sat.attendance_rate >= 80 THEN 'A評価'
        WHEN sa.avg_score >= 65 AND sat.attendance_rate >= 70 THEN 'B評価'
        ELSE 'C評価'
    END AS 総合評価
FROM student_averages sa
LEFT JOIN student_attendance sat ON sa.student_id = sat.student_id
LEFT JOIN student_courses_count scc ON sa.student_id = scc.student_id
WHERE sa.total_grades > 0  -- 成績記録がある学生のみ
ORDER BY sa.avg_score DESC, sat.attendance_rate DESC;
```

この例では、3つのCTEを使用して：

1. 学生の平均成績を計算
2. 学生の出席率を計算
3. 学生の受講講座数を計算
4. 最終的にすべてを統合して総合評価を行っています

実行結果：

student_id	学生名	平均成績	出席率	受講講座数	総合評価
311	鈴木健太	89.8	92.5	6	S評価
302	新垣愛留	86.5	88.9	7	A評価
308	永田悦子	85.9	95.0	5	S評価
301	黒沢春馬	82.3	85.7	8	A評価
…	…	…	…	…	…

CTEの再利用

同じCTEを複数回参照することで、計算の重複を避けることができます。

例4：CTEの再利用による効率化

```
WITH course_stats AS (
    SELECT
        c.course_id,
        c.course_name,
        t.teacher_name,
        COUNT(DISTINCT sc.student_id) AS enrollment_count,
        ROUND(AVG(g.score), 1) AS avg_score,
        ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS
attendance_rate
    FROM courses c
    JOIN teachers t ON c.teacher_id = t.teacher_id
    LEFT JOIN student_courses sc ON c.course_id = sc.course_id
```

```
    LEFT JOIN grades g ON c.course_id = g.course_id
    LEFT JOIN attendance a ON sc.student_id = a.student_id
    GROUP BY c.course_id, c.course_name, t.teacher_name
)
SELECT
    '講座統計' AS カテゴリ,
    COUNT(*) AS 講座数,
    ROUND(AVG(enrollment_count), 1) AS 平均受講者数,
    ROUND(AVG(avg_score), 1) AS 全体平均成績,
    ROUND(AVG(attendance_rate), 1) AS 全体平均出席率
FROM course_stats

UNION ALL

SELECT
    '優秀講座' AS カテゴリ,
    COUNT(*) AS 講座数,
    ROUND(AVG(enrollment_count), 1) AS 平均受講者数,
    ROUND(AVG(avg_score), 1) AS 平均成績,
    ROUND(AVG(attendance_rate), 1) AS 平均出席率
FROM course_stats
WHERE avg_score >= 80 AND attendance_rate >= 85

UNION ALL

SELECT
    '改善必要講座' AS カテゴリ,
    COUNT(*) AS 講座数,
    ROUND(AVG(enrollment_count), 1) AS 平均受講者数,
    ROUND(AVG(avg_score), 1) AS 平均成績,
    ROUND(AVG(attendance_rate), 1) AS 平均出席率
FROM course_stats
WHERE avg_score < 75 OR attendance_rate < 75;
```

この例では、`course_stats` CTEを3回再利用して、全体統計、優秀講座、改善が必要な講座の統計をまとめて取得しています。

再帰CTE

再帰CTEは、自分自身を参照して階層データや連続データを処理するために使用されます。

> **注意**：再帰CTEはMySQL 8.0以降でサポートされています。

再帰CTEの基本構文

```
WITH RECURSIVE CTE名 AS (
    -- アンカーメンバー（初期データ）
    SELECT ...

    UNION ALL
```

```
    -- 再帰メンバー（自分自身を参照）
    SELECT ...
    FROM CTE名
    WHERE 終了条件
)
SELECT * FROM CTE名;
```

例5：数列の生成（再帰CTEの基本例）

1から10までの数列を生成：

```
WITH RECURSIVE number_sequence AS (
    -- アンカーメンバー：開始値
    SELECT 1 AS n

    UNION ALL

    -- 再帰メンバー：次の値を生成
    SELECT n + 1
    FROM number_sequence
    WHERE n < 10  -- 終了条件
)
SELECT n AS 番号
FROM number_sequence;
```

実行結果：

番号
1
2
3
4
5
6
7
8
9
10

例6：日付系列の生成

指定期間の全日付を生成して、授業日と休日を識別：

```
WITH RECURSIVE date_series AS (
    -- アンカーメンバー：開始日
    SELECT DATE('2025-05-01') AS date_value

    UNION ALL

    -- 再帰メンバー：次の日を生成
    SELECT DATE_ADD(date_value, INTERVAL 1 DAY)
    FROM date_series
    WHERE date_value < DATE('2025-05-31')  -- 終了条件
)
SELECT
    ds.date_value AS 日付,
    CASE DAYOFWEEK(ds.date_value)
        WHEN 1 THEN '日曜日'
        WHEN 2 THEN '月曜日'
        WHEN 3 THEN '火曜日'
        WHEN 4 THEN '水曜日'
        WHEN 5 THEN '木曜日'
        WHEN 6 THEN '金曜日'
        WHEN 7 THEN '土曜日'
    END AS 曜日,
    CASE
        WHEN cs.schedule_date IS NOT NULL THEN '授業日'
        WHEN DAYOFWEEK(ds.date_value) IN (1, 7) THEN '休日'
        ELSE '平日（授業なし）'
    END AS 種別,
    COUNT(cs.schedule_id) AS 授業数
FROM date_series ds
LEFT JOIN course_schedule cs ON ds.date_value = cs.schedule_date
GROUP BY ds.date_value
ORDER BY ds.date_value;
```

例7：学習進捗の累積計算（再帰CTE）

各学生の学習進捗を段階的に追跡：

```
WITH RECURSIVE learning_progress AS (
    -- アンカーメンバー：最初の成績記録
    SELECT
        g.student_id,
        s.student_name,
        g.grade_id,
        g.course_id,
        g.grade_type,
        g.score,
        g.submission_date,
        1 AS level,
        g.score AS cumulative_score,
        1 AS test_count
```

```
    FROM grades g
    JOIN students s ON g.student_id = s.student_id
    WHERE g.submission_date = (
        SELECT MIN(g2.submission_date)
        FROM grades g2
        WHERE g2.student_id = g.student_id
    )

    UNION ALL

    -- 再帰メンバー：次の成績記録
    SELECT
        g.student_id,
        lp.student_name,
        g.grade_id,
        g.course_id,
        g.grade_type,
        g.score,
        g.submission_date,
        lp.level + 1,
        lp.cumulative_score + g.score,
        lp.test_count + 1
    FROM learning_progress lp
    JOIN grades g ON lp.student_id = g.student_id
    WHERE g.submission_date > lp.submission_date
    AND g.submission_date = (
        SELECT MIN(g3.submission_date)
        FROM grades g3
        WHERE g3.student_id = lp.student_id
        AND g3.submission_date > lp.submission_date
    )
    AND lp.level < 10  -- 無限ループ防止
)
SELECT
    student_name AS 学生名,
    level AS レベル,
    grade_type AS 評価種別,
    score AS 今回点数,
    ROUND(cumulative_score / test_count, 1) AS 累積平均,
    submission_date AS 提出日
FROM learning_progress
WHERE student_id = 301
ORDER BY level;
```

CTEとウィンドウ関数の組み合わせ

CTEとウィンドウ関数を組み合わせることで、さらに高度な分析が可能になります。

例8：段階的な分析とランキング

```
WITH
-- ステップ1：基本統計の計算
basic_stats AS (
    SELECT
        c.course_id,
        c.course_name,
        t.teacher_name,
        COUNT(DISTINCT sc.student_id) AS student_count,
        ROUND(AVG(g.score), 1) AS avg_score,
        ROUND(STDDEV(g.score), 1) AS score_stddev,
        ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS
attendance_rate
    FROM courses c
    JOIN teachers t ON c.teacher_id = t.teacher_id
    LEFT JOIN student_courses sc ON c.course_id = sc.course_id
    LEFT JOIN grades g ON c.course_id = g.course_id AND g.grade_type = '中間テスト'
    LEFT JOIN attendance a ON sc.student_id = a.student_id
    GROUP BY c.course_id, c.course_name, t.teacher_name
),
-- ステップ2：ランキングの追加
ranked_courses AS (
    SELECT
        *,
        RANK() OVER (ORDER BY avg_score DESC) AS score_rank,
        RANK() OVER (ORDER BY attendance_rate DESC) AS attendance_rank,
        RANK() OVER (ORDER BY student_count DESC) AS popularity_rank
    FROM basic_stats
    WHERE student_count > 0
)
-- ステップ3：総合評価
SELECT
    course_name AS 講座名,
    teacher_name AS 担当教師,
    student_count AS 受講者数,
    avg_score AS 平均点,
    attendance_rate AS 出席率,
    score_rank AS 成績順位,
    attendance_rank AS 出席率順位,
    popularity_rank AS 人気順位,
    ROUND((score_rank + attendance_rank + popularity_rank) / 3.0, 1) AS 総合順位
FROM ranked_courses
ORDER BY 総合順位, score_rank;
```

CTEのパフォーマンス考慮点

CTEを効率的に使用するためのポイント：

1. 物理的実体化（Materialization）

CTEは、データベースによって実際のテーブルとして一時的に物理化される場合があります：

```
-- 効率的なCTEの例
WITH recent_grades AS (
    SELECT student_id, course_id, score
    FROM grades
    WHERE submission_date >= '2025-05-01'  -- 事前フィルタリング
)
SELECT
    s.student_name,
    rg.score
FROM recent_grades rg
JOIN students s ON rg.student_id = s.student_id
WHERE rg.score >= 80;  -- さらなるフィルタリング
```

2. インデックスの活用

CTEで使用するカラムには適切なインデックスを設定：

```
-- インデックスが有効なCTEの例
WITH high_performers AS (
    SELECT student_id  -- student_idにインデックスが必要
    FROM grades
    WHERE score >= 90  -- scoreにインデックスが有効
)
SELECT s.student_name
FROM high_performers hp
JOIN students s ON hp.student_id = s.student_id;  -- 結合キーにインデックス
```

3. 再帰CTEの制限

再帰CTEでは無限ループを防ぐため、適切な終了条件と制限を設定：

```
WITH RECURSIVE safe_recursion AS (
    SELECT 1 AS level, student_id
    FROM students
    WHERE student_id = 301

    UNION ALL

    SELECT level + 1, student_id
    FROM safe_recursion
    WHERE level < 100  -- 明確な終了条件
)
SELECT * FROM safe_recursion;
```

CTEの実践的な応用例

例9：学習ダッシュボードの作成

```
WITH
-- 全体統計
overall_stats AS (
    SELECT
        COUNT(DISTINCT s.student_id) AS total_students,
        COUNT(DISTINCT c.course_id) AS total_courses,
        COUNT(DISTINCT t.teacher_id) AS total_teachers,
        ROUND(AVG(g.score), 1) AS overall_avg_score
    FROM students s
    CROSS JOIN courses c
    CROSS JOIN teachers t
    LEFT JOIN grades g ON 1=1  -- 全体平均計算用
),
-- 今月の活動統計
monthly_stats AS (
    SELECT
        COUNT(DISTINCT cs.schedule_id) AS classes_this_month,
        COUNT(DISTINCT CASE WHEN a.status = 'present' THEN a.student_id END) AS
active_students,
        ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS
monthly_attendance_rate
    FROM course_schedule cs
    LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id
    WHERE cs.schedule_date >= DATE_FORMAT(CURRENT_DATE, '%Y-%m-01')
),
-- トップパフォーマー
top_performers AS (
    SELECT
        s.student_name,
        ROUND(AVG(g.score), 1) AS avg_score,
        ROW_NUMBER() OVER (ORDER BY AVG(g.score) DESC) AS rank
    FROM students s
    JOIN grades g ON s.student_id = g.student_id
    GROUP BY s.student_id, s.student_name
    HAVING COUNT(g.grade_id) >= 3  -- 最低3つの成績記録
    LIMIT 5
),
-- 問題のある学生
at_risk_students AS (
    SELECT
        s.student_name,
        ROUND(AVG(g.score), 1) AS avg_score,
        ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS
attendance_rate
    FROM students s
    LEFT JOIN grades g ON s.student_id = g.student_id
    LEFT JOIN attendance a ON s.student_id = a.student_id
    GROUP BY s.student_id, s.student_name
    HAVING (AVG(g.score) < 65 OR AVG(CASE WHEN a.status = 'present' THEN 100.0
ELSE 0 END) < 70)
    AND COUNT(g.grade_id) > 0
)
```

```
-- ダッシュボード結果の統合
SELECT
    '全体統計' AS カテゴリ,
    CONCAT('学生数: ', os.total_students, ', 講座数: ', os.total_courses, ', 教師
数: ', os.total_teachers) AS 詳細,
    CONCAT('全体平均: ', os.overall_avg_score, '点') AS 追加情報
FROM overall_stats os

UNION ALL

SELECT
    '今月の活動',
    CONCAT('授業数: ', ms.classes_this_month, ', アクティブ学生: ',
ms.active_students),
    CONCAT('出席率: ', ms.monthly_attendance_rate, '%')
FROM monthly_stats ms

UNION ALL

SELECT
    'トップ5学生',
    CONCAT(tp.rank, '位: ', tp.student_name),
    CONCAT('平均点: ', tp.avg_score, '点')
FROM top_performers tp

UNION ALL

SELECT
    '要注意学生',
    ars.student_name,
    CONCAT('平均点: ', ars.avg_score, '点, 出席率: ', ars.attendance_rate, '%')
FROM at_risk_students ars;
```

CTEのデバッグとトラブルシューティング

複雑なCTEのデバッグ方法：

1. 段階的な確認

```
-- ステップ1：最初のCTEのみを確認
WITH step1 AS (
    SELECT student_id, AVG(score) AS avg_score
    FROM grades
    GROUP BY student_id
)
SELECT * FROM step1 LIMIT 10;

-- ステップ2：2番目のCTEを追加
WITH
step1 AS (
    SELECT student_id, AVG(score) AS avg_score
```

```
    FROM grades
    GROUP BY student_id
),
step2 AS (
    SELECT student_id, COUNT(*) AS course_count
    FROM student_courses
    GROUP BY student_id
)
SELECT s1.*, s2.course_count
FROM step1 s1
LEFT JOIN step2 s2 ON s1.student_id = s2.student_id
LIMIT 10;
```

2. 中間結果の確認

```
WITH detailed_analysis AS (
    SELECT
        s.student_id,
        s.student_name,
        AVG(g.score) AS avg_score,
        COUNT(g.grade_id) AS grade_count,
        'デバッグ用' AS debug_flag  -- デバッグ用カラム
    FROM students s
    LEFT JOIN grades g ON s.student_id = g.student_id
    GROUP BY s.student_id, s.student_name
)
-- デバッグ時は中間結果を直接確認
SELECT * FROM detailed_analysis
WHERE grade_count IS NULL OR grade_count = 0;  -- 問題のあるレコードを特定
```

練習問題

問題25-1

CTEを使用して、各講座の受講者数、平均点、出席率を計算し、それらの全体平均と比較するSQLを書いてください。結果には講座名、各統計値、および全体平均との差を含めてください。

問題25-2

複数のCTEを使用して、「優秀な学生」（平均点85点以上）と「出席率の高い学生」（出席率90%以上）を定義し、両方の条件を満たす学生、どちらか一方だけを満たす学生、どちらも満たさない学生を分類するSQLを書いてください。

問題25-3

再帰CTEを使用して、2025年5月の全日付（1日～31日）を生成し、各日の授業数と出席者数を表示するSQLを書いてください。授業がない日は0と表示してください。

問題25-4

CTEとウィンドウ関数を組み合わせて、各学生の成績推移（中間テスト→レポート1→最終評価）を分析し、改善傾向、悪化傾向、安定傾向に分類するSQLを書いてください。

問題25-5

CTEを使用して教師の負荷分析を行い、担当講座数、総受講者数、平均成績、平均出席率を計算し、負荷が高い教師（担当講座数4以上または総受講者数40人以上）を特定するSQLを書いてください。

問題25-6

再帰CTEを使用して、各学生の「学習レベル」を定義するSQLを書いてください。レベル1は最初のテスト（60点以上）、レベル2は2回目のテスト（65点以上）、以降5点ずつ基準を上げて、最大レベル10まで計算してください。

解答

解答25-1

```
WITH
course_stats AS (
    SELECT
        c.course_id,
        c.course_name,
        COUNT(DISTINCT sc.student_id) AS enrollment_count,
        ROUND(AVG(g.score), 1) AS avg_score,
        ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS
attendance_rate
    FROM courses c
    LEFT JOIN student_courses sc ON c.course_id = sc.course_id
    LEFT JOIN grades g ON c.course_id = g.course_id
    LEFT JOIN attendance a ON sc.student_id = a.student_id
    GROUP BY c.course_id, c.course_name
),
overall_averages AS (
    SELECT
        ROUND(AVG(enrollment_count), 1) AS avg_enrollment,
        ROUND(AVG(avg_score), 1) AS overall_avg_score,
        ROUND(AVG(attendance_rate), 1) AS overall_attendance_rate
    FROM course_stats
    WHERE enrollment_count > 0
)
SELECT
    cs.course_name AS 講座名,
    cs.enrollment_count AS 受講者数,
    cs.avg_score AS 平均点,
    cs.attendance_rate AS 出席率,
    oa.overall_avg_score AS 全体平均点,
    oa.overall_attendance_rate AS 全体平均出席率,
    ROUND(cs.avg_score - oa.overall_avg_score, 1) AS 平均点差,
    ROUND(cs.attendance_rate - oa.overall_attendance_rate, 1) AS 出席率差
FROM course_stats cs
CROSS JOIN overall_averages oa
```

```
WHERE cs.enrollment_count > 0
ORDER BY cs.avg_score DESC;
```

解答25-2

```
WITH
excellent_students AS (
    SELECT s.student_id, s.student_name
    FROM students s
    JOIN grades g ON s.student_id = g.student_id
    GROUP BY s.student_id, s.student_name
    HAVING AVG(g.score) >= 85
),
high_attendance_students AS (
    SELECT s.student_id, s.student_name
    FROM students s
    JOIN attendance a ON s.student_id = a.student_id
    GROUP BY s.student_id, s.student_name
    HAVING AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END) >= 90
)
SELECT
    s.student_id,
    s.student_name AS 学生名,
    CASE
        WHEN es.student_id IS NOT NULL THEN '優秀'
        ELSE '一般'
    END AS 成績区分,
    CASE
        WHEN has.student_id IS NOT NULL THEN '高出席'
        ELSE '通常出席'
    END AS 出席区分,
    CASE
        WHEN es.student_id IS NOT NULL AND has.student_id IS NOT NULL THEN '両方満
足'
        WHEN es.student_id IS NOT NULL AND has.student_id IS NULL THEN '成績のみ優
秀'
        WHEN es.student_id IS NULL AND has.student_id IS NOT NULL THEN '出席のみ良
好'
        ELSE 'どちらも未達'
    END AS 総合分類
FROM students s
LEFT JOIN excellent_students es ON s.student_id = es.student_id
LEFT JOIN high_attendance_students has ON s.student_id = has.student_id
ORDER BY
    CASE
        WHEN es.student_id IS NOT NULL AND has.student_id IS NOT NULL THEN 1
        WHEN es.student_id IS NOT NULL OR has.student_id IS NOT NULL THEN 2
        ELSE 3
    END,
    s.student_name;
```

解答25-3

```
WITH RECURSIVE may_dates AS (
    -- アンカーメンバー：5月1日
    SELECT DATE('2025-05-01') AS date_value

    UNION ALL

    -- 再帰メンバー：次の日
    SELECT DATE_ADD(date_value, INTERVAL 1 DAY)
    FROM may_dates
    WHERE date_value < DATE('2025-05-31')
)
SELECT
    md.date_value AS 日付,
    CASE DAYOFWEEK(md.date_value)
        WHEN 1 THEN '日曜日'
        WHEN 2 THEN '月曜日'
        WHEN 3 THEN '火曜日'
        WHEN 4 THEN '水曜日'
        WHEN 5 THEN '木曜日'
        WHEN 6 THEN '金曜日'
        WHEN 7 THEN '土曜日'
    END AS 曜日,
    COALESCE(COUNT(cs.schedule_id), 0) AS 授業数,
    COALESCE(SUM(CASE WHEN a.status = 'present' THEN 1 ELSE 0 END), 0) AS 出席者数
FROM may_dates md
LEFT JOIN course_schedule cs ON md.date_value = cs.schedule_date
LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id
GROUP BY md.date_value
ORDER BY md.date_value;
```

解答25-4

```
WITH
student_progress AS (
    SELECT
        s.student_id,
        s.student_name,
        g.course_id,
        c.course_name,
        g.grade_type,
        g.score,
        ROW_NUMBER() OVER (PARTITION BY s.student_id, g.course_id
                           ORDER BY CASE g.grade_type
                               WHEN '中間テスト' THEN 1
                               WHEN 'レポート1' THEN 2
                               WHEN '最終評価' THEN 3
                           END) AS test_sequence,
        LAG(g.score) OVER (PARTITION BY s.student_id, g.course_id
```

```
                            ORDER BY CASE g.grade_type
                                WHEN '中間テスト' THEN 1
                                WHEN 'レポート1' THEN 2
                                WHEN '最終評価' THEN 3
                            END) AS prev_score
    FROM students s
    JOIN grades g ON s.student_id = g.student_id
    JOIN courses c ON g.course_id = c.course_id
    WHERE g.grade_type IN ('中間テスト', 'レポート1', '最終評価')
),
trend_analysis AS (
    SELECT
        student_id,
        student_name,
        course_id,
        course_name,
        COUNT(*) AS test_count,
        SUM(CASE WHEN score > prev_score THEN 1 ELSE 0 END) AS improvements,
        SUM(CASE WHEN score < prev_score THEN 1 ELSE 0 END) AS declines,
        SUM(CASE WHEN score = prev_score THEN 1 ELSE 0 END) AS stable
    FROM student_progress
    WHERE prev_score IS NOT NULL
    GROUP BY student_id, student_name, course_id, course_name
)
SELECT
    student_name AS 学生名,
    course_name AS 講座名,
    test_count AS 比較可能テスト数,
    improvements AS 改善回数,
    declines AS 悪化回数,
    stable AS 維持回数,
    CASE
        WHEN improvements > declines THEN '改善傾向'
        WHEN declines > improvements THEN '悪化傾向'
        ELSE '安定傾向'
    END AS 総合傾向
FROM trend_analysis
WHERE test_count >= 2
ORDER BY student_name, course_name;
```

解答25-5

```
WITH
teacher_workload AS (
    SELECT
        t.teacher_id,
        t.teacher_name,
        COUNT(DISTINCT c.course_id) AS course_count,
        COUNT(DISTINCT sc.student_id) AS total_students,
        ROUND(AVG(g.score), 1) AS avg_grade,
        ROUND(AVG(CASE WHEN a.status = 'present' THEN 100.0 ELSE 0 END), 1) AS
```

```
avg_attendance
    FROM teachers t
    LEFT JOIN courses c ON t.teacher_id = c.teacher_id
    LEFT JOIN student_courses sc ON c.course_id = sc.course_id
    LEFT JOIN grades g ON c.course_id = g.course_id
    LEFT JOIN attendance a ON sc.student_id = a.student_id
    GROUP BY t.teacher_id, t.teacher_name
),
workload_classification AS (
    SELECT
        *,
        CASE
            WHEN course_count >= 4 OR total_students >= 40 THEN '高負荷'
            WHEN course_count >= 2 OR total_students >= 20 THEN '中負荷'
            ELSE '低負荷'
        END AS load_level
    FROM teacher_workload
    WHERE course_count > 0
)
SELECT
    teacher_name AS 教師名,
    course_count AS 担当講座数,
    total_students AS 総受講者数,
    avg_grade AS 平均成績,
    avg_attendance AS 平均出席率,
    load_level AS 負荷レベル
FROM workload_classification
ORDER BY
    CASE load_level
        WHEN '高負荷' THEN 1
        WHEN '中負荷' THEN 2
        ELSE 3
    END,
    total_students DESC;
```

解答25-6

```
WITH RECURSIVE student_levels AS (
    -- アンカーメンバー：レベル1（最初のテストで60点以上）
    SELECT
        s.student_id,
        s.student_name,
        1 AS level,
        60 AS required_score,
        MIN(g.score) AS achieved_score,
        MIN(g.submission_date) AS achievement_date
    FROM students s
    JOIN grades g ON s.student_id = g.student_id
    WHERE g.score >= 60
    GROUP BY s.student_id, s.student_name
```

```
    UNION ALL

    -- 再帰メンバー：次のレベル
    SELECT
        sl.student_id,
        sl.student_name,
        sl.level + 1,
        sl.required_score + 5,  -- 5点ずつ基準を上げる
        MIN(g.score),
        MIN(g.submission_date)
    FROM student_levels sl
    JOIN grades g ON sl.student_id = g.student_id
    WHERE g.score >= (sl.required_score + 5)
    AND g.submission_date > sl.achievement_date
    AND sl.level < 10  -- 最大レベル10
    GROUP BY sl.student_id, sl.student_name, sl.level, sl.required_score
    HAVING MIN(g.score) >= (sl.required_score + 5)
)
SELECT
    student_name AS 学生名,
    MAX(level) AS 到達レベル,
    MAX(required_score) AS 最終基準点,
    COUNT(*) AS レベルアップ回数
FROM student_levels
GROUP BY student_id, student_name
ORDER BY MAX(level) DESC, student_name;
```

まとめ

この章では、共通テーブル式（CTE）について詳しく学びました：

1. **CTEの基本概念**：
    - WITH句を使用した一時的な名前付き結果セットの定義
    - クエリの可読性と再利用性の向上
    - 複雑なクエリの構造化
2. **CTEの基本構文**：
    - 単一CTEと複数CTEの記述方法
    - CTEの参照と再利用
    - サブクエリとの比較
3. **複数CTEの活用**：
    - 段階的な処理の分解
    - 複雑な分析の構造化
    - 同一CTEの複数回参照
4. **再帰CTE**：
    - 自分自身を参照する再帰的な処理

- アンカーメンバーと再帰メンバー
- 階層データや連続データの処理

5. **CTEとウィンドウ関数の組み合わせ**：

- 高度な分析機能の実現
- ランキングと統計の複合処理
- 段階的な計算とランキング

6. **パフォーマンス考慮点**：

- 物理的実体化の理解
- インデックスの効果的な活用
- 再帰CTEの制限設定

7. **実践的な応用例**：

- 学習ダッシュボードの作成
- 総合分析システム
- デバッグとトラブルシューティング

CTEは、複雑なSQLクエリをより読みやすく、保守しやすくするための重要な機能です。特に、段階的な処理や再帰的な操作において威力を発揮し、従来のサブクエリでは実現困難な高度な分析を可能にします。

これで第4章「高度なクエリ技術」が完了しました。サブクエリから始まり、相関サブクエリ、集合演算、EXISTS演算子、CASE式、ウィンドウ関数、そしてCTEまで、SQLの高度な機能を体系的に学習できました。これらの技術を組み合わせることで、実務で求められる複雑なデータ分析や処理を効率的に実現できるようになります。
