SQL学習テキスト 完全版

このテキストは、SQLの基礎から応用まで体系的に学習できるように構成されています。 学校データベースを使った実践的な例題と練習問題を通じて、実務で使えるSQLスキルを身につけることができます。

目次

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

```

```

```sql
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

```sql
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

student_id	student_name	平均点
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）

```sql
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

```sql
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

student_id	student_name	category
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
```

```sql
-- 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：

```sql
-- 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

student_id	student_name
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
...	...

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

```
-- 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

```
-- 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

```
-- 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
```

```sql
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

```sql
-- 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

```sql
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

学生名	日付	出席状況
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

student_id	学生名	平均点	受講講座数	出席率	総合評価
...	...	...	...	...	...

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

```sql
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
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

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：

```sql
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：

```sql
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

```sql
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

```sql
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。

SQL学習テキスト 完全版

このテキストは、SQLの基礎から応用まで体系的に学習できるように構成されています。 学校データベースを使った実践的な例題と練習問題を通じて、実務で使えるSQLスキルを身につけることができます。

目次

---

1. SELECT基本：単一テーブルから特定カラムを取得する

はじめに

データベースからデータを取り出す作業は、料理人が大きな冷蔵庫から必要な材料だけを取り出すようなものです。SQLでは、この「取り出す」作業を「SELECT文」で行います。

SELECT文はSQLの中で最も基本的で、最もよく使われる命令です。この章では、単一のテーブル（データの表）から必要な情報だけを取り出す方法を学びます。

基本構文

SELECT文の最も基本的な形は次のとおりです：

```
SELECT カラム名 FROM テーブル名;
```

この文は「テーブル名というテーブルからカラム名という列のデータを取り出してください」という意味です。

> **用語解説**：
>
> - **SELECT**：「選択する」という意味のSQLコマンドで、データを取り出すときに使います。
> - **カラム**：テーブルの縦の列のことで、同じ種類のデータが並んでいます（例：名前のカラム、年齢のカラムなど）。
> - **FROM**：「～から」という意味で、どのテーブルからデータを取るかを指定します。
> - **テーブル**：データベース内の表のことで、行と列で構成されています。

実践例：単一カラムの取得

学校データベースの中の「teachers」テーブル（教師テーブル）から、教師の名前だけを取得してみましょう。

```
SELECT teacher_name FROM teachers;
```

実行結果：

teacher_name
寺内鞍
田尻朋美
内村海凪
藤本理恵

teacher_name
黒木大介
星野涼子
深山誠一
吉岡由佳
山田太郎
佐藤花子
…

これは「teachers」テーブルの「teacher_name」という列（先生の名前）だけを取り出しています。

複数のカラムを取得する

料理に複数の材料が必要なように、データを取り出すときも複数の列が必要なことがよくあります。複数のカラムを取得するには、カラム名をカンマ（,）で区切って指定します。

```
SELECT カラム名1, カラム名2, カラム名3 FROM テーブル名;
```

例えば、教師の番号（ID）と名前を一緒に取得してみましょう：

```
SELECT teacher_id, teacher_name FROM teachers;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍
102	田尻朋美
103	内村海凪
104	藤本理恵
105	黒木大介
106	星野涼子
…	…

すべてのカラムを取得する

テーブルのすべての列を取得したい場合は、アスタリスク（*）を使います。これは「すべての列」を意味するワイルドカードです。

```
SELECT * FROM テーブル名;
```

例：

```
SELECT * FROM teachers;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍
102	田尻朋美
103	内村海凪
…	…

> **注意**：SELECT *は便利ですが、実際の業務では必要なカラムだけを指定する方が良いとされています。これは、データ量が多いときに処理速度が遅くなるのを防ぐためです。

カラムに別名をつける（AS句）

取得したカラムに分かりやすい名前（別名）をつけることができます。これは「AS」句を使います。

```
SELECT カラム名 AS 別名 FROM テーブル名;
```

> **用語解説**：
>
> - **AS**：「～として」という意味で、カラムに別名をつけるときに使います。この別名は結果を表示するときだけ使われます。

例えば、教師IDを「番号」、教師名を「名前」として表示してみましょう：

```
SELECT teacher_id AS 番号, teacher_name AS 名前 FROM teachers;
```

実行結果：

番号	名前
101	寺内鞍
102	田尻朋美
103	内村海凪

番号	名前
…	…

ASは省略することも可能です：

```
SELECT teacher_id 番号, teacher_name 名前 FROM teachers;
```

計算式を使う

SELECT文では、カラムの値を使った計算もできます。例えば、成績テーブルから点数と満点を取得して、達成率（パーセント）を計算してみましょう。

```
SELECT student_id, course_id, grade_type,
       score, max_score,
       (score / max_score) * 100 AS 達成率
FROM grades;
```

実行結果：

student_id	course_id	grade_type	score	max_score	達成率
301	1	中間テスト	85.5	100.0	85.5
302	1	中間テスト	92.0	100.0	92.0
…	…	…	…	…	…

重複を除外する（DISTINCT）

同じ値が複数ある場合に、重複を除いて一意の値だけを表示するには「DISTINCT」キーワードを使います。

> **用語解説：**
>
> - **DISTINCT**：「異なる」「区別された」という意味で、重複する値を除外して一意の値だけを取得します。

例えば、どの講座にどの教師が担当しているかを重複なしで確認してみましょう：

```
SELECT DISTINCT teacher_id FROM courses;
```

実行結果：

teacher_id
101

teacher_id
102
103
104

...

これにより、courses（講座）テーブルで使われている教師IDが重複なく表示されます。

文字列の結合

文字列を結合するには、MySQLでは「CONCAT」関数を使います。例えば、教師のIDと名前を組み合わせて表示してみましょう：

```
SELECT CONCAT('教師ID:', teacher_id, ' 名前:', teacher_name) AS 教師情報
FROM teachers;
```

実行結果：

教師情報
教師ID:101 名前:寺内鞍
教師ID:102 名前:田尻朋美

...

> **用語解説**：
>
> - **CONCAT**：複数の文字列を一つにつなげる関数です。

SELECT文と終了記号

SQLの文は通常、セミコロン（;）で終わります。これは「この命令はここで終わりです」という合図です。

複数のSQL文を一度に実行する場合は、それぞれの文の最後にセミコロンをつけます。

練習問題

問題1-1

students（学生）テーブルから、すべての学生の名前（student_name）を取得するSQLを書いてください。

問題1-2

classrooms（教室）テーブルから、教室ID（classroom_id）と教室名（classroom_name）を取得するSQLを書いてください。

問題1-3

coursesテーブルから、すべての列（カラム）を取得するSQLを書いてください。

問題1-4

class_periodsテーブルから、時限ID（period_id）、開始時間（start_time）、終了時間（end_time）を取得し、開始時間には「開始」、終了時間には「終了」という別名をつけるSQLを書いてください。

問題1-5

gradesテーブルから、学生ID（student_id）、講座ID（course_id）、評価タイプ（grade_type）、得点（score）、満点（max_score）、そして得点を満点で割って100を掛けた値を「パーセント」という別名で取得するSQLを書いてください。

問題1-6

course_scheduleテーブルから、schedule_date（予定日）カラムだけを重複なしで取得するSQLを書いてください。

解答

解答1-1

```
SELECT student_name FROM students;
```

解答1-2

```
SELECT classroom_id, classroom_name FROM classrooms;
```

解答1-3

```
SELECT * FROM courses;
```

解答1-4

```
SELECT period_id, start_time AS 開始, end_time AS 終了 FROM class_periods;
```

解答1-5

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score / max_score) * 100 AS パーセント
FROM grades;
```

解答1-6

```
SELECT DISTINCT schedule_date FROM course_schedule;
```

まとめ

この章では、SQLのSELECT文の基本を学びました：

1. 単一カラムの取得: `SELECT カラム名 FROM テーブル名;`
2. 複数カラムの取得: `SELECT カラム名1, カラム名2 FROM テーブル名;`
3. すべてのカラムの取得: `SELECT * FROM テーブル名;`
4. カラムに別名をつける: `SELECT カラム名 AS 別名 FROM テーブル名;`
5. 計算式を使う: `SELECT カラム名, (計算式) AS 別名 FROM テーブル名;`
6. 重複を除外する: `SELECT DISTINCT カラム名 FROM テーブル名;`
7. 文字列の結合: `SELECT CONCAT(文字列1, カラム名, 文字列2) FROM テーブル名;`

これらの基本操作を使いこなせるようになれば、データベースから必要な情報を効率よく取り出せるようになります。次の章では、WHERE句を使って条件に合ったデータだけを取り出す方法を学びます。

---

2. WHERE句：条件に合ったレコードを絞り込む

はじめに

前章では、テーブルからデータを取得する基本的な方法を学びました。しかし実際の業務では、すべてのデータではなく、特定の条件に合ったデータだけを取得したいことがほとんどです。

例えば、「全生徒の情報」ではなく「特定の学科の生徒だけ」や「成績が80点以上の学生だけ」といった形で、データを絞り込みたい場合があります。

このような場合に使用するのが「WHERE句」です。WHERE句は、SELECTコマンドの後に追加して使い、条件に合致するレコード（行）だけを取得します。

基本構文

WHERE句の基本的な形は次のとおりです：

```
SELECT カラム名 FROM テーブル名 WHERE 条件式;
```

> **用語解説：**
>
> - **WHERE**：「～の場所で」「～の条件で」という意味のSQLコマンドで、条件に合うデータだけを抽出するために使います。

> - **条件式**：データが満たすべき条件を指定するための式です。例えば「age > 20」（年齢が20より大きい）などです。
> - **レコード**：テーブルの横の行のことで、1つのデータの集まりを表します。

基本的な比較演算子

WHERE句では、様々な比較演算子を使って条件を指定できます：

演算子	意味	例
=	等しい	age = 25
<>	等しくない（≠と同じ）	gender <> 'male'
>	より大きい	score > 80
<	より小さい	price < 1000
>=	以上	height >= 170
<=	以下	weight <= 70

実践例：基本的な条件での絞り込み

例1：等しい（=）

例えば、教師ID（teacher_id）が101の教師のみを取得するには：

```
SELECT * FROM teachers WHERE teacher_id = 101;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍

例2：より大きい（>）

成績（grades）テーブルから、90点を超える成績だけを取得するには：

```
SELECT * FROM grades WHERE score > 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20

student_id	course_id	grade_type	score	max_score	submission_date
320	1	中間テスト	93.5	100.0	2025-05-20
...	...	...	...	...	...

例3：等しくない（<>）

講座IDが3ではない講座に関する成績を取得するには：

```
SELECT * FROM grades WHERE course_id <> '3';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
301	2	実技試験	88.0	100.0	2025-05-18
...	...	...	...	...	...

文字列の比較

テキスト（文字列）を条件にする場合は、シングルクォーテーション（'）またはダブルクォーテーション（"）で囲みます。MySQLではどちらも使えますが、多くの場合シングルクォーテーションが推奨されます。

```
SELECT * FROM テーブル名 WHERE テキストカラム = 'テキスト値';
```

例えば、教師名（teacher_name）が「田尻朋美」の教師を検索するには：

```
SELECT * FROM teachers WHERE teacher_name = '田尻朋美';
```

実行結果：

teacher_id	teacher_name
102	田尻朋美

日付の比較

日付の比較も同様にシングルクォーテーションで囲みます。日付の形式はデータベースの設定によって異なりますが、一般的にはISO形式（YYYY-MM-DD）が使われます。

```
SELECT * FROM テーブル名 WHERE 日付カラム = '日付';
```

例えば、2025年5月20日に提出された成績を検索するには：

```
SELECT * FROM grades WHERE submission_date = '2025-05-20';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
303	1	中間テスト	78.5	100.0	2025-05-20
…	…	…	…	…	…

また、日付同士の大小関係も比較できます：

```
SELECT * FROM grades WHERE submission_date < '2025-05-15';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	レポート1	45.0	50.0	2025-05-10
302	1	レポート1	48.0	50.0	2025-05-10
304	5	ER図作成課題	27.5	30.0	2025-05-14
…	…	…	…	…	…

複数の条件を指定する（次章の内容）

WHERE句では、複数の条件を組み合わせることもできます。この詳細は次章「論理演算子：AND、OR、NOTを使った複合条件」で説明します。

例えば、講座IDが1で、かつ、得点が90以上の成績を取得するには：

```
SELECT * FROM grades WHERE course_id = '1' AND score >= 90;
```

この例の詳細な説明は次章で行います。

練習問題

問題2-1

students（学生）テーブルから、学生ID（student_id）が「310」の学生情報を取得するSQLを書いてください。

問題2-2

classrooms（教室）テーブルから、収容人数（capacity）が30人より多い教室の情報をすべて取得するSQLを書いてください。

問題2-3

courses（講座）テーブルから、教師ID（teacher_id）が「105」の講座情報を取得するSQLを書いてください。

問題2-4

course_schedule（授業カレンダー）テーブルから、「2025-05-15」の授業スケジュールをすべて取得するSQLを書いてください。

問題2-5

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」で、得点（score）が80点未満の成績を取得するSQLを書いてください。

問題2-6

teachers（教師）テーブルから、教師ID（teacher_id）が「101」ではない教師の名前を取得するSQLを書いてください。

解答

解答2-1

```
SELECT * FROM students WHERE student_id = 310;
```

解答2-2

```
SELECT * FROM classrooms WHERE capacity > 30;
```

解答2-3

```
SELECT * FROM courses WHERE teacher_id = 105;
```

解答2-4

```sql
SELECT * FROM course_schedule WHERE schedule_date = '2025-05-15';
```

解答2-5

```sql
SELECT * FROM grades WHERE grade_type = '中間テスト' AND score < 80;
```

解答2-6

```sql
SELECT teacher_name FROM teachers WHERE teacher_id <> 101;
```

まとめ

この章では、WHERE句を使って条件に合ったレコードを取得する方法を学びました：

1. 基本的な比較演算子（=, <>, >, <, >=, <=）の使い方
2. 数値による条件絞り込み
3. 文字列（テキスト）による条件絞り込み
4. 日付による条件絞り込み

WHERE句は、大量のデータから必要な情報だけを取り出すための非常に重要な機能です。実際のデータベース操作では、この条件絞り込みを頻繁に使います。

次の章では、複数の条件を組み合わせるための「論理演算子（AND、OR、NOT）」について学びます。

---

3. 論理演算子：AND、OR、NOTを使った複合条件

はじめに

前章では、WHERE句を使って単一の条件でデータを絞り込む方法を学びました。しかし実際の業務では、より複雑な条件でデータを絞り込む必要があることがよくあります。

例えば：

- 「成績が80点以上**かつ**出席率が90%以上の学生」
- 「数学**または**英語の成績が優秀な学生」
- 「課題を**まだ提出していない**学生」

このような複合条件を指定するために使うのが論理演算子です。主な論理演算子は次の3つです：

- **AND**：両方の条件を満たす（かつ）
- **OR**：いずれかの条件を満たす（または）
- **NOT**：条件を満たさない（〜ではない）

AND演算子

AND演算子は、指定した**すべての条件を満たす**レコードだけを取得したいときに使います。

> **用語解説**：
>
> - **AND**：「かつ」「そして」という意味の論理演算子です。複数の条件をすべて満たすデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 条件1 AND 条件2;
```

例：ANDを使った複合条件

例えば、中間テストで90点以上かつ満点が100点の成績レコードを取得するには：

```
SELECT * FROM grades
WHERE score >= 90 AND max_score = 100;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

この例では、「score >= 90」と「max_score = 100」の両方の条件を満たすレコードだけが取得されます。

3つ以上の条件の組み合わせ

ANDを使って3つ以上の条件を組み合わせることもできます：

```
SELECT * FROM grades
WHERE score >= 90 AND max_score = 100 AND grade_type = '中間テスト';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20

student_id	course_id	grade_type	score	max_score	submission_date
320	1	中間テスト	93.5	100.0	2025-05-20
...	...	...	...	...	...

OR演算子

OR演算子は、指定した条件の**いずれか一つでも満たす**レコードを取得したいときに使います。

> **用語解説：**
>
> - **OR**：「または」「もしくは」という意味の論理演算子です。複数の条件のうち少なくとも1つを満たすデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 条件1 OR 条件2;
```

例：ORを使った複合条件

例えば、教師IDが101または102の講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id = 101 OR teacher_id = 102;
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
2	UNIX入門	102
3	Cプログラミング演習	101
29	コードリファクタリングとクリーンコード	101
40	ソフトウェアアーキテクチャパターン	102
...	...	...

この例では、「teacher_id = 101」または「teacher_id = 102」のいずれかの条件を満たすレコードが取得されます。

NOT演算子

NOT演算子は、指定した条件を**満たさない**レコードを取得したいときに使います。

> **用語解説：**

> - **NOT**：「～ではない」という意味の論理演算子です。条件を否定して、その条件を満たさないデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE NOT 条件;
```

例：NOTを使った否定条件

例えば、完了（completed）状態ではない授業スケジュールを取得するには：

```
SELECT * FROM course_schedule
WHERE NOT status = 'completed';
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
45	11	2025-05-20	5	401G	106	scheduled
46	12	2025-05-21	1	301E	107	scheduled
50	2	2025-05-23	3	101A	102	cancelled
…	…	…	…	…	…	…

この例では、statusが「completed」ではないレコード（scheduled状態やcancelled状態）が取得されます。

NOT演算子は、「～ではない」という否定の条件を作るために使われます。例えば次の2つの書き方は同じ意味になります：

```
SELECT * FROM teachers WHERE NOT teacher_id = 101;
SELECT * FROM teachers WHERE teacher_id <> 101;
```

複合論理条件（AND、OR、NOTの組み合わせ）

AND、OR、NOTを組み合わせて、より複雑な条件を指定することもできます。

例：ANDとORの組み合わせ

例えば、「成績が90点以上で中間テストである」または「成績が45点以上でレポートである」レコードを取得するには：

```
SELECT * FROM grades
WHERE (score >= 90 AND grade_type = '中間テスト')
   OR (score >= 45 AND grade_type = 'レポート1');
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
302	1	レポート1	48.0	50.0	2025-05-10
308	1	レポート1	47.0	50.0	2025-05-09
…	…	…	…	…	…

優先順位と括弧の使用

論理演算子を組み合わせる場合、演算子の優先順位に注意が必要です。基本的に**ANDはORよりも優先順位が高い**です。つまり、ANDが先に処理されます。

例えば：

```
WHERE 条件1 OR 条件2 AND 条件3
```

これは次のように解釈されます：

```
WHERE 条件1 OR (条件2 AND 条件3)
```

意図した条件と異なる場合は、**括弧（）を使って明示的にグループ化**することが重要です：

```
WHERE (条件1 OR 条件2) AND 条件3
```

例：括弧を使った条件のグループ化

教師IDが101または102で、かつ、講座名に「プログラミング」という単語が含まれる講座を取得するには：

```sql
SELECT * FROM courses
WHERE (teacher_id = 101 OR teacher_id = 102)
  AND course_name LIKE '%プログラミング%';
```

実行結果：

course_id	course_name	teacher_id
3	Cプログラミング演習	101

course_id	course_name	teacher_id
...	...	...

この例では、括弧を使って「teacher_id = 101 OR teacher_id = 102」の部分をグループ化し、その条件と「course_name LIKE '%プログラミング%'」の条件をANDで結合しています。

練習問題

問題3-1

grades（成績）テーブルから、課題タイプ（grade_type）が「中間テスト」かつ点数（score）が85点以上のレコードを取得するSQLを書いてください。

問題3-2

classrooms（教室）テーブルから、収容人数（capacity）が40人以下または建物（building）が「1号館」の教室を取得するSQLを書いてください。

問題3-3

teachers（教師）テーブルから、教師ID（teacher_id）が101、102、103ではない教師の情報を取得するSQLを書いてください。

問題3-4

course_schedule（授業カレンダー）テーブルから、2025年5月20日の授業で、時限（period_id）が1か2で、かつ状態（status）が「scheduled」の授業を取得するSQLを書いてください。

問題3-5

students（学生）テーブルから、学生名（student_name）に「田」または「山」を含む学生を取得するSQLを書いてください。

問題3-6

grades（成績）テーブルから、提出日（submission_date）が2025年5月15日以降で、かつ（「中間テスト」で90点以上または「レポート1」で45点以上）の成績を取得するSQLを書いてください。

解答

解答3-1

```sql
SELECT * FROM grades
WHERE grade_type = '中間テスト' AND score >= 85;
```

解答3-2

```
SELECT * FROM classrooms
WHERE capacity <= 40 OR building = '1号館';
```

解答3-3

```
SELECT * FROM teachers
WHERE NOT (teacher_id = 101 OR teacher_id = 102 OR teacher_id = 103);
```

または

```
SELECT * FROM teachers
WHERE teacher_id <> 101 AND teacher_id <> 102 AND teacher_id <> 103;
```

解答3-4

```
SELECT * FROM course_schedule
WHERE schedule_date = '2025-05-20'
  AND (period_id = 1 OR period_id = 2)
  AND status = 'scheduled';
```

解答3-5

```
SELECT * FROM students
WHERE student_name LIKE '%田%' OR student_name LIKE '%山%';
```

解答3-6

```
SELECT * FROM grades
WHERE submission_date >= '2025-05-15'
  AND ((grade_type = '中間テスト' AND score >= 90)
       OR (grade_type = 'レポート1' AND score >= 45));
```

まとめ

この章では、論理演算子（AND、OR、NOT）を使って複合条件を作る方法を学びました：

1. **AND**：すべての条件を満たすレコードを取得（条件1 AND 条件2）
2. **OR**：いずれかの条件を満たすレコードを取得（条件1 OR 条件2）
3. **NOT**：指定した条件を満たさないレコードを取得（NOT 条件）

4. **複合条件**：AND、OR、NOTを組み合わせたより複雑な条件
5. **括弧（）**：条件をグループ化して優先順位を明示的に指定

これらの論理演算子を使いこなすことで、より複雑で細かな条件でデータを絞り込むことができるようになります。実際のデータベース操作では、複数の条件を組み合わせることが頻繁にあるため、この章で学んだ内容は非常に重要です。

次の章では、テキストデータに対する検索を行う「パターンマッチング」について学びます。

---

4. パターンマッチング：LIKE演算子と%、_ワイルドカード

はじめに

前章までは、データの完全一致や数値の比較といった条件での絞り込みを学びました。しかし実際の業務では、もっと柔軟な検索が必要なケースがあります。例えば：

- 「山」で始まる名前の学生を検索したい
- 「プログラミング」という単語を含む講座名を探したい
- 電話番号の一部だけ覚えているデータを探したい

このような「部分一致」や「パターン一致」の検索を行うためのSQLの機能が「パターンマッチング」です。この章では、パターンマッチングを行うための「LIKE演算子」と「ワイルドカード文字」について学びます。

LIKE演算子の基本

LIKE演算子は、文字列のパターンマッチングを行うための演算子です。WHERE句と組み合わせて使用します。

> **用語解説**：
>
> - **LIKE**：「～のような」という意味の演算子で、パターンに一致する文字列を検索します。
> - **パターンマッチング**：完全一致ではなく、一定のパターンに合致するデータを検索する方法です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 文字列カラム LIKE 'パターン';
```

パターンには、通常の文字に加えて、特別な意味を持つ「ワイルドカード文字」を使用できます。

ワイルドカード文字

SQLでは主に2つのワイルドカード文字があります：

1. **%（パーセント）**：0文字以上の任意の文字列に一致します。

2. _**（アンダースコア）**：任意の1文字に一致します。

> **用語解説**：
>
> - **ワイルドカード**：任意の文字や文字列に一致する特殊な文字記号です。トランプのジョーカーのように、様々な値に代用できます。

LIKE演算子の使い方：実践例

例1：%（パーセント）を使ったパターンマッチング

「〜で始まる」パターン：前方一致

例えば、「山」で始まる学生名を検索するには：

```
SELECT * FROM students WHERE student_name LIKE '山%';
```

実行結果：

student_id	student_name
312	山本裕子
325	山田翔太
…	…

ここでの「山%」は「山で始まり、その後に0文字以上の任意の文字が続く」という意味です。

「〜で終わる」パターン：後方一致

例えば、「子」で終わる教師名を検索するには：

```
SELECT * FROM teachers WHERE teacher_name LIKE '%子';
```

実行結果：

teacher_id	teacher_name
102	田尻朋美
106	星野涼子
108	吉岡由佳
110	佐藤花子
…	…

「〜を含む」パターン：部分一致

例えば、「プログラミング」という単語を含む講座名を検索するには：

```
SELECT * FROM courses WHERE course_name LIKE '%プログラミング%';
```

実行結果：

course_id	course_name	teacher_id
3	Cプログラミング演習	101
14	IoTデバイスプログラミング実践	110
…	…	…

例2：_（アンダースコア）を使ったパターンマッチング

アンダースコアは任意の1文字に一致します。例えば、教室IDが「10_A」パターン（最初の2文字が「10」、3文字目が任意の1文字、最後が「A」）の教室を検索するには：

```
SELECT * FROM classrooms WHERE classroom_id LIKE '10_A';
```

実行結果：

classroom_id	classroom_name	capacity	building	facilities
101A	1号館コンピュータ実習室A	30	1号館	パソコン30台、プロジェクター
…	…	…	…	…

例3：%と_の組み合わせ

ワイルドカード文字は組み合わせて使うこともできます。例えば、「2文字目が田」の学生を検索するには：

```
SELECT * FROM students WHERE student_name LIKE '_田%';
```

実行結果：

student_id	student_name
321	井上竜也
384	櫻井翼
…	…

NOT LIKEを使った否定形のパターンマッチング

特定のパターンに一致しないレコードを検索したい場合は、「NOT LIKE」を使います。

> **用語解説：**
>
> - **NOT LIKE**：「～のパターンに一致しない」という意味で、指定したパターンに一致しないデータを検索します。

例えば、講座名に「入門」を含まない講座を検索するには：

```
SELECT * FROM courses WHERE course_name NOT LIKE '%入門%';
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
3	Cプログラミング演習	101
…	…	…

エスケープ文字の使用

もし検索したいパターンに「%」や「_」自体が含まれている場合は、それらを特別な文字としてではなく、通常の文字として扱うために「エスケープ文字」を使います。

> **用語解説：**
>
> - **エスケープ文字**：特別な意味を持つ文字を通常の文字として扱うための印です。

MySQLでは、バックスラッシュ（\）をエスケープ文字として使用できます。例えば、「50%」という値そのものを検索するには：

```
SELECT * FROM テーブル名 WHERE カラム LIKE '50\%';
```

または、ESCAPE句を使って明示的にエスケープ文字を指定することもできます：

```
SELECT * FROM テーブル名 WHERE カラム LIKE '50!%' ESCAPE '!';
```

この例では「!」をエスケープ文字として指定しています。

大文字と小文字の区別

MySQLのデフォルト設定では、LIKE演算子は大文字と小文字を区別しません（大文字小文字を同じものとして扱います）。

例えば、次の2つのクエリは同じ結果を返します：

```
SELECT * FROM courses WHERE course_name LIKE '%web%';
SELECT * FROM courses WHERE course_name LIKE '%Web%';
```

もし大文字と小文字を区別した検索が必要な場合は、「BINARY」キーワードを使用します：

```
SELECT * FROM courses WHERE course_name LIKE BINARY '%Web%';
```

この場合、「Web」は「web」とは一致しません。

複合条件との組み合わせ

LIKE演算子は、これまで学んだAND、OR、NOTなどの論理演算子と組み合わせて使うこともできます。

例えば、「田」で始まる名前で、かつ教師IDが102から105の間の教師を検索するには：

```
SELECT * FROM teachers
WHERE teacher_name LIKE '田%'
  AND teacher_id BETWEEN 102 AND 105;
```

実行結果：

teacher_id	teacher_name
102	田尻朋美
…	…

練習問題

問題4-1

students（学生）テーブルから、学生名（student_name）が「佐藤」で始まる学生の情報をすべて取得するSQLを書いてください。

問題4-2

courses（講座）テーブルから、講座名（course_name）に「データ」という単語を含む講座の情報を取得するSQLを書いてください。

問題4-3

classrooms（教室）テーブルから、教室名（classroom_name）が「コンピュータ実習室」で終わる教室の情報を取得するSQLを書いてください。

問題4-4

teachers（教師）テーブルから、教師名（teacher_name）の2文字目が「木」である教師の情報を取得するSQLを書いてください。

問題4-5

courses（講座）テーブルから、講座名（course_name）に「入門」または「基礎」を含む講座を取得するSQLを書いてください。

問題4-6

students（学生）テーブルから、学生名（student_name）が「山」で始まり、かつ「子」で終わらない学生を取得するSQLを書いてください。

解答

解答4-1

```
SELECT * FROM students WHERE student_name LIKE '佐藤%';
```

解答4-2

```
SELECT * FROM courses WHERE course_name LIKE '%データ%';
```

解答4-3

```
SELECT * FROM classrooms WHERE classroom_name LIKE '%コンピュータ実習室';
```

解答4-4

```
SELECT * FROM teachers WHERE teacher_name LIKE '_木%';
```

解答4-5

```
SELECT * FROM courses
WHERE course_name LIKE '%入門%' OR course_name LIKE '%基礎%';
```

解答4-6

```
SELECT * FROM students
WHERE student_name LIKE '山%' AND student_name NOT LIKE '%子';
```

まとめ

この章では、パターンマッチングを行うためのLIKE演算子と、その中で使用するワイルドカード文字（%と_）について学びました：

1. **LIKE演算子**：文字列パターンに一致するデータを検索するための演算子
2. **%（パーセント）**：0文字以上の任意の文字列に一致するワイルドカード
3. **_（アンダースコア）**：任意の1文字に一致するワイルドカード
4. **前方一致**：「パターン%」で「パターンで始まる」文字列に一致
5. **後方一致**：「%パターン」で「パターンで終わる」文字列に一致
6. **部分一致**：「%パターン%」で「パターンを含む」文字列に一致
7. **NOT LIKE**：指定したパターンに一致しないデータを検索
8. **エスケープ文字**：特殊文字（%や_）を通常の文字として扱うための方法
9. **複合条件との組み合わせ**：AND、ORなどと組み合わせたより複雑な条件

パターンマッチングは、特にテキストデータを扱う際に非常に便利な機能です。部分的な情報しか持っていない場合や、特定のパターンを持つデータを探す場合に活用できます。

次の章では、範囲指定のための「BETWEEN演算子」と「IN演算子」について学びます。

---

5. 範囲指定：BETWEEN、IN演算子

はじめに

これまでの章では、等号（=）や不等号（>、<）を使って条件を指定する方法や、LIKE演算子を使ったパターンマッチングを学びました。この章では、値の範囲を指定する「BETWEEN演算子」と、複数の値を一度に指定できる「IN演算子」について学びます。

これらの演算子を使うと、次のような検索がより簡単になります：

- 「80点から90点の間の成績」
- 「2025年4月から6月の間のスケジュール」
- 「特定の教師IDリストに該当する講座」

BETWEEN演算子：範囲を指定する

BETWEEN演算子は、ある値が指定した範囲内にあるかどうかを調べるために使います。

> **用語解説**：
>
> - **BETWEEN**：「～の間に」という意味の演算子で、ある値が指定した最小値と最大値の間（両端の値を含む）にあるかどうかを判定します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 BETWEEN 最小値 AND 最大値;
```

この構文は次の条件と同じ意味です：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 >= 最小値 AND カラム名 <= 最大値;
```

例1：数値範囲の指定

例えば、成績（grades）テーブルから、80点から90点の間の成績を取得するには：

```
SELECT * FROM grades
WHERE score BETWEEN 80 AND 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
308	6	小テスト1	89.0	100.0	2025-05-15
…	…	…	…	…	…

例2：日付範囲の指定

日付にもBETWEEN演算子が使えます。例えば、2025年5月10日から2025年5月20日までに提出された成績を取得するには：

```
SELECT * FROM grades
WHERE submission_date BETWEEN '2025-05-10' AND '2025-05-20';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	レポート1	45.0	50.0	2025-05-10
302	1	レポート1	48.0	50.0	2025-05-10
301	1	中間テスト	85.5	100.0	2025-05-20
…	…	…	…	…	…

NOT BETWEEN：範囲外を指定する

NOT BETWEENを使うと、指定した範囲の外にある値を持つレコードを取得できます。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 NOT BETWEEN 最小値 AND 最大値;
```

これは次の条件と同じです：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 < 最小値 OR カラム名 > 最大値;
```

例：範囲外の値を取得

例えば、80点未満または90点より高い成績を取得するには：

```
SELECT * FROM grades
WHERE score NOT BETWEEN 80 AND 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
303	1	中間テスト	78.5	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
…	…	…	…	…	…

IN演算子：複数の値を指定する

IN演算子は、ある値が指定した複数の値のリストのいずれかに一致するかどうかを調べるために使います。

> **用語解説**：
>
> - **IN**：「～の中に含まれる」という意味の演算子で、ある値が指定したリストの中に含まれているかどうかを判定します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 IN (値1, 値2, 値3, ...);
```

この構文は次のOR条件の組み合わせと同じ意味です：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 = 値1 OR カラム名 = 値2 OR カラム名 = 値3 OR ...;
```

例1：数値リストの指定

例えば、教師ID（teacher_id）が101、103、105のいずれかである講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id IN (101, 103, 105);
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
3	Cプログラミング演習	101
5	データベース設計と実装	105
10	プロジェクト管理手法	103
…	…	…

例2：文字列リストの指定

文字列のリストにも適用できます。例えば、特定の教室ID（classroom_id）のみの教室情報を取得するには：

```
SELECT * FROM classrooms
WHERE classroom_id IN ('101A', '202D', '301E');
```

実行結果：

classroom_id	classroom_name	capacity	building	facilities
101A	1号館コンピュータ実習室A	30	1号館	パソコン30台、プロジェクター
202D	2号館コンピュータ実習室D	25	2号館	パソコン25台、プロジェクター、3Dプリンター
301E	3号館講義室E	80	3号館	プロジェクター、マイク設備、録画設備

NOT IN：リストに含まれない値を指定する

NOT IN演算子を使うと、指定したリストに含まれない値を持つレコードを取得できます。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 NOT IN (値1, 値2, 値3, ...);
```

これは次の条件と同じです：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 <> 値1 AND カラム名 <> 値2 AND カラム名 <> 値3 AND ...;
```

例：リストに含まれない値を取得

例えば、教師IDが101、102、103以外の教師が担当する講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id NOT IN (101, 102, 103);
```

実行結果：

course_id	course_name	teacher_id
4	Webアプリケーション開発	104
5	データベース設計と実装	105
6	ネットワークセキュリティ	107
…	…	…

IN演算子でのサブクエリの利用（基本）

IN演算子の括弧内には、直接値を書く代わりに、サブクエリ（別のSELECT文）を指定することもできます。これにより、動的に値のリストを生成できます。

> **用語解説**：
>
> - **サブクエリ**：SQL文の中に含まれる別のSQL文のことで、外側のSQL文（メインクエリ）に値や条件を提供します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 IN (SELECT カラム名 FROM 別テーブル WHERE 条件);
```

例：サブクエリを使ったIN条件

例えば、教師名（teacher_name）に「田」を含む教師が担当している講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id IN (
    SELECT teacher_id FROM teachers
    WHERE teacher_name LIKE '%田%'
);
```

実行結果：

course_id	course_name	teacher_id
2	UNIX入門	102
10	プロジェクト管理手法	103
…	…	…

この例では、まず「teachers」テーブルから名前に「田」を含む教師のIDを取得し、それらのIDを持つ講座を「courses」テーブルから取得しています。

BETWEEN演算子とIN演算子の組み合わせ

BETWEEN演算子とIN演算子は、論理演算子（AND、OR）と組み合わせて、さらに複雑な条件を作ることができます。

例：BETWEENとINの組み合わせ

例えば、「教師IDが101、103、105のいずれかで、かつ、2025年5月15日から2025年6月15日の間に実施される授業」を取得するには：

```
SELECT * FROM course_schedule
WHERE teacher_id IN (101, 103, 105)
  AND schedule_date BETWEEN '2025-05-15' AND '2025-06-15';
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
38	1	2025-05-20	1	102B	101	scheduled
42	10	2025-05-23	3	201C	103	scheduled

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
45	5	2025-05-28	2	402H	105	scheduled
...	...	...	...	...	...	...

練習問題

問題5-1

grades（成績）テーブルから、得点（score）が85点から95点の間にある成績を取得するSQLを書いてください。

問題5-2

course_schedule（授業カレンダー）テーブルから、2025年5月1日から2025年5月31日までの授業スケジュールを取得するSQLを書いてください。

問題5-3

courses（講座）テーブルから、教師ID（teacher_id）が104、106、108のいずれかである講座の情報を取得するSQLを書いてください。

問題5-4

classrooms（教室）テーブルから、教室ID（classroom_id）が「101A」、「201C」、「301E」、「401G」以外の教室情報を取得するSQLを書いてください。

問題5-5

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」または「実技試験」で、かつ得点（score）が80点から90点の間にない成績を取得するSQLを書いてください。

問題5-6

course_schedule（授業カレンダー）テーブルから、教室ID（classroom_id）が「101A」、「202D」のいずれかで、かつ2025年5月15日から2025年5月30日の間に実施される授業スケジュールを取得するSQLを書いてください。

解答

解答5-1

```sql
SELECT * FROM grades
WHERE score BETWEEN 85 AND 95;
```

解答5-2

```
SELECT * FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31';
```

解答5-3

```
SELECT * FROM courses
WHERE teacher_id IN (104, 106, 108);
```

解答5-4

```
SELECT * FROM classrooms
WHERE classroom_id NOT IN ('101A', '201C', '301E', '401G');
```

解答5-5

```
SELECT * FROM grades
WHERE grade_type IN ('中間テスト', '実技試験')
  AND score NOT BETWEEN 80 AND 90;
```

解答5-6

```
SELECT * FROM course_schedule
WHERE classroom_id IN ('101A', '202D')
  AND schedule_date BETWEEN '2025-05-15' AND '2025-05-30';
```

まとめ

この章では、範囲指定のための「BETWEEN演算子」と複数値指定のための「IN演算子」について学びました：

1. **BETWEEN演算子**：値が指定した範囲内（両端を含む）にあるかどうかをチェック
2. **NOT BETWEEN**：値が指定した範囲外にあるかどうかをチェック
3. **IN演算子**：値が指定したリストのいずれかに一致するかどうかをチェック
4. **NOT IN**：値が指定したリストのいずれにも一致しないかどうかをチェック
5. **サブクエリとIN**：動的に生成された値のリストを使用する方法
6. **複合条件**：BETWEEN、IN、論理演算子を組み合わせたより複雑な条件

これらの演算子を使うことで、複数の条件を指定する場合に、SQLをより簡潔に書くことができます。特に、多くの値を指定する場合や範囲条件を指定する場合に便利です。

次の章では、「NULL値の処理：IS NULL、IS NOT NULL」について学びます。

6. NULL値の処理：IS NULL、IS NOT NULL

はじめに

データベースの世界では、データがない状態を表すために「NULL」という特別な値が使われます。NULLは「空」や「0」や「空白文字」とは異なる、「値が存在しない」または「不明」であることを表す特殊な概念です。

例えば、学校データベースでは次のようなシナリオがあります：

- まだ成績が付けられていない（NULL）
- コメントが入力されていない（NULL）
- 授業がキャンセルされたため教室が割り当てられていない（NULL）

この章では、NULL値を正しく処理するための「IS NULL」と「IS NOT NULL」演算子について学びます。

NULLとは何か？

NULL値には、いくつかの特徴があります：

1. **値がない**：NULLは値がないことを表します。0でも空文字列（''）でもなく、値そのものが存在しないことを示します。
2. **不明**：データが不明であることを表す場合もあります。
3. **未設定**：まだ値が設定されていないことを表す場合もあります。
4. **比較できない**：NULLは通常の比較演算子（=, <, >など）で比較できません。

> **用語解説**：
>
> - **NULL**：データベースにおいて「値がない」または「不明」を表す特殊な値です。0や空文字とは異なります。

NULL値と通常の比較演算子

通常の比較演算子（=, <>, >, <, >=, <=）では、NULL値を正しく検出できません。例えば：

```
-- この条件はNULL値に対して常にFALSEを返す
SELECT * FROM テーブル名 WHERE カラム名 = NULL;

-- この条件もNULL値に対して常にFALSEを返す
SELECT * FROM テーブル名 WHERE カラム名 <> NULL;
```

これは、NULL値との等価比較は「不明」と評価されるためです。つまり、NULL = NULLでさえFALSEではなく「不明」になります。

IS NULL演算子

NULL値を持つレコードを検索するには、「IS NULL」演算子を使います。

> **用語解説：**
>
> - **IS NULL**：カラムの値がNULLかどうかを調べる演算子です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE カラム名 IS NULL;
```

例：IS NULLの使用

例えば、コメントが入力されていない（NULL）出席レコードを検索するには：

```
SELECT * FROM attendance WHERE comment IS NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	NULL
1	306	present	NULL
1	307	present	NULL
...	...	...	NULL

IS NOT NULL演算子

逆に、NULL値を持たないレコード（つまり、何らかの値を持つレコード）を検索するには、「IS NOT NULL」演算子を使います。

> **用語解説：**
>
> - **IS NOT NULL**：カラムの値がNULLでないかどうかを調べる演算子です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE カラム名 IS NOT NULL;
```

例：IS NOT NULLの使用

例えば、コメントが入力されている（NOT NULL）出席レコードを検索するには：

```
SELECT * FROM attendance WHERE comment IS NOT NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	302	late	15分遅刻
1	303	absent	事前連絡あり
1	308	late	5分遅刻
…	…	…	…

NULL値の論理的な扱い

NULL値は論理演算（AND、OR、NOT）でも特殊な扱いを受けます。

- **NULL AND TRUE** → NULL（不明）
- **NULL AND FALSE** → FALSE
- **NULL OR TRUE** → TRUE
- **NULL OR FALSE** → NULL（不明）
- **NOT NULL** → NULL（不明）

この特殊な振る舞いが、バグや誤った結果の原因になることがあります。

NULL値と結合条件

テーブル結合（JOINなど、後の章で学習）の際も、NULL値は特殊な扱いを受けます。NULL値同士は「等しい」とは判定されないため、通常の結合条件ではNULL値を持つレコードは結合されません。

IS NULLとIS NOT NULLを使った複合条件

IS NULLとIS NOT NULLも、他の条件と組み合わせて使用できます。

例：複合条件でのIS NULLの使用

例えば、「出席状態が "absent"（欠席）で、コメントがNULLでない（理由が入力されている）レコード」を検索するには：

```
SELECT * FROM attendance
WHERE status = 'absent' AND comment IS NOT NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	303	absent	事前連絡あり
1	317	absent	体調不良
…	…	…	…

NVL/IFNULL/COALESCE関数：NULL値の置換

NULL値を別の値に置き換えるための関数が用意されています。データベースによって関数名が異なることがありますが、機能は似ています：

- MySQL/MariaDB: **IFNULL(expr, replace_value)**
- Oracle: **NVL(expr, replace_value)**
- SQL Server: **ISNULL(expr, replace_value)**
- 標準SQL: **COALESCE(expr1, expr2, ..., exprN)** - 最初のNULLでない式を返します

例：IFNULL関数の使用（MySQL）

例えば、コメントがNULLの場合は「特記事項なし」と表示するには：

```
SELECT schedule_id, student_id, status,
       IFNULL(comment, '特記事項なし') AS comment
FROM attendance;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	特記事項なし
1	302	late	15分遅刻
1	303	absent	事前連絡あり
...	...	...	...

NULLを使う際の注意点

1. **除外の罠**：`WHERE カラム名 <> 値` だけでは、NULL値を持つレコードは含まれません。すべてのレコードを対象にするには：

   ```
   WHERE カラム名 <> 値 OR カラム名 IS NULL
   ```

2. **集計関数**：COUNT(*)はすべての行を数えますが、COUNT(カラム名)はそのカラムがNULLでない行だけを数えます。

3. **インデックス**：多くのデータベースでは、NULL値にもインデックスを適用できますが、データベースによって動作が異なる場合があります。

4. **一意性制約**：一般的に、UNIQUE制約ではNULL値は重複としてカウントされません（複数のNULL値が許可されます）。

練習問題

問題6-1

attendance（出席）テーブルから、コメント（comment）がNULLの出席レコードをすべて取得するSQLを書いてください。

問題6-2

course_schedule（授業カレンダー）テーブルから、状態（status）が「cancelled」で、かつ教室ID（classroom_id）がNULLでないレコードを取得するSQLを書いてください。

問題6-3

grades（成績）テーブルから、提出日（submission_date）がNULLの成績レコードを取得するSQLを書いてください。

問題6-4

attendance（出席）テーブルから、出席状態（status）が「present」か「late」で、かつコメント（comment）がNULLのレコードを取得するSQLを書いてください。

問題6-5

以下のSQLで教師（teachers）テーブルから「佐藤」という名前を持つ教師を検索する場合、NULL値を持つレコードも含めるにはどう修正すべきですか？

```
SELECT * FROM teachers WHERE teacher_name <> '佐藤花子';
```

問題6-6

attendance（出席）テーブルのすべてのレコードを取得し、コメント（comment）がNULLの場合は「記録なし」と表示するSQLを書いてください。

解答

解答6-1

```
SELECT * FROM attendance WHERE comment IS NULL;
```

解答6-2

```
SELECT * FROM course_schedule
WHERE status = 'cancelled' AND classroom_id IS NOT NULL;
```

解答6-3

```
SELECT * FROM grades WHERE submission_date IS NULL;
```

解答6-4

```
SELECT * FROM attendance
WHERE (status = 'present' OR status = 'late') AND comment IS NULL;
```

または

```
SELECT * FROM attendance
WHERE status IN ('present', 'late') AND comment IS NULL;
```

解答6-5

```
SELECT * FROM teachers
WHERE teacher_name <> '佐藤花子' OR teacher_name IS NULL;
```

解答6-6

```
SELECT schedule_id, student_id, status,
       IFNULL(comment, '記録なし') AS comment
FROM attendance;
```

まとめ

この章では、データベースにおけるNULL値の概念と、NULL値を扱うための演算子や関数について学びました：

1. **NULL値の概念**：値がない、不明、未設定を表す特殊な値
2. **IS NULL演算子**：NULL値を持つレコードを検索する方法
3. **IS NOT NULL演算子**：NULL値を持たないレコードを検索する方法
4. **NULL値の論理的扱い**：論理演算（AND、OR、NOT）におけるNULLの振る舞い
5. **複合条件**：IS NULL/IS NOT NULLと他の条件の組み合わせ
6. **NULL値の置換**：IFNULL/NVL/COALESCE関数の使用方法
7. **注意点**：NULL値を扱う際の一般的な落とし穴

NULL値の正確な理解と適切な処理は、SQLプログラミングの重要な部分です。不適切なNULL処理は、予期しない結果やバグの原因になります。

次の章では、クエリ結果の並び替えを行うための「ORDER BY：結果の並び替え」について学びます。

---

7. ORDER BY：結果の並び替え

はじめに

これまでの章では、データベースから条件に合ったレコードを取得する方法を学んできました。しかし実際の業務では、取得したデータを見やすく整理する必要があります。例えば：

- 成績を高い順に表示したい
- 学生を名前の五十音順に並べたい
- 日付の新しい順にスケジュールを確認したい

このようなデータの「並び替え」を行うためのSQLコマンドが「ORDER BY」です。この章では、クエリ結果を特定の順序で並べる方法を学びます。

ORDER BYの基本

ORDER BY句は、SELECT文の結果を指定したカラムの値に基づいて並び替えるために使います。

> **用語解説**：
>
> - **ORDER BY**：「～の順に並べる」という意味のSQLコマンドで、クエリ結果の並び順を指定します。

基本構文

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] ORDER BY 並び替えカラム;
```

ORDER BY句は通常、SELECT文の最後に記述します。

例1：単一カラムでの並び替え

例えば、学生（students）テーブルから、学生名（student_name）の五十音順（辞書順）でデータを取得するには：

```
SELECT * FROM students ORDER BY student_name;
```

実行結果：

student_id	student_name
309	相沢吉夫
303	柴崎春花
306	河田咲奈
305	河口菜恵子

student_id	student_name
...	...

デフォルトの並び順

ORDER BYを使わない場合、結果の順序は保証されません。多くの場合、データがデータベースに保存された順序で返されますが、これは信頼できるものではありません。

昇順と降順の指定

ORDER BY句では、並び順を「昇順」か「降順」のどちらかで指定できます。

> **用語解説**：
>
> - **昇順（ASC）**：小さい値から大きい値へ（A→Z、1→9）の順に並べます。
> - **降順（DESC）**：大きい値から小さい値へ（Z→A、9→1）の順に並べます。

構文

```
SELECT カラム名 FROM テーブル名 ORDER BY 並び替えカラム [ASC|DESC];
```

ASC（昇順）がデフォルトのため、省略可能です。

例2：降順での並び替え

例えば、成績（grades）テーブルから、得点（score）の高い順（降順）に成績を取得するには：

```
SELECT * FROM grades ORDER BY score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
...	...	...	...	...	...

複数カラムでの並び替え

複数のカラムを使って並び替えることもできます。最初に指定したカラムで並び替え、値が同じレコードがある場合は次のカラムで並び替えます。

構文

```
SELECT カラム名 FROM テーブル名
ORDER BY 並び替えカラム1 [ASC|DESC], 並び替えカラム2 [ASC|DESC], ...;
```

例3：複数カラムでの並び替え

例えば、成績（grades）テーブルから、課題タイプ（grade_type）の五十音順に並べ、同じ課題タイプ内では得点（score）の高い順に成績を取得するには：

```
SELECT * FROM grades
ORDER BY grade_type ASC, score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	2	実技試験	88.0	100.0	2025-05-18
321	2	実技試験	85.5	100.0	2025-05-18
…	…	…	…	…	…
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…
311	1	レポート1	49.0	50.0	2025-05-08
302	1	レポート1	48.0	50.0	2025-05-10
…	…	…	…	…	…

例4：昇順と降順の混合

各カラムごとに並び順を指定することもできます。例えば、講座ID（course_id）の昇順、評価タイプ（grade_type）の昇順、得点（score）の降順で並べるには：

```
SELECT * FROM grades
ORDER BY course_id ASC, grade_type ASC, score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	レポート1	49.0	50.0	2025-05-08
…	…	…	…	…	…

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
...	...	...	...	...	...
301	2	実技試験	88.0	100.0	2025-05-18
...	...	...	...	...	...

NULLの扱い

ORDER BYでNULL値を並び替える場合、データベース製品によって動作が異なります。多くのデータベースでは、NULL値は最小値または最大値として扱われます。

- MySQL/MariaDBでは、NULL値は昇順（ASC）の場合は最小値として（最初に表示）、降順（DESC）の場合は最大値として（最後に表示）扱われます。

一部のデータベース（PostgreSQLなど）では、NULL値の位置を明示的に指定するための「NULLS FIRST」「NULLS LAST」構文がサポートされています。

例5：NULL値の扱い

例えば、出席（attendance）テーブルからコメント（comment）でソートすると、NULLが最初に来ます：

```
SELECT * FROM attendance ORDER BY comment;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	NULL
1	306	present	NULL
...	...	...	NULL
1	308	late	5分遅刻
1	323	late	電車遅延
...	...	...	...

カラム番号を使った並び替え

カラム名の代わりに、SELECT文の結果セットにおけるカラムの位置（番号）を使って並び替えることもできます。最初のカラムは1、2番目のカラムは2、という具合です。

構文

```
SELECT カラム名1, カラム名2, ... FROM テーブル名 ORDER BY カラム位置;
```

例6：カラム番号を使った並び替え

例えば、学生（students）テーブルから学生ID（student_id）と名前（student_name）を取得し、名前（2番目のカラム）で並べ替えるには：

```
SELECT student_id, student_name FROM students ORDER BY 2;
```

この場合、「ORDER BY 2」は「ORDER BY student_name」と同じ意味になります。

> **注意**：カラム番号を使う方法は、カラムの順序を変更すると問題が起きるため、実際の業務では使用を避けた方が良いとされています。

式や関数を使った並び替え

ORDER BY句で式や関数を使うことにより、計算結果に基づいて並び替えることもできます。

例7：式を使った並び替え

例えば、成績（grades）テーブルから、得点の達成率（score/max_score）の高い順に並べるには：

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score/max_score)*100 AS 達成率
FROM grades
ORDER BY (score/max_score) DESC;
```

または

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score/max_score)*100 AS 達成率
FROM grades
ORDER BY 達成率 DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	達成率
311	1	レポート1	49.0	50.0	98.0
320	1	レポート1	48.5	50.0	97.0
311	1	中間テスト	95.0	100.0	95.0
…	…	…	…	…	…

例8：関数を使った並び替え

文字列関数を使って並び替えることもできます。例えば、月名で並べることを考えましょう：

```
SELECT schedule_date, MONTH(schedule_date) AS month
FROM course_schedule
ORDER BY MONTH(schedule_date);
```

実行結果：

schedule_date	month
2025-04-07	4
2025-04-08	4
…	…
2025-05-01	5
2025-05-02	5
…	…
2025-06-01	6
…	…

CASE式を使った条件付き並び替え

さらに高度な並び替えとして、CASE式を使って条件に応じた並び順を定義することもできます。

例9：CASE式を使った並び替え

例えば、出席（attendance）テーブルから、出席状況（status）を「欠席→遅刻→出席」の順に優先して表示するには：

```
SELECT * FROM attendance
ORDER BY CASE
           WHEN status = 'absent' THEN 1
           WHEN status = 'late' THEN 2
           WHEN status = 'present' THEN 3
           ELSE 4
         END;
```

実行結果：

schedule_id	student_id	status	comment
1	303	absent	事前連絡あり
1	317	absent	体調不良

schedule_id	student_id	status	comment
...	...	...	...
1	302	late	15分遅刻
1	308	late	5分遅刻
...	...	...	...
1	301	present	NULL
1	306	present	NULL
...	...	...	...

練習問題

問題7-1

students（学生）テーブルから、すべての学生情報を学生名（student_name）の降順（逆五十音順）で取得するSQLを書いてください。

問題7-2

grades（成績）テーブルから、得点（score）が85点以上の成績を得点の高い順に取得するSQLを書いてください。

問題7-3

course_schedule（授業カレンダー）テーブルから、2025年5月の授業スケジュールを日付（schedule_date）の昇順で取得するSQLを書いてください。

問題7-4

teachers（教師）テーブルから、教師IDと名前を取得し、名前（teacher_name）の五十音順で並べるSQLを書いてください。

問題7-5

grades（成績）テーブルから、講座ID（course_id）ごとに、成績を評価タイプ（grade_type）の五十音順に、同じ評価タイプ内では得点（score）の高い順に並べて取得するSQLを書いてください。

問題7-6

attendance（出席）テーブルから、すべての出席情報を出席状況（status）が「absent」「late」「present」の順番で、同じ状態内ではコメント（comment）の有無（NULLが後）で並べて取得するSQLを書いてください。

解答

解答7-1

```
SELECT * FROM students ORDER BY student_name DESC;
```

解答7-2

```
SELECT * FROM grades WHERE score >= 85 ORDER BY score DESC;
```

解答7-3

```
SELECT * FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31'
ORDER BY schedule_date;
```

解答7-4

```
SELECT teacher_id, teacher_name FROM teachers ORDER BY teacher_name;
```

解答7-5

```
SELECT * FROM grades
ORDER BY course_id, grade_type, score DESC;
```

解答7-6

```
SELECT * FROM attendance
ORDER BY
  CASE
    WHEN status = 'absent' THEN 1
    WHEN status = 'late' THEN 2
    WHEN status = 'present' THEN 3
    ELSE 4
  END,
  CASE
    WHEN comment IS NULL THEN 2
    ELSE 1
  END;
```

まとめ

この章では、クエリ結果を特定の順序で並べるための「ORDER BY」句について学びました：

1. **基本的な並び替え**：指定したカラムの値に基づいて結果を並べる方法
2. **昇順と降順**：ASC（昇順）とDESC（降順）の指定方法
3. **複数カラムでの並び替え**：優先順位の高いカラムから順に指定する方法
4. **NULL値の扱い**：NULL値が並び替えでどのように扱われるか
5. **カラム番号**：カラム名の代わりに位置で指定する方法（あまり推奨されない）
6. **式や関数**：計算結果に基づいて並べる方法
7. **CASE式**：条件付きの複雑な並び替え

ORDER BY句は、データを見やすく整理するために非常に重要です。特に大量のデータを扱う場合、適切な並び順はデータの理解を大きく助けます。

次の章では、取得する結果の件数を制限する「LIMIT句：結果件数の制限とページネーション」について学びます。

---

8. LIMIT句：結果件数の制限とページネーション

はじめに

これまでの章では、条件に合うデータを取得し、それを特定の順序で並べる方法を学びました。しかし実際のアプリケーションでは、大量のデータがあるときに、その一部だけを表示したいことがよくあります。例えば：

- 成績上位10件だけを表示したい
- Webページで一度に20件ずつ表示したい（ページネーション）
- 最新の5件のお知らせだけを取得したい

このような「結果の件数を制限する」ためのSQLコマンドが「LIMIT句」です。この章では、クエリ結果の件数を制限する方法と、ページネーションの実装方法を学びます。

LIMIT句の基本

LIMIT句は、SELECT文の結果から指定した件数だけを取得するために使います。

> **用語解説**：
>
> - **LIMIT**：「制限する」という意味のSQLコマンドで、取得する行数を制限します。

基本構文（MySQL/MariaDB）

MySQLやMariaDBでのLIMIT句の基本構文は次のとおりです：

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT 件数;
```

LIMIT句は通常、SELECT文の最後に記述します（ORDER BYの後）。

例1：単純なLIMIT

例えば、学生（students）テーブルから最初の5人だけを取得するには：

```
SELECT * FROM students LIMIT 5;
```

実行結果：

student_id	student_name
301	黒沢春馬
302	新垣愛留
303	柴崎春花
304	森下風凛
305	河口菜恵子

ORDER BYとLIMITの組み合わせ

通常、LIMIT句はORDER BY句と組み合わせて使用します。これにより、「上位N件」「最新N件」などの操作が可能になります。

例2：ORDER BYとLIMITの組み合わせ

例えば、成績（grades）テーブルから得点（score）の高い順に上位3件を取得するには：

```
SELECT * FROM grades ORDER BY score DESC LIMIT 3;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20

例3：最新のレコードを取得

日付でソートして最新のデータを取得することもよくあります。例えば、最新の3つの授業スケジュールを取得するには：

```
SELECT * FROM course_schedule
ORDER BY schedule_date DESC LIMIT 3;
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
95	28	2026-12-21	3	202D	119	scheduled
94	1	2026-12-21	1	102B	101	scheduled
93	14	2026-12-15	4	202D	110	scheduled

OFFSETとページネーション

Webアプリケーションなどでは、大量のデータを「ページ」に分けて表示することがよくあります（ページネーション）。この機能を実現するためには、「OFFSET」（オフセット）という機能が必要です。

> **用語解説**：
>
> - **OFFSET**：「ずらす」という意味で、結果セットの先頭から指定した数だけ行をスキップします。
> - **ページネーション**：大量のデータを複数のページに分割して表示する技術です。

基本構文（MySQL/MariaDB）

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT 件数 OFFSET スキップ数;
```

または、短縮形として：

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT スキップ数, 件数;
```

例4：OFFSETを使ったスキップ

例えば、学生（students）テーブルから6番目から10番目までの学生を取得するには：

```
SELECT * FROM students LIMIT 5 OFFSET 5;
```

または：

```
SELECT * FROM students LIMIT 5, 5;
```

実行結果：

student_id	student_name
306	河田咲奈
307	織田柚夏

student_id	student_name
308	永田悦子
309	相沢吉夫
310	吉川伽羅

例5：ページネーションの実装

ページネーションを実装する場合、通常は以下の式を使ってOFFSETを計算します：

```
OFFSET = (ページ番号 - 1) × ページあたりの件数
```

例えば、1ページあたり10件表示で、3ページ目のデータを取得するには：

```sql
SELECT * FROM students ORDER BY student_id LIMIT 10 OFFSET 20;
```

または：

```sql
SELECT * FROM students ORDER BY student_id LIMIT 20, 10;
```

実行結果：

student_id	student_name
321	井上竜也
322	木村結衣
323	林正義
324	清水香織
325	山田翔太
326	葉山陽太
327	青山凛
328	沢村大和
329	白石優月
330	月岡星奈

LIMIT句を使用する際の注意点

1. ORDER BYの重要性

LIMIT句を使用する場合、通常はORDER BY句も一緒に使うべきです。ORDER BYがなければ、どのレコードが取得されるかは保証されません。

```
-- 良い例：結果が予測可能
SELECT * FROM students ORDER BY student_id LIMIT 5;

-- 悪い例：結果が不確定
SELECT * FROM students LIMIT 5;
```

2. パフォーマンスへの影響

大規模なテーブルで大きなOFFSET値を使用すると、パフォーマンスが低下する可能性があります。これは、データベースがOFFSET分のレコードを読み込んでから破棄する必要があるためです。

3. データベース製品による構文の違い

LIMIT句の構文はデータベース製品によって異なります：

- **MySQL/MariaDB/SQLite**：`LIMIT 件数 OFFSET スキップ数` または `LIMIT スキップ数, 件数`
- **PostgreSQL**：`LIMIT 件数 OFFSET スキップ数`
- **Oracle**：`OFFSET スキップ数 ROWS FETCH NEXT 件数 ROWS ONLY`
- **SQL Server**：`OFFSET スキップ数 ROWS FETCH NEXT 件数 ROWS ONLY` または旧バージョンでは `TOP`句

この章では主にMySQL/MariaDBの構文を使用します。

実践的なページネーションの実装

実際のアプリケーションでページネーションを実装する場合、以下のようなコードになります（疑似コード）：

```
ページ番号 = URLから取得またはデフォルト値（例：1）
1ページあたりの件数 = 設定値（例：10）
総レコード数 = SELECTで取得（COUNT(*)を使用）
総ページ数 = CEILING(総レコード数 ÷ 1ページあたりの件数)
OFFSET = (ページ番号 - 1) × 1ページあたりの件数

SQLクエリ = "SELECT * FROM テーブル ORDER BY カラム LIMIT " + 1ページあたりの件数 + " OFFSET
" + OFFSET
```

例6：総レコード数と総ページ数の取得

総レコード数を取得するには：

```
SELECT COUNT(*) AS total_records FROM students;
```

実行結果：

total_records
100

この場合、1ページあたり10件表示なら、総ページ数は10ページ（CEILING(100 ÷ 10)）になります。

練習問題

問題8-1

grades（成績）テーブルから、得点（score）の高い順に上位5件の成績レコードを取得するSQLを書いてください。

問題8-2

course_schedule（授業カレンダー）テーブルから、日付（schedule_date）の新しい順に3件のスケジュールを取得するSQLを書いてください。

問題8-3

students（学生）テーブルを学生ID（student_id）の昇順で並べ、11番目から15番目までの学生（5件）を取得するSQLを書いてください。

問題8-4

teachers（教師）テーブルから、教師名（teacher_name）の五十音順で6番目から10番目までの教師情報を取得するSQLを書いてください。

問題8-5

1ページあたり20件表示で、grades（成績）テーブルの3ページ目のデータを得点（score）の高い順に取得するSQLを書いてください。

問題8-6

course_schedule（授業カレンダー）テーブルから、状態（status）が「scheduled」のスケジュールを日付（schedule_date）の昇順で並べ、先頭から10件スキップして次の5件を取得するSQLを書いてください。

解答

解答8-1

```
SELECT * FROM grades ORDER BY score DESC LIMIT 5;
```

解答8-2

```
SELECT * FROM course_schedule ORDER BY schedule_date DESC LIMIT 3;
```

解答8-3

```
SELECT * FROM students ORDER BY student_id LIMIT 5 OFFSET 10;
```

または

```
SELECT * FROM students ORDER BY student_id LIMIT 10, 5;
```

解答8-4

```
SELECT * FROM teachers ORDER BY teacher_name LIMIT 5 OFFSET 5;
```

または

```
SELECT * FROM teachers ORDER BY teacher_name LIMIT 5, 5;
```

解答8-5

```
SELECT * FROM grades ORDER BY score DESC LIMIT 20 OFFSET 40;
```

または

```
SELECT * FROM grades ORDER BY score DESC LIMIT 40, 20;
```

解答8-6

```
SELECT * FROM course_schedule
WHERE status = 'scheduled'
ORDER BY schedule_date
LIMIT 5 OFFSET 10;
```

または

```
SELECT * FROM course_schedule
WHERE status = 'scheduled'
ORDER BY schedule_date
LIMIT 10, 5;
```

まとめ

この章では、クエリ結果の件数を制限するためのLIMIT句と、ページネーションの実装方法について学びました：

1. **LIMIT句の基本**：指定した件数だけのレコードを取得する方法
2. **ORDER BYとの組み合わせ**：順序付けされた結果から一部だけを取得する方法
3. **OFFSET**：結果の先頭から指定した数だけレコードをスキップする方法
4. **ページネーション**：大量のデータを複数のページに分けて表示する実装方法
5. **注意点**：LIMIT句を使用する際の留意事項
6. **データベース製品による違い**：異なるデータベースでの構文の違い

LIMIT句は特にWebアプリケーションの開発で重要な機能です。大量のデータを効率よく表示するためのページネーション機能を実装するために欠かせません。また、トップN（上位N件）やボトムN（下位N件）のデータを取得する際にも使われます。

次の章では、データの集計分析を行うための「集計関数：COUNT、SUM、AVG、MAX、MIN」について学びます。

SQL学習テキスト 完全版

このテキストは、SQLの基礎から応用まで体系的に学習できるように構成されています。 学校データベースを使った実践的な例題と練習問題を通じて、実務で使えるSQLスキルを身につけることができます。

目次

---

9. 集計関数：COUNT、SUM、AVG、MAX、MIN

はじめに

これまでの章では、データベースからレコードを取得して表示する方法を学んできました。しかし、データベースを使う目的の一つは、大量のデータから有用な情報（集計、統計など）を抽出することです。例えば：

- 「学生の総数は何人か？」
- 「成績の平均点は？」
- 「最高点と最低点は？」

- 「全講座の合計受講者数は？」

このような「データの集計」を行うためのSQLの機能が「集計関数」です。この章では、最もよく使われる5つの集計関数（COUNT、SUM、AVG、MAX、MIN）について学びます。

集計関数の基本

集計関数は、複数の行のデータをまとめて一つの値を返す関数です。

> **用語解説**：
>
> - **集計関数**：複数の行（レコード）から計算された単一の値を返す関数です。データの集計や統計に使用されます。

主な集計関数

関数	説明	例
COUNT	レコード数を数える	学生の総数
SUM	値の合計を計算する	全成績の点数合計
AVG	値の平均を計算する	平均点
MAX	最大値を取得する	最高点
MIN	最小値を取得する	最低点

COUNT関数：レコード数をカウントする

COUNT関数は、条件に一致するレコードの数を数えるのに使用します。

基本構文

```
SELECT COUNT(カラム名) FROM テーブル名 [WHERE 条件];
```

または、すべての行を数える場合：

```
SELECT COUNT(*) FROM テーブル名 [WHERE 条件];
```

例1：テーブル内の全レコード数

例えば、学生（students）テーブルの全学生数を数えるには：

```
SELECT COUNT(*) AS 学生総数 FROM students;
```

実行結果：

学生総数
100

例2：条件付きのカウント

WHERE句と組み合わせることで、条件に一致するレコードだけをカウントできます。例えば、「出席（present）」状態の出席レコードの数を数えるには：

```
SELECT COUNT(*) AS 出席数 FROM attendance WHERE status = 'present';
```

実行結果：

出席数
42

例3：NULL以外の値をカウント

COUNT(カラム名)を使うと、そのカラムがNULLでない行だけがカウントされます。これは、COUNT(*)との大きな違いです。例えば、コメント（comment）が入力されている出席レコードの数を数えるには：

```
SELECT COUNT(comment) AS コメントあり FROM attendance;
```

実行結果：

コメントあり
23

例4：DISTINCTを使った重複のないカウント

DISTINCTを使うと、重複を除外してカウントできます。例えば、何種類の評価タイプ（grade_type）があるかを数えるには：

```
SELECT COUNT(DISTINCT grade_type) AS 評価タイプ数 FROM grades;
```

実行結果：

評価タイプ数
3

SUM関数：合計を計算する

SUM関数は、数値カラムの合計を計算するのに使用します。

基本構文

```
SELECT SUM(数値カラム) FROM テーブル名 [WHERE 条件];
```

例1：単純な合計

例えば、全成績レコードの得点（score）の合計を計算するには：

```
SELECT SUM(score) AS 総得点 FROM grades;
```

実行結果：

総得点
3542.5

例2：条件付きの合計

WHERE句と組み合わせることで、条件に一致するレコードだけの合計を計算できます。例えば、「中間テスト」の得点合計を計算するには：

```
SELECT SUM(score) AS 中間テスト合計点
FROM grades
WHERE grade_type = '中間テスト';
```

実行結果：

中間テスト合計点
1728.0

例3：計算式を使った合計

計算式と組み合わせることもできます。例えば、全成績の達成率（score/max_score）の合計を計算するには：

```
SELECT SUM(score/max_score) AS 達成率合計 FROM grades;
```

実行結果：

達成率合計
39.86

AVG関数：平均を計算する

AVG関数は、数値カラムの平均値を計算するのに使用します。

基本構文

```
SELECT AVG(数値カラム) FROM テーブル名 [WHERE 条件];
```

例1：単純な平均

例えば、全成績レコードの得点（score）の平均を計算するには：

```
SELECT AVG(score) AS 平均点 FROM grades;
```

実行結果：

平均点
82.38

例2：条件付きの平均

WHERE句と組み合わせることで、条件に一致するレコードだけの平均を計算できます。例えば、「実技試験」の平均点を計算するには：

```
SELECT AVG(score) AS 実技試験平均点
FROM grades
WHERE grade_type = '実技試験';
```

実行結果：

実技試験平均点
85.6

例3：達成率（%）の計算

計算式と組み合わせることで、例えば平均達成率（%）を計算できます：

```
SELECT AVG(score/max_score * 100) AS 平均達成率 FROM grades;
```

実行結果：

平均達成率

平均達成率
87.24

MAX関数：最大値を取得する

MAX関数は、数値、文字列、日付などのカラムから最大値を取得するのに使用します。

基本構文

```
SELECT MAX(カラム名) FROM テーブル名 [WHERE 条件];
```

例1：数値の最大値

例えば、全成績レコードの中での最高点を取得するには：

```
SELECT MAX(score) AS 最高点 FROM grades;
```

実行結果：

最高点
95.0

例2：文字列の最大値（辞書順で最後）

MAX関数は文字列にも使用でき、この場合は辞書順で「最後」の値を返します。例えば、学生名の辞書順で最後の値（最も「わ行」に近い名前）を取得するには：

```
SELECT MAX(student_name) AS 学生名最終 FROM students;
```

実行結果：

学生名最終
吉川伽羅

例3：日付の最大値（最新の日付）

日付にMAX関数を使うと、最新（最も未来）の日付が取得できます。例えば、最も新しい提出日を取得するには：

```
SELECT MAX(submission_date) AS 最新提出日 FROM grades;
```

実行結果：

最新提出日
2025-05-20

MIN関数：最小値を取得する

MIN関数は、数値、文字列、日付などのカラムから最小値を取得するのに使用します。

基本構文

```
SELECT MIN(カラム名) FROM テーブル名 [WHERE 条件];
```

例1：数値の最小値

例えば、全成績レコードの中での最低点を取得するには：

```
SELECT MIN(score) AS 最低点 FROM grades;
```

実行結果：

最低点
68.0

例2：文字列の最小値（辞書順で最初）

MIN関数は文字列にも使用でき、この場合は辞書順で「最初」の値を返します。例えば、学生名の辞書順で最初の値（最も「あ行」に近い名前）を取得するには：

```
SELECT MIN(student_name) AS 学生名最初 FROM students;
```

実行結果：

学生名最初
相沢吉夫

例3：日付の最小値（最古の日付）

日付にMIN関数を使うと、最も古い日付が取得できます。例えば、最も古い提出日を取得するには：

```
SELECT MIN(submission_date) AS 最古提出日 FROM grades;
```

実行結果：

最古提出日
2025-05-06

複数の集計関数の組み合わせ

複数の集計関数を一つのクエリで使用することもできます。

例：成績の統計情報

例えば、成績（grades）テーブルの統計情報を一度に取得するには：

```
SELECT
    COUNT(*) AS レコード数,
    AVG(score) AS 平均点,
    SUM(score) AS 合計点,
    MAX(score) AS 最高点,
    MIN(score) AS 最低点
FROM grades;
```

実行結果：

レコード数	平均点	合計点	最高点	最低点
43	82.38	3542.5	95.0	68.0

集計関数とGROUP BY（次章の内容）

集計関数をより強力に使うためには、「GROUP BY」句と組み合わせます。これにより、データをグループ化して各グループごとに集計できます。GROUP BYについては次章で詳しく学びます。

例えば、講座（course_id）ごとの平均点を計算するには：

```
SELECT course_id, AVG(score) AS 平均点
FROM grades
GROUP BY course_id;
```

実行結果：

course_id	平均点
1	86.2
2	83.8
3	80.5

course_id	平均点
...	...

この例の詳細な説明は次章で行います。

練習問題

問題9-1

students（学生）テーブルから、学生の総数を取得するSQLを書いてください。

問題9-2

grades（成績）テーブルから、全成績の平均点（score）を取得するSQLを書いてください。

問題9-3

attendance（出席）テーブルから、出席状況（status）が「absent」のレコード数を取得するSQLを書いてください。

問題9-4

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」の成績の最高点と最低点を取得するSQLを書いてください。

問題9-5

course_schedule（授業カレンダー）テーブルから、最新（schedule_dateが最大）の授業スケジュールの日付を取得するSQLを書いてください。

問題9-6

grades（成績）テーブルから、成績の総数、平均点、合計点、最高点、最低点を一度に取得するSQLを書いてください。

解答

解答9-1

```
SELECT COUNT(*) AS 学生総数 FROM students;
```

解答9-2

```
SELECT AVG(score) AS 全成績平均点 FROM grades;
```

解答9-3

```
SELECT COUNT(*) AS 欠席数 FROM attendance WHERE status = 'absent';
```

解答9-4

```
SELECT
    MAX(score) AS 中間テスト最高点,
    MIN(score) AS 中間テスト最低点
FROM grades
WHERE grade_type = '中間テスト';
```

解答9-5

```
SELECT MAX(schedule_date) AS 最新授業日 FROM course_schedule;
```

解答9-6

```
SELECT
    COUNT(*) AS 成績総数,
    AVG(score) AS 平均点,
    SUM(score) AS 合計点,
    MAX(score) AS 最高点,
    MIN(score) AS 最低点
FROM grades;
```

まとめ

この章では、データの集計と分析に使用される主要な集計関数について学びました：

1. **COUNT**：レコード数をカウントする関数
2. **SUM**：数値の合計を計算する関数
3. **AVG**：数値の平均を計算する関数
4. **MAX**：最大値（数値、文字列、日付など）を取得する関数
5. **MIN**：最小値（数値、文字列、日付など）を取得する関数

これらの集計関数を使うことで、大量のデータから意味のある統計情報を簡単に抽出できます。ビジネスにおける意思決定や、データ分析において非常に重要な機能です。

次の章では、データをグループ化して集計を行うための「GROUP BY：データのグループ化」について学びます。

---

10. GROUP BY：データのグループ化

はじめに

前章では、集計関数（COUNT、SUM、AVG、MAX、MIN）を使って全体的な集計や統計を計算する方法を学びました。しかし実際のデータ分析では、全体の集計だけでなく、特定のカテゴリや条件ごとに集計したい場合が多くあります。例えば：

- 「講座ごとの平均点は？」
- 「教師ごとの担当講座数は？」
- 「日付ごとの授業数は？」
- 「出席状況（出席/遅刻/欠席）の割合は？」

このような「グループごとの集計」を行うためのSQLコマンドが「GROUP BY」です。この章では、データをグループ化して集計する方法について学びます。

GROUP BYの基本

GROUP BY句は、指定したカラムの値が同じレコードをグループ化し、それぞれのグループに対して集計関数を適用するために使います。

> **用語解説**：
>
> - **GROUP BY**：「～でグループ化する」という意味のSQLコマンドで、同じ値を持つレコードをまとめてグループにします。
> - **グループ化**：データを特定の条件で分類し、それぞれの分類ごとに集計を行うこと。

基本構文

```
SELECT カラム名, 集計関数
FROM テーブル名
[WHERE 条件]
GROUP BY グループ化するカラム名;
```

重要なのは、SELECT句に含めるカラムは、GROUP BY句で指定したカラムか、集計関数（COUNT、SUM、AVG、MAX、MINなど）のいずれかでなければならないということです。

例1：単純なグループ化

例えば、成績（grades）テーブルから、評価タイプ（grade_type）ごとの成績レコード数を集計するには：

```
SELECT grade_type, COUNT(*) AS レコード数
FROM grades
GROUP BY grade_type;
```

実行結果：

grade_type	レコード数

grade_type	レコード数
中間テスト	12
レポート1	22
実技試験	9

例2：グループごとの平均

グループごとの平均を計算することも一般的です。例えば、各評価タイプごとの平均点を計算するには：

```
SELECT grade_type, AVG(score) AS 平均点
FROM grades
GROUP BY grade_type;
```

実行結果：

grade_type	平均点
中間テスト	86.25
レポート1	44.77
実技試験	85.61

複数のカラムによるグループ化

複数のカラムを指定してグループ化することもできます。この場合、指定したすべてのカラムの値の組み合わせがグループの単位になります。

例3：複数カラムでのグループ化

例えば、講座（course_id）と評価タイプ（grade_type）の組み合わせごとに成績の平均を計算するには：

```
SELECT course_id, grade_type, AVG(score) AS 平均点
FROM grades
GROUP BY course_id, grade_type;
```

実行結果：

course_id	grade_type	平均点
1	中間テスト	87.33
1	レポート1	45.39
2	実技試験	86.83
…	…	…

グループ化の順序

GROUP BY句でカラムを指定する順序は、結果には影響しません。しかし、可読性のためにSELECT句と同じ順序で指定することが一般的です。

WHERE句とGROUP BYの組み合わせ

WHERE句は、グループ化する前に行単位でレコードを絞り込むのに使います。

> **用語解説：**
>
> - **絞り込み順序**：WHERE句はGROUP BY句の前に評価され、条件に合うレコードだけがグループ化の対象になります。

例4：WHEREとGROUP BYの組み合わせ

例えば、得点（score）が80以上の成績だけを対象に、評価タイプごとの平均を計算するには：

```
SELECT grade_type, AVG(score) AS 平均点
FROM grades
WHERE score >= 80
GROUP BY grade_type;
```

実行結果：

grade_type	平均点
中間テスト	88.92
実技試験	87.25

この例では、score >= 80の条件に一致するレコードだけがグループ化され、集計されています。レポート1はすべての点数が80未満なので、結果には表示されていません。

GROUP BYとORDER BYの組み合わせ

GROUP BY句とORDER BY句を組み合わせることで、グループ化した結果を任意の順序で並べることができます。

例5：GROUP BYとORDER BYの組み合わせ

例えば、講座（course_id）ごとの平均点を計算し、平均点の高い順に並べるには：

```
SELECT course_id, AVG(score) AS 平均点
FROM grades
GROUP BY course_id
ORDER BY 平均点 DESC;
```

実行結果：

course_id	平均点
1	86.21
2	83.79
5	82.33
...	...

集計関数の複数使用

一つのクエリで複数の集計関数を使用することもできます。

例6：複数の集計関数の使用

例えば、評価タイプごとの成績数、平均点、最高点、最低点を一度に取得するには：

```
SELECT grade_type,
       COUNT(*) AS 成績数,
       AVG(score) AS 平均点,
       MAX(score) AS 最高点,
       MIN(score) AS 最低点
FROM grades
GROUP BY grade_type;
```

実行結果：

grade_type	成績数	平均点	最高点	最低点
中間テスト	12	86.25	95.0	78.5
レポート1	22	44.77	49.0	37.0
実技試験	9	85.61	88.0	79.5

計算式を含むグループ化

GROUP BY句で集計した結果に対して、さらに計算を加えることができます。

例7：計算式を含むグループ化

例えば、評価タイプごとに平均達成率（score/max_score）を計算するには：

```
SELECT grade_type,
       AVG(score/max_score * 100) AS 平均達成率
FROM grades
GROUP BY grade_type;
```

実行結果：

grade_type	平均達成率
中間テスト	86.25
レポート1	89.54
実技試験	85.61

GROUP BYとNULLの扱い

GROUP BY句でグループ化する際、NULL値も一つのグループとして扱われます。

例8：NULLを含むグループ化

例えば、出席（attendance）テーブルから、コメント（comment）の有無でグループ化し、それぞれの件数を数えるには：

```
SELECT
    CASE
        WHEN comment IS NULL THEN 'コメントなし'
        ELSE 'コメントあり'
    END AS コメント状態,
    COUNT(*) AS 件数
FROM attendance
GROUP BY
    CASE
        WHEN comment IS NULL THEN 'コメントなし'
        ELSE 'コメントあり'
    END;
```

実行結果：

コメント状態	件数
コメントなし	20
コメントあり	23

HAVINGを使ったグループの絞り込み（次章の内容）

グループ化した後にさらに条件でグループを絞り込むには、「HAVING」句を使います。これについては次章で詳しく学びます。

例えば、5人以上の学生が受講している講座だけを取得するには：

```
SELECT course_id, COUNT(student_id) AS 学生数
FROM student_courses
GROUP BY course_id
HAVING COUNT(student_id) >= 5;
```

この例の詳細な説明は次章で行います。

練習問題

問題10-1

courses（講座）テーブルから、教師ID（teacher_id）ごとに担当している講座の数を取得するSQLを書いてください。

問題10-2

attendance（出席）テーブルから、出席状況（status）ごとのレコード数を取得するSQLを書いてください。

問題10-3

grades（成績）テーブルから、学生ID（student_id）ごとの平均点を計算し、平均点の高い順に並べるSQLを書いてください。

問題10-4

course_schedule（授業カレンダー）テーブルから、教室ID（classroom_id）ごとに予定されている授業の数を取得するSQLを書いてください。

問題10-5

grades（成績）テーブルから、講座ID（course_id）と評価タイプ（grade_type）の組み合わせごとの平均点を計算し、講座ID順、さらに評価タイプ順で並べるSQLを書いてください。

問題10-6

student_courses（受講）テーブルから、講座ID（course_id）ごとの受講者数を取得し、受講者数の多い順に並べるSQLを書いてください。

解答

解答10-1

```
SELECT teacher_id, COUNT(course_id) AS 担当講座数
FROM courses
GROUP BY teacher_id;
```

解答10-2

```
SELECT status, COUNT(*) AS レコード数
FROM attendance
GROUP BY status;
```

解答10-3

```
SELECT student_id, AVG(score) AS 平均点
FROM grades
GROUP BY student_id
ORDER BY 平均点 DESC;
```

解答10-4

```
SELECT classroom_id, COUNT(*) AS 授業数
FROM course_schedule
GROUP BY classroom_id;
```

解答10-5

```
SELECT course_id, grade_type, AVG(score) AS 平均点
FROM grades
GROUP BY course_id, grade_type
ORDER BY course_id, grade_type;
```

解答10-6

```
SELECT course_id, COUNT(student_id) AS 受講者数
FROM student_courses
GROUP BY course_id
ORDER BY 受講者数 DESC;
```

まとめ

この章では、データをグループ化して集計するためのGROUP BY句について学びました：

1. **GROUP BYの基本**：同じ値を持つレコードをグループ化する方法
2. **集計関数との組み合わせ**：グループごとの集計を行う方法
3. **複数カラムによるグループ化**：複数の条件でのグループ化
4. **WHERE句との組み合わせ**：グループ化前のレコード絞り込み
5. **ORDER BYとの組み合わせ**：グループ化結果の並び替え
6. **複数の集計関数の使用**：一度に複数の統計値を取得する方法
7. **計算式を含むグループ化**：より複雑な集計の実現
8. **NULLの扱い**：グループ化におけるNULL値の取り扱い

GROUP BY句を使うことで、データのさまざまな側面から分析が可能になり、意思決定のための有用な情報を抽出できるようになります。

次の章では、グループ化した結果をさらに条件で絞り込むための「HAVING：グループ化後の絞り込み」について学びます。

---

11. HAVING：グループ化後の絞り込み

はじめに

前章では、GROUP BY句を使ってデータをグループ化し、各グループごとに集計を行う方法を学びました。しかし、すべてのグループの集計結果を表示するのではなく、特定の条件を満たすグループだけを表示したい場合があります。例えば：

- 「平均点が80点以上の講座だけを表示したい」
- 「5人以上の学生が登録している講座だけを取得したい」
- 「出席率が90%を超える学生だけをリストアップしたい」

このような「グループ化した結果をさらに絞り込む」ためのSQLコマンドが「HAVING」句です。この章では、グループ化した結果に条件を適用する方法について学びます。

HAVINGの基本

HAVING句は、GROUP BY句でグループ化した後の結果に対して条件を適用し、条件を満たすグループだけを取得するために使います。

> **用語解説**：
>
> - **HAVING**：「～を持っている」という意味のSQLコマンドで、グループ化した結果に対して条件を適用します。

基本構文

```
SELECT カラム名, 集計関数
FROM テーブル名
[WHERE 行レベルの条件]
GROUP BY グループ化するカラム名
HAVING グループレベルの条件;
```

例1：基本的なHAVINGの使用

例えば、受講者数が5人以上の講座だけを取得するには：

```
SELECT course_id, COUNT(student_id) AS 受講者数
FROM student_courses
GROUP BY course_id
HAVING COUNT(student_id) >= 5;
```

実行結果：

course_id	受講者数
1	12
2	8
4	7
5	7
...	...

この例では、まず講座ID（course_id）ごとに学生ID（student_id）の数を数え、その後でHAVING句を使って受講者数が5人以上のグループだけを抽出しています。

WHEREとHAVINGの違い

WHERE句とHAVING句は似ていますが、適用されるタイミングと対象が異なります：

- **WHERE**：グループ化される前の個々の行（レコード）に対して条件を適用します。
- **HAVING**：グループ化された後のグループに対して条件を適用します。

> **用語解説**：
>
> - **行レベルのフィルタリング**：WHEREによる個々のレコードの絞り込み
> - **グループレベルのフィルタリング**：HAVINGによるグループの絞り込み

例2：WHEREとHAVINGの違い

以下の例で違いを確認しましょう：

```
-- WHERE（行レベル）：まず80点以上の成績だけを選び、それからグループ化
SELECT course_id, AVG(score) AS 平均点
FROM grades
WHERE score >= 80
GROUP BY course_id;

-- HAVING（グループレベル）：まずグループ化して平均点を計算し、それから平均点が80以上のグループを選ぶ
SELECT course_id, AVG(score) AS 平均点
FROM grades
GROUP BY course_id
HAVING AVG(score) >= 80;
```

1つ目のクエリは、「80点以上の成績だけ」を対象に講座ごとの平均点を計算します。 2つ目のクエリは、「講座の平均点が80点以上」の講座だけを取得します。

WHERE句の結果：

course_id	平均点
1	88.92

course_id	平均点
2	87.25
...	...

HAVING句の結果：

course_id	平均点
1	86.21
2	83.79
5	82.33
...	...

HAVINGで使用できる条件

HAVING句では、集計関数（COUNT、SUM、AVG、MAX、MINなど）を使った条件や、GROUP BY句で指定したカラムに対する条件を指定できます。

例3：集計関数を使った条件

例えば、平均点が85点以上の講座を取得するには：

```
SELECT course_id, AVG(score) AS 平均点
FROM grades
GROUP BY course_id
HAVING AVG(score) >= 85;
```

実行結果：

course_id	平均点
1	86.21
...	...

例4：複数条件の指定

HAVING句も、WHERE句と同様に複数の条件を組み合わせることができます：

```
SELECT grade_type, COUNT(*) AS 件数, AVG(score) AS 平均点
FROM grades
GROUP BY grade_type
HAVING COUNT(*) > 10 AND AVG(score) > 80;
```

実行結果：

grade_type	件数	平均点
中間テスト	12	86.25
...	...	...

WHEREとHAVINGの組み合わせ

実際のクエリでは、WHERE句とHAVING句を組み合わせることが多いです。WHERE句でグループ化前の行を絞り込み、HAVING句でグループ化後の結果をさらに絞り込みます。

例5：WHEREとHAVINGの組み合わせ

例えば、「中間テストのみを対象として、平均点が85点以上の講座」を取得するには：

```
SELECT course_id, AVG(score) AS 平均点
FROM grades
WHERE grade_type = '中間テスト'
GROUP BY course_id
HAVING AVG(score) >= 85;
```

実行結果：

course_id	平均点
1	87.33
...	...

HAVINGとORDER BYの組み合わせ

HAVING句で絞り込んだ結果を並べ替えるには、ORDER BY句を追加します。

例6：HAVINGとORDER BYの組み合わせ

例えば、受講者数が5人以上の講座を、受講者数の多い順に並べて取得するには：

```
SELECT course_id, COUNT(student_id) AS 受講者数
FROM student_courses
GROUP BY course_id
HAVING COUNT(student_id) >= 5
ORDER BY 受講者数 DESC;
```

実行結果：

course_id	受講者数
1	12

course_id	受講者数
20	11
9	10
…	…

実践的なHAVINGの使用例

例7：「平均達成率が90%を超える評価タイプ」の抽出

各評価タイプごとの平均達成率（score/max_score）を計算し、90%を超えるものだけを取得します：

```
SELECT grade_type,
       AVG(score/max_score * 100) AS 平均達成率
FROM grades
GROUP BY grade_type
HAVING AVG(score/max_score * 100) > 90;
```

実行結果：

grade_type	平均達成率
レポート1	93.54
…	…

例8：「特定の講座で成績データが存在する学生」の抽出

例えば、講座ID「1」の成績が2件以上ある学生を取得します：

```
SELECT student_id, COUNT(*) AS 成績件数
FROM grades
WHERE course_id = '1'
GROUP BY student_id
HAVING COUNT(*) >= 2;
```

実行結果：

student_id	成績件数
301	2
302	2
…	…

HAVINGでの注意点

1. **集計関数の使用**: HAVINGで使用する条件には、通常、集計関数（COUNT、SUM、AVG、MAX、MINなど）が含まれます。単純なカラム比較だけならWHERE句を使用すべきです。

2. **パフォーマンスの考慮**: WHERE句はグループ化前に適用されるため、処理対象のレコード数を減らすことができます。可能な限り、グループ化前の条件はWHERE句で指定し、グループ化後の条件だけをHAVING句で指定するようにすると、効率的なクエリになります。

3. **グループレベルの条件**: HAVING句は、GROUP BY句で指定したカラムや集計関数の結果に対する条件でなければならないことを覚えておきましょう。

練習問題

問題11-1

student_courses（受講）テーブルから、受講者数が8人以上の講座ID（course_id）を取得するSQLを書いてください。

問題11-2

grades（成績）テーブルから、平均点（score）が85点以上の評価タイプ（grade_type）を取得するSQLを書いてください。

問題11-3

courses（講座）テーブルから、各教師（teacher_id）が担当する講座数を計算し、担当講座が3つ以上の教師だけを取得するSQLを書いてください。

問題11-4

grades（成績）テーブルから、講座ID（course_id）ごとの最高点（score）を計算し、最高点が90点以上の講座を、最高点の高い順に並べて取得するSQLを書いてください。

問題11-5

attendance（出席）テーブルから、欠席（status = 'absent'）の回数が2回以上ある学生ID（student_id）を取得するSQLを書いてください。

問題11-6

grades（成績）テーブルを使って、中間テスト（grade_type = '中間テスト'）のデータのみを対象に、平均点が85点以上で、かつ最低点が75点以上の講座ID（course_id）を取得するSQLを書いてください。

解答

解答11-1

```
SELECT course_id, COUNT(student_id) AS 受講者数
FROM student_courses
GROUP BY course_id
HAVING COUNT(student_id) >= 8;
```

解答11-2

```
SELECT grade_type, AVG(score) AS 平均点
FROM grades
GROUP BY grade_type
HAVING AVG(score) >= 85;
```

解答11-3

```
SELECT teacher_id, COUNT(course_id) AS 担当講座数
FROM courses
GROUP BY teacher_id
HAVING COUNT(course_id) >= 3;
```

解答11-4

```
SELECT course_id, MAX(score) AS 最高点
FROM grades
GROUP BY course_id
HAVING MAX(score) >= 90
ORDER BY 最高点 DESC;
```

解答11-5

```
SELECT student_id, COUNT(*) AS 欠席回数
FROM attendance
WHERE status = 'absent'
GROUP BY student_id
HAVING COUNT(*) >= 2;
```

解答11-6

```
SELECT course_id, AVG(score) AS 平均点, MIN(score) AS 最低点
FROM grades
WHERE grade_type = '中間テスト'
GROUP BY course_id
HAVING AVG(score) >= 85 AND MIN(score) >= 75;
```

まとめ

この章では、グループ化した結果に条件を適用するためのHAVING句について学びました：

1. **HAVINGの基本**：グループ化した結果に条件を適用する方法
2. **WHEREとHAVINGの違い**：
   - WHERE：グループ化前の行レベルのフィルタリング
   - HAVING：グループ化後のグループレベルのフィルタリング
3. **HAVING句での条件**：集計関数を使った条件設定
4. **複合条件**：ANDやORを使った複数条件の組み合わせ
5. **WHERE、HAVING、ORDER BYの組み合わせ**：複雑なデータ抽出の手法
6. **実践的な使用例**：実際の業務で使われるようなクエリパターン

HAVING句はデータ分析において非常に重要です。グループ化されたデータに対して「どのグループが重要か」「どのグループに注目すべきか」という条件を設定することで、より意味のある情報を抽出することができます。

次の章では、重複データを除外するための「DISTINCT：重複の除外」について学びます。

---

12. DISTINCT：重複の除外

はじめに

データベースから情報を取得する際、同じ値が複数回出現することがよくあります。例えば、「どの講座がどの教室で行われているか」を調べると、同じ教室が複数回リストされるでしょう。しかし、時には重複を除いた一意の値のリストだけが必要な場合があります。例えば：

- 「学校にはどんな教室があるか」（重複なく知りたい）
- 「どの教師が授業を担当しているか」（重複なく知りたい）
- 「どのような評価タイプがあるか」（重複なく知りたい）

このような「重複を除外」するためのSQLキーワードが「DISTINCT」です。この章では、クエリ結果から重複するデータを除外する方法について学びます。

DISTINCTの基本

DISTINCT句は、SELECT文の結果から重複する行を除外するために使います。

> **用語解説**：
>
> - **DISTINCT**：「異なる」「区別される」という意味のSQLキーワードで、クエリ結果から重複する行を除外します。
> - **重複除外**：同じ値を持つレコードを1つだけにして、残りを除外すること。

基本構文

```
SELECT DISTINCT カラム名 FROM テーブル名 [WHERE 条件];
```

DISTINCTは、SELECT文の直後に配置され、すべての指定されたカラムの組み合わせに対して働きます。

例1：単一カラムでの重複除外

例えば、成績テーブル（grades）から、どのような評価タイプ（grade_type）があるかを重複なしで取得するには：

```
SELECT DISTINCT grade_type FROM grades;
```

実行結果：

grade_type
中間テスト
レポート1
実技試験

通常のSELECT文では、テーブル内の各行の評価タイプが表示されますが、DISTINCTを使うと、一意の評価タイプだけが表示されます。

例2：出席状況の種類の取得

出席（attendance）テーブルから、どのような出席状況（status）があるかを重複なしで取得するには：

```
SELECT DISTINCT status FROM attendance;
```

実行結果：

status
present
late
absent

複数カラムでのDISTINCT

DISTINCTは複数のカラムにも適用できます。この場合、指定したすべてのカラムの値の組み合わせが一意であるレコードだけが返されます。

基本構文

```
SELECT DISTINCT カラム名1, カラム名2, ... FROM テーブル名 [WHERE 条件];
```

例3：複数カラムでの重複除外

例えば、授業スケジュール（course_schedule）テーブルから、どの講座（course_id）がどの時限（period_id）に開講されているかを重複なしで取得するには：

```
SELECT DISTINCT course_id, period_id FROM course_schedule;
```

実行結果：

course_id	period_id
1	1
2	3
3	4
4	2
...	...

この結果は、course_idとperiod_idの組み合わせが一意のレコードのみを表示します。同じ講座が異なる時限に開講される場合や、異なる講座が同じ時限に開講される場合は、別々のレコードとして表示されます。

COUNT関数との組み合わせ

DISTINCTはCOUNT関数と組み合わせることで、一意の値の数を数えることができます。

例4：一意の値の数を数える

例えば、学生（students）テーブルに何人の学生が登録されているかを数えるには：

```
SELECT COUNT(*) AS 学生総数 FROM students;
```

実行結果：

学生総数
100

一方、受講（student_courses）テーブルに登録されている一意の学生数を数えるには：

```
SELECT COUNT(DISTINCT student_id) AS 受講学生数 FROM student_courses;
```

実行結果：

受講学生数
85

この2つの結果の差は、まだ1つも講座を受講していない学生が15人いることを意味します。

DISTINCTとNULLの扱い

DISTINCTを使う場合、NULL値も1つの値として扱われます。複数のNULL値は1つのNULL値として集約されます。

例5：NULLを含むデータでのDISTINCT

例えば、出席（attendance）テーブルから、コメント（comment）の一意の値を取得するとします：

```
SELECT DISTINCT comment FROM attendance;
```

実行結果：

comment
NULL
15分遅刻
5分遅刻
事前連絡あり
体調不良
...

この結果には、NULL値も1つの行として含まれています。

DISTINCTとORDER BYの組み合わせ

DISTINCTはORDER BY句と組み合わせて使うことができます。これにより、重複を除外した後で結果を並べ替えることができます。

例6：DISTINCTとORDER BYの組み合わせ

例えば、講座（courses）テーブルから、どの教師（teacher_id）が講座を担当しているかを重複なしで取得し、教師IDの順に並べるには：

```
SELECT DISTINCT teacher_id FROM courses ORDER BY teacher_id;
```

実行結果：

teacher_id
101
102

teacher_id
103
104
…

DISTINCTとWHEREの組み合わせ

DISTINCTはWHERE句と組み合わせて使うこともできます。これにより、特定の条件に合うレコードだけを対象に重複を除外できます。

例7：DISTINCTとWHEREの組み合わせ

例えば、2025年5月に授業がある教室（classroom_id）の一覧を重複なしで取得するには：

```
SELECT DISTINCT classroom_id
FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31';
```

実行結果：

classroom_id
101A
102B
201C
202D
…

DISTINCTとGROUP BYの違い

DISTINCTとGROUP BYはどちらも「重複を除外する」という点では似ていますが、目的と使い方に違いがあります。

- **DISTINCT**：単純に重複する行を除外します。
- **GROUP BY**：グループごとに集計を行うために使います。集計関数（COUNT、SUM、AVG、MAX、MINなど）と一緒に使うことが一般的です。

例8：DISTINCTとGROUP BYの比較

例えば、講座（courses）テーブルから、どの教師（teacher_id）が講座を担当しているかを調べる場合：

DISTINCTを使う方法：

```
SELECT DISTINCT teacher_id FROM courses;
```

GROUP BYを使う方法：

```
SELECT teacher_id FROM courses GROUP BY teacher_id;
```

両方とも同じ結果を返しますが、目的が異なります。GROUP BYは集計を行うためのもので、例えば各教師が担当する講座数を数えたい場合は：

```
SELECT teacher_id, COUNT(*) AS 担当講座数
FROM courses
GROUP BY teacher_id;
```

このように、GROUP BYと集計関数を組み合わせて使います。

パフォーマンスへの影響と注意点

1. **処理コスト**：DISTINCTはすべての行を調査して重複を除外するため、大量のデータがある場合は処理コストが高くなる可能性があります。
2. **代替手段の検討**：場合によっては、DISTINCTの代わりにGROUP BYを使ったり、EXISTS/NOT EXISTSを使ったりと、より効率的な方法があることがあります。
3. **部分一致には使えない**：DISTINCTは完全に一致するレコードのみを対象とします。部分的な一致や似ているレコードの除外には使えません。

練習問題

問題12-1

course_schedule（授業カレンダー）テーブルから、授業が行われる日付（schedule_date）のリストを重複なしで取得するSQLを書いてください。

問題12-2

grades（成績）テーブルから、何種類の評価タイプ（grade_type）があるかを数えるSQLを書いてください。

問題12-3

student_courses（受講）テーブルから、講座を受講している学生ID（student_id）を重複なしで取得し、IDの昇順に並べるSQLを書いてください。

問題12-4

course_schedule（授業カレンダー）テーブルから、どの教室（classroom_id）でどの時限（period_id）に授業が行われているかの組み合わせを重複なしで取得するSQLを書いてください。

問題12-5

attendance（出席）テーブルから、コメント（comment）が入力されている（NULLでない）一意のコメント内容を取得するSQLを書いてください。

問題12-6

grades（成績）テーブルから、どの学生（student_id）がどの講座（course_id）を受講したかの組み合わせを重複なしで取得し、学生ID、講座IDの順に並べるSQLを書いてください。

解答

解答12-1

```
SELECT DISTINCT schedule_date FROM course_schedule;
```

解答12-2

```
SELECT COUNT(DISTINCT grade_type) AS 評価タイプ数 FROM grades;
```

解答12-3

```
SELECT DISTINCT student_id FROM student_courses ORDER BY student_id;
```

解答12-4

```
SELECT DISTINCT classroom_id, period_id FROM course_schedule;
```

解答12-5

```
SELECT DISTINCT comment FROM attendance WHERE comment IS NOT NULL;
```

解答12-6

```
SELECT DISTINCT student_id, course_id
FROM grades
ORDER BY student_id, course_id;
```

まとめ

この章では、クエリ結果から重複を除外するためのDISTINCTキーワードについて学びました：

1. **DISTINCTの基本**：クエリ結果から重複する行を除外する方法
2. **単一カラムでの使用**：1つのカラムの重複を除外する方法
3. **複数カラムでの使用**：複数カラムの組み合わせの重複を除外する方法
4. **COUNT関数との組み合わせ**：一意の値の数を数える方法
5. **NULLの扱い**：DISTINCT使用時のNULL値の取り扱い
6. **ORDER BYとの組み合わせ**：重複除外後の並べ替え
7. **WHEREとの組み合わせ**：条件付きでの重複除外
8. **GROUP BYとの違い**：似た機能を持つGROUP BYとの使い分け
9. **パフォーマンスへの影響**：DISTINCTを使用する際の注意点

DISTINCTは、データベースから一意の値のリストを取得するための便利な機能です。特に、テーブル内の特定のカラムにどのような値が存在するかを調べたい場合や、重複のないマスターリストを作成したい場合に役立ちます。

次の章では、複数のテーブルを結合して情報を取得するための「JOIN基本：テーブル結合の概念」について学びます。

SQL学習テキスト 完全版

このテキストは、SQLの基礎から応用まで体系的に学習できるように構成されています。 学校データベースを使った実践的な例題と練習問題を通じて、実務で使えるSQLスキルを身につけることができます。

目次

---

1. SELECT基本：単一テーブルから特定カラムを取得する

はじめに

データベースからデータを取り出す作業は、料理人が大きな冷蔵庫から必要な材料だけを取り出すようなものです。SQLでは、この「取り出す」作業を「SELECT文」で行います。

SELECT文はSQLの中で最も基本的で、最もよく使われる命令です。この章では、単一のテーブル（データの表）から必要な情報だけを取り出す方法を学びます。

基本構文

SELECT文の最も基本的な形は次のとおりです：

```
SELECT カラム名 FROM テーブル名;
```

この文は「テーブル名というテーブルからカラム名という列のデータを取り出してください」という意味です。

> **用語解説**：
>
> - **SELECT**：「選択する」という意味のSQLコマンドで、データを取り出すときに使います。
> - **カラム**：テーブルの縦の列のことで、同じ種類のデータが並んでいます（例：名前のカラム、年齢のカラムなど）。
> - **FROM**：「～から」という意味で、どのテーブルからデータを取るかを指定します。
> - **テーブル**：データベース内の表のことで、行と列で構成されています。

実践例：単一カラムの取得

学校データベースの中の「teachers」テーブル（教師テーブル）から、教師の名前だけを取得してみましょう。

```
SELECT teacher_name FROM teachers;
```

実行結果：

teacher_name
寺内鞍
田尻朋美
内村海凪
藤本理恵
黒木大介
星野涼子
深山誠一
吉岡由佳
山田太郎
佐藤花子

teacher_name
…

これは「teachers」テーブルの「teacher_name」という列（先生の名前）だけを取り出しています。

複数のカラムを取得する

料理に複数の材料が必要なように、データを取り出すときも複数の列が必要なことがよくあります。複数のカラムを取得するには、カラム名をカンマ（,）で区切って指定します。

```
SELECT カラム名1, カラム名2, カラム名3 FROM テーブル名;
```

例えば、教師の番号（ID）と名前を一緒に取得してみましょう：

```
SELECT teacher_id, teacher_name FROM teachers;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍
102	田尻朋美
103	内村海凪
104	藤本理恵
105	黒木大介
106	星野涼子
…	…

すべてのカラムを取得する

テーブルのすべての列を取得したい場合は、アスタリスク（*）を使います。これは「すべての列」を意味するワイルドカードです。

```
SELECT * FROM テーブル名;
```

例：

```
SELECT * FROM teachers;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍
102	田尻朋美
103	内村海凪
…	…

> **注意**：`SELECT *`は便利ですが、実際の業務では必要なカラムだけを指定する方が良いとされています。これは、データ量が多いときに処理速度が遅くなるのを防ぐためです。

カラムに別名をつける（AS句）

取得したカラムに分かりやすい名前（別名）をつけることができます。これは「AS」句を使います。

```
SELECT カラム名 AS 別名 FROM テーブル名;
```

> **用語解説**：
>
> - **AS**：「～として」という意味で、カラムに別名をつけるときに使います。この別名は結果を表示するときだけ使われます。

例えば、教師IDを「番号」、教師名を「名前」として表示してみましょう：

```
SELECT teacher_id AS 番号, teacher_name AS 名前 FROM teachers;
```

実行結果：

番号	名前
101	寺内鞍
102	田尻朋美
103	内村海凪
…	…

ASは省略することも可能です：

```
SELECT teacher_id 番号, teacher_name 名前 FROM teachers;
```

計算式を使う

SELECT文では、カラムの値を使った計算もできます。例えば、成績テーブルから点数と満点を取得して、達成率（パーセント）を計算してみましょう。

```
SELECT student_id, course_id, grade_type,
       score, max_score,
       (score / max_score) * 100 AS 達成率
FROM grades;
```

実行結果：

student_id	course_id	grade_type	score	max_score	達成率
301	1	中間テスト	85.5	100.0	85.5
302	1	中間テスト	92.0	100.0	92.0
…	…	…	…	…	…

重複を除外する（DISTINCT）

同じ値が複数ある場合に、重複を除いて一意の値だけを表示するには「DISTINCT」キーワードを使います。

> **用語解説**：
>
> - **DISTINCT**：「異なる」「区別された」という意味で、重複する値を除外して一意の値だけを取得します。

例えば、どの講座にどの教師が担当しているかを重複なしで確認してみましょう：

```
SELECT DISTINCT teacher_id FROM courses;
```

実行結果：

teacher_id
101
102
103
104
…

これにより、courses（講座）テーブルで使われている教師IDが重複なく表示されます。

文字列の結合

文字列を結合するには、MySQLでは「CONCAT」関数を使います。例えば、教師のIDと名前を組み合わせて表示してみましょう：

```
SELECT CONCAT('教師ID:', teacher_id, ' 名前:', teacher_name) AS 教師情報
FROM teachers;
```

実行結果：

教師情報
教師ID:101 名前:寺内鞍
教師ID:102 名前:田尻朋美

...

> **用語解説**：
>
> - **CONCAT**：複数の文字列を一つにつなげる関数です。

SELECT文と終了記号

SQLの文は通常、セミコロン（;）で終わります。これは「この命令はここで終わりです」という合図です。

複数のSQL文を一度に実行する場合は、それぞれの文の最後にセミコロンをつけます。

練習問題

問題1-1

students（学生）テーブルから、すべての学生の名前（student_name）を取得するSQLを書いてください。

問題1-2

classrooms（教室）テーブルから、教室ID（classroom_id）と教室名（classroom_name）を取得するSQLを書いてください。

問題1-3

courses（講座）テーブルから、すべての列（カラム）を取得するSQLを書いてください。

問題1-4

class_periods（授業時間）テーブルから、時限ID（period_id）、開始時間（start_time）、終了時間（end_time）を取得し、開始時間には「開始」、終了時間には「終了」という別名をつけるSQLを書いてください。

問題1-5

grades（成績）テーブルから、学生ID（student_id）、講座ID（course_id）、評価タイプ（grade_type）、得点（score）、満点（max_score）、そして得点を満点で割って100を掛けた値を「パーセント」という別名で

取得するSQLを書いてください。

問題1-6

course_schedule（授業カレンダー）テーブルから、schedule_date（予定日）カラムだけを重複なしで取得するSQLを書いてください。

解答

解答1-1

```
SELECT student_name FROM students;
```

解答1-2

```
SELECT classroom_id, classroom_name FROM classrooms;
```

解答1-3

```
SELECT * FROM courses;
```

解答1-4

```
SELECT period_id, start_time AS 開始, end_time AS 終了 FROM class_periods;
```

解答1-5

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score / max_score) * 100 AS パーセント
FROM grades;
```

解答1-6

```
SELECT DISTINCT schedule_date FROM course_schedule;
```

まとめ

この章では、SQLのSELECT文の基本を学びました：

1. 単一カラムの取得: `SELECT カラム名 FROM テーブル名;`
2. 複数カラムの取得: `SELECT カラム名1, カラム名2 FROM テーブル名;`
3. すべてのカラムの取得: `SELECT * FROM テーブル名;`
4. カラムに別名をつける: `SELECT カラム名 AS 別名 FROM テーブル名;`
5. 計算式を使う: `SELECT カラム名, (計算式) AS 別名 FROM テーブル名;`
6. 重複を除外する: `SELECT DISTINCT カラム名 FROM テーブル名;`
7. 文字列の結合: `SELECT CONCAT(文字列1, カラム名, 文字列2) FROM テーブル名;`

これらの基本操作を使いこなせるようになれば、データベースから必要な情報を効率よく取り出せるようになります。次の章では、WHERE句を使って条件に合ったデータだけを取り出す方法を学びます。

---

2. WHERE句：条件に合ったレコードを絞り込む

はじめに

前章では、テーブルからデータを取得する基本的な方法を学びました。しかし実際の業務では、すべてのデータではなく、特定の条件に合ったデータだけを取得したいことがほとんどです。

例えば、「全生徒の情報」ではなく「特定の学科の生徒だけ」や「成績が80点以上の学生だけ」といった形で、データを絞り込みたい場合があります。

このような場合に使用するのが「WHERE句」です。WHERE句は、SELECTコマンドの後に追加して使い、条件に合致するレコード（行）だけを取得します。

基本構文

WHERE句の基本的な形は次のとおりです：

```
SELECT カラム名 FROM テーブル名 WHERE 条件式;
```

> **用語解説**：
>
> - **WHERE**：「～の場所で」「～の条件で」という意味のSQLコマンドで、条件に合うデータだけを抽出するために使います。
> - **条件式**：データが満たすべき条件を指定するための式です。例えば「age > 20」（年齢が20より大きい）などです。
> - **レコード**：テーブルの横の行のことで、1つのデータの集まりを表します。

基本的な比較演算子

WHERE句では、様々な比較演算子を使って条件を指定できます：

演算子	意味	例
=	等しい	age = 25
<>	等しくない（≠と同じ）	gender <> 'male'

演算子	意味	例
>	より大きい	score > 80
<	より小さい	price < 1000
>=	以上	height >= 170
<=	以下	weight <= 70

実践例：基本的な条件での絞り込み

例1：等しい（=）

例えば、教師ID（teacher_id）が101の教師のみを取得するには：

```
SELECT * FROM teachers WHERE teacher_id = 101;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍

例2：より大きい（>）

成績（grades）テーブルから、90点を超える成績だけを取得するには：

```
SELECT * FROM grades WHERE score > 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

例3：等しくない（<>）

講座IDが3ではない講座に関する成績を取得するには：

```
SELECT * FROM grades WHERE course_id <> '3';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
301	2	実技試験	88.0	100.0	2025-05-18
…	…	…	…	…	…

文字列の比較

テキスト（文字列）を条件にする場合は、シングルクォーテーション（'）またはダブルクォーテーション（"）で囲みます。MySQLではどちらも使えますが、多くの場合シングルクォーテーションが推奨されます。

```
SELECT * FROM テーブル名 WHERE テキストカラム = 'テキスト値';
```

例えば、教師名（teacher_name）が「田尻朋美」の教師を検索するには：

```
SELECT * FROM teachers WHERE teacher_name = '田尻朋美';
```

実行結果：

teacher_id	teacher_name
102	田尻朋美

日付の比較

日付の比較も同様にシングルクォーテーションで囲みます。日付の形式はデータベースの設定によって異なりますが、一般的にはISO形式（YYYY-MM-DD）が使われます。

```
SELECT * FROM テーブル名 WHERE 日付カラム = '日付';
```

例えば、2025年5月20日に提出された成績を検索するには：

```
SELECT * FROM grades WHERE submission_date = '2025-05-20';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
303	1	中間テスト	78.5	100.0	2025-05-20
...	...	...	...	...	...

また、日付同士の大小関係も比較できます：

```
SELECT * FROM grades WHERE submission_date < '2025-05-15';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	レポート1	45.0	50.0	2025-05-10
302	1	レポート1	48.0	50.0	2025-05-10
304	5	ER図作成課題	27.5	30.0	2025-05-14
...	...	...	...	...	...

複数の条件を指定する（次章の内容）

WHERE句では、複数の条件を組み合わせることもできます。この詳細は次章「論理演算子：AND、OR、NOTを使った複合条件」で説明します。

例えば、講座IDが1で、かつ、得点が90以上の成績を取得するには：

```
SELECT * FROM grades WHERE course_id = '1' AND score >= 90;
```

この例の詳細な説明は次章で行います。

練習問題

問題2-1

students（学生）テーブルから、学生ID（student_id）が「310」の学生情報を取得するSQLを書いてください。

問題2-2

classrooms（教室）テーブルから、収容人数（capacity）が30人より多い教室の情報をすべて取得するSQLを書いてください。

問題2-3

courses（講座）テーブルから、教師ID（teacher_id）が「105」の講座情報を取得するSQLを書いてください。

問題2-4

course_schedule（授業カレンダー）テーブルから、「2025-05-15」の授業スケジュールをすべて取得するSQLを書いてください。

問題2-5

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」で、得点（score）が80点未満の成績を取得するSQLを書いてください。

問題2-6

teachers（教師）テーブルから、教師ID（teacher_id）が「101」ではない教師の名前を取得するSQLを書いてください。

解答

解答2-1

```
SELECT * FROM students WHERE student_id = 310;
```

解答2-2

```
SELECT * FROM classrooms WHERE capacity > 30;
```

解答2-3

```
SELECT * FROM courses WHERE teacher_id = 105;
```

解答2-4

```
SELECT * FROM course_schedule WHERE schedule_date = '2025-05-15';
```

解答2-5

```
SELECT * FROM grades WHERE grade_type = '中間テスト' AND score < 80;
```

解答2-6

```
SELECT teacher_name FROM teachers WHERE teacher_id <> 101;
```

まとめ

この章では、WHERE句を使って条件に合ったレコードを取得する方法を学びました：

1. 基本的な比較演算子（=, <>, >, <, >=, <=）の使い方
2. 数値による条件絞り込み
3. 文字列（テキスト）による条件絞り込み
4. 日付による条件絞り込み

WHERE句は、大量のデータから必要な情報だけを取り出すための非常に重要な機能です。実際のデータベース操作では、この条件絞り込みを頻繁に使います。

次の章では、複数の条件を組み合わせるための「論理演算子（AND、OR、NOT）」について学びます。

---

3. 論理演算子：AND、OR、NOTを使った複合条件

はじめに

前章では、WHERE句を使って単一の条件でデータを絞り込む方法を学びました。しかし実際の業務では、より複雑な条件でデータを絞り込む必要があることがよくあります。

例えば：

- 「成績が80点以上**かつ**出席率が90%以上の学生」
- 「数学**または**英語の成績が優秀な学生」
- 「課題を**まだ提出していない**学生」

このような複合条件を指定するために使うのが論理演算子です。主な論理演算子は次の3つです：

- **AND**：両方の条件を満たす（かつ）
- **OR**：いずれかの条件を満たす（または）
- **NOT**：条件を満たさない（～ではない）

AND演算子

AND演算子は、指定した**すべての条件を満たす**レコードだけを取得したいときに使います。

> **用語解説**：
>
> - **AND**：「かつ」「そして」という意味の論理演算子です。複数の条件をすべて満たすデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 条件1 AND 条件2;
```

例：ANDを使った複合条件

例えば、中間テストで90点以上かつ満点が100点の成績レコードを取得するには：

```
SELECT * FROM grades
WHERE score >= 90 AND max_score = 100;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

この例では、「score >= 90」と「max_score = 100」の両方の条件を満たすレコードだけが取得されます。

3つ以上の条件の組み合わせ

ANDを使って3つ以上の条件を組み合わせることもできます：

```
SELECT * FROM grades
WHERE score >= 90 AND max_score = 100 AND grade_type = '中間テスト';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

OR演算子

OR演算子は、指定した条件の**いずれか一つでも満たす**レコードを取得したいときに使います。

> **用語解説**：
>
> - **OR**：「または」「もしくは」という意味の論理演算子です。複数の条件のうち少なくとも1つを満たすデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 条件1 OR 条件2;
```

例：ORを使った複合条件

例えば、教師IDが101または102の講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id = 101 OR teacher_id = 102;
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
2	UNIX入門	102
3	Cプログラミング演習	101
29	コードリファクタリングとクリーンコード	101
40	ソフトウェアアーキテクチャパターン	102
…	…	…

この例では、「teacher_id = 101」または「teacher_id = 102」のいずれかの条件を満たすレコードが取得されます。

NOT演算子

NOT演算子は、指定した条件を**満たさない**レコードを取得したいときに使います。

> **用語解説**：
>
> - **NOT**：「～ではない」という意味の論理演算子です。条件を否定して、その条件を満たさないデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE NOT 条件;
```

例：NOTを使った否定条件

例えば、完了（completed）状態ではない授業スケジュールを取得するには：

```
SELECT * FROM course_schedule
WHERE NOT status = 'completed';
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
45	11	2025-05-20	5	401G	106	scheduled
46	12	2025-05-21	1	301E	107	scheduled
50	2	2025-05-23	3	101A	102	cancelled
…	…	…	…	…	…	…

この例では、statusが「completed」ではないレコード（scheduled状態やcancelled状態）が取得されます。

NOT演算子は、「～ではない」という否定の条件を作るために使われます。例えば次の2つの書き方は同じ意味になります：

```
SELECT * FROM teachers WHERE NOT teacher_id = 101;
SELECT * FROM teachers WHERE teacher_id <> 101;
```

複合論理条件（AND、OR、NOTの組み合わせ）

AND、OR、NOTを組み合わせて、より複雑な条件を指定することもできます。

例：ANDとORの組み合わせ

例えば、「成績が90点以上で中間テストである」または「成績が45点以上でレポートである」レコードを取得するには：

```
SELECT * FROM grades
WHERE (score >= 90 AND grade_type = '中間テスト')
   OR (score >= 45 AND grade_type = 'レポート1');
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
302	1	レポート1	48.0	50.0	2025-05-10
308	1	レポート1	47.0	50.0	2025-05-09

student_id	course_id	grade_type	score	max_score	submission_date
...	...	...	...	...	...

優先順位と括弧の使用

論理演算子を組み合わせる場合、演算子の優先順位に注意が必要です。基本的に**ANDはORよりも優先順位が高い**です。つまり、ANDが先に処理されます。

例えば：

```
WHERE 条件1 OR 条件2 AND 条件3
```

これは次のように解釈されます：

```
WHERE 条件1 OR (条件2 AND 条件3)
```

意図した条件と異なる場合は、**括弧（）を使って明示的にグループ化**することが重要です：

```
WHERE (条件1 OR 条件2) AND 条件3
```

例：括弧を使った条件のグループ化

教師IDが101または102で、かつ、講座名に「プログラミング」という単語が含まれる講座を取得するには：

```sql
SELECT * FROM courses
WHERE (teacher_id = 101 OR teacher_id = 102)
  AND course_name LIKE '%プログラミング%';
```

実行結果：

course_id	course_name	teacher_id
3	Cプログラミング演習	101
...	...	...

この例では、括弧を使って「teacher_id = 101 OR teacher_id = 102」の部分をグループ化し、その条件と「course_name LIKE '%プログラミング%'」の条件をANDで結合しています。

練習問題

問題3-1

grades（成績）テーブルから、課題タイプ（grade_type）が「中間テスト」かつ点数（score）が85点以上のレコードを取得するSQLを書いてください。

問題3-2

classrooms（教室）テーブルから、収容人数（capacity）が40人以下または建物（building）が「1号館」の教室を取得するSQLを書いてください。

問題3-3

teachers（教師）テーブルから、教師ID（teacher_id）が101、102、103ではない教師の情報を取得するSQLを書いてください。

問題3-4

course_schedule（授業カレンダー）テーブルから、2025年5月20日の授業で、時限（period_id）が1か2で、かつ状態（status）が「scheduled」の授業を取得するSQLを書いてください。

問題3-5

students（学生）テーブルから、学生名（student_name）に「田」または「山」を含む学生を取得するSQLを書いてください。

問題3-6

grades（成績）テーブルから、提出日（submission_date）が2025年5月15日以降で、かつ（「中間テスト」で90点以上または「レポート1」で45点以上）の成績を取得するSQLを書いてください。

解答

解答3-1

```sql
SELECT * FROM grades
WHERE grade_type = '中間テスト' AND score >= 85;
```

解答3-2

```sql
SELECT * FROM classrooms
WHERE capacity <= 40 OR building = '1号館';
```

解答3-3

```sql
SELECT * FROM teachers
WHERE NOT (teacher_id = 101 OR teacher_id = 102 OR teacher_id = 103);
```

または

```
SELECT * FROM teachers
WHERE teacher_id <> 101 AND teacher_id <> 102 AND teacher_id <> 103;
```

解答3-4

```
SELECT * FROM course_schedule
WHERE schedule_date = '2025-05-20'
  AND (period_id = 1 OR period_id = 2)
  AND status = 'scheduled';
```

解答3-5

```
SELECT * FROM students
WHERE student_name LIKE '%田%' OR student_name LIKE '%山%';
```

解答3-6

```
SELECT * FROM grades
WHERE submission_date >= '2025-05-15'
  AND ((grade_type = '中間テスト' AND score >= 90)
       OR (grade_type = 'レポート1' AND score >= 45));
```

まとめ

この章では、論理演算子（AND、OR、NOT）を使って複合条件を作る方法を学びました：

1. **AND**：すべての条件を満たすレコードを取得（条件1 AND 条件2）
2. **OR**：いずれかの条件を満たすレコードを取得（条件1 OR 条件2）
3. **NOT**：指定した条件を満たさないレコードを取得（NOT 条件）
4. **複合条件**：AND、OR、NOTを組み合わせたより複雑な条件
5. **括弧（）**：条件をグループ化して優先順位を明示的に指定

これらの論理演算子を使いこなすことで、より複雑で細かな条件でデータを絞り込むことができるようになります。実際のデータベース操作では、複数の条件を組み合わせることが頻繁にあるため、この章で学んだ内容は非常に重要です。

次の章では、テキストデータに対する検索を行う「パターンマッチング」について学びます。

---

4. パターンマッチング：LIKE演算子と%、_ワイルドカード

はじめに

前章までは、データの完全一致や数値の比較といった条件での絞り込みを学びました。しかし実際の業務では、もっと柔軟な検索が必要なケースがあります。例えば：

- 「山」で始まる名前の学生を検索したい
- 「プログラミング」という単語を含む講座名を探したい
- 電話番号の一部だけ覚えているデータを探したい

このような「部分一致」や「パターン一致」の検索を行うためのSQLの機能が「パターンマッチング」です。この章では、パターンマッチングを行うための「LIKE演算子」と「ワイルドカード文字」について学びます。

LIKE演算子の基本

LIKE演算子は、文字列のパターンマッチングを行うための演算子です。WHERE句と組み合わせて使用します。

> **用語解説**：
>
> - **LIKE**：「～のような」という意味の演算子で、パターンに一致する文字列を検索します。
> - **パターンマッチング**：完全一致ではなく、一定のパターンに合致するデータを検索する方法です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 文字列カラム LIKE 'パターン';
```

パターンには、通常の文字に加えて、特別な意味を持つ「ワイルドカード文字」を使用できます。

ワイルドカード文字

SQLでは主に2つのワイルドカード文字があります：

1. **%（パーセント）**：0文字以上の任意の文字列に一致します。
2. _**（アンダースコア）**：任意の1文字に一致します。

> **用語解説**：
>
> - **ワイルドカード**：任意の文字や文字列に一致する特殊な文字記号です。トランプのジョーカーのように、様々な値に代用できます。

LIKE演算子の使い方：実践例

例1：%（パーセント）を使ったパターンマッチング

「〜で始まる」パターン：前方一致

例えば、「山」で始まる学生名を検索するには：

```
SELECT * FROM students WHERE student_name LIKE '山%';
```

実行結果：

student_id	student_name
312	山本裕子
325	山田翔太
…	…

ここでの「山%」は「山で始まり、その後に0文字以上の任意の文字が続く」という意味です。

「〜で終わる」パターン：後方一致

例えば、「子」で終わる教師名を検索するには：

```
SELECT * FROM teachers WHERE teacher_name LIKE '%子';
```

実行結果：

teacher_id	teacher_name
102	田尻朋美
106	星野涼子
108	吉岡由佳
110	佐藤花子
…	…

「〜を含む」パターン：部分一致

例えば、「プログラミング」という単語を含む講座名を検索するには：

```
SELECT * FROM courses WHERE course_name LIKE '%プログラミング%';
```

実行結果：

course_id	course_name	teacher_id

course_id	course_name	teacher_id
3	Cプログラミング演習	101
14	IoTデバイスプログラミング実践	110
…	…	…

例2：_（アンダースコア）を使ったパターンマッチング

アンダースコアは任意の1文字に一致します。例えば、教室IDが「10_A」パターン（最初の2文字が「10」、3文字目が任意の1文字、最後が「A」）の教室を検索するには：

```
SELECT * FROM classrooms WHERE classroom_id LIKE '10_A';
```

実行結果：

classroom_id	classroom_name	capacity	building	facilities
101A	1号館コンピュータ実習室A	30	1号館	パソコン30台、プロジェクター
…	…	…	…	…

例3：%と_の組み合わせ

ワイルドカード文字は組み合わせて使うこともできます。例えば、「2文字目が田」の学生を検索するには：

```
SELECT * FROM students WHERE student_name LIKE '_田%';
```

実行結果：

student_id	student_name
321	井上竜也
384	櫻井翼
…	…

NOT LIKEを使った否定形のパターンマッチング

特定のパターンに一致しないレコードを検索したい場合は、「NOT LIKE」を使います。

> **用語解説**：
>
> - **NOT LIKE**：「～のパターンに一致しない」という意味で、指定したパターンに一致しないデータを検索します。

例えば、講座名に「入門」を含まない講座を検索するには：

```
SELECT * FROM courses WHERE course_name NOT LIKE '%入門%';
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
3	Cプログラミング演習	101
...	...	...

エスケープ文字の使用

もし検索したいパターンに「%」や「_」自体が含まれている場合は、それらを特別な文字としてではなく、通常の文字として扱うために「エスケープ文字」を使います。

> **用語解説**：
>
> - **エスケープ文字**：特別な意味を持つ文字を通常の文字として扱うための印です。

MySQLでは、バックスラッシュ（\）をエスケープ文字として使用できます。例えば、「50%」という値そのものを検索するには：

```
SELECT * FROM テーブル名 WHERE カラム LIKE '50\%';
```

または、ESCAPE句を使って明示的にエスケープ文字を指定することもできます：

```
SELECT * FROM テーブル名 WHERE カラム LIKE '50!%' ESCAPE '!';
```

この例では「!」をエスケープ文字として指定しています。

大文字と小文字の区別

MySQLのデフォルト設定では、LIKE演算子は大文字と小文字を区別しません（大文字小文字を同じものとして扱います）。

例えば、次の2つのクエリは同じ結果を返します：

```
SELECT * FROM courses WHERE course_name LIKE '%web%';
SELECT * FROM courses WHERE course_name LIKE '%Web%';
```

もし大文字と小文字を区別した検索が必要な場合は、「BINARY」キーワードを使用します：

```
SELECT * FROM courses WHERE course_name LIKE BINARY '%Web%';
```

この場合、「Web」は「web」とは一致しません。

複合条件との組み合わせ

LIKE演算子は、これまで学んだAND、OR、NOTなどの論理演算子と組み合わせて使うこともできます。

例えば、「田」で始まる名前で、かつ教師IDが102から105の間の教師を検索するには：

```
SELECT * FROM teachers
WHERE teacher_name LIKE '田%'
  AND teacher_id BETWEEN 102 AND 105;
```

実行結果：

teacher_id	teacher_name
102	田尻朋美
...	...

練習問題

問題4-1

students（学生）テーブルから、学生名（student_name）が「佐藤」で始まる学生の情報をすべて取得するSQLを書いてください。

問題4-2

courses（講座）テーブルから、講座名（course_name）に「データ」という単語を含む講座の情報を取得するSQLを書いてください。

問題4-3

classrooms（教室）テーブルから、教室名（classroom_name）が「コンピュータ実習室」で終わる教室の情報を取得するSQLを書いてください。

問題4-4

teachers（教師）テーブルから、教師名（teacher_name）の2文字目が「木」である教師の情報を取得するSQLを書いてください。

問題4-5

courses（講座）テーブルから、講座名（course_name）に「入門」または「基礎」を含む講座を取得するSQLを書いてください。

問題4-6

students（学生）テーブルから、学生名（student_name）が「山」で始まり、かつ「子」で終わらない学生を取得するSQLを書いてください。

解答

解答4-1

```
SELECT * FROM students WHERE student_name LIKE '佐藤%';
```

解答4-2

```
SELECT * FROM courses WHERE course_name LIKE '%データ%';
```

解答4-3

```
SELECT * FROM classrooms WHERE classroom_name LIKE '%コンピュータ実習室';
```

解答4-4

```
SELECT * FROM teachers WHERE teacher_name LIKE '_木%';
```

解答4-5

```
SELECT * FROM courses
WHERE course_name LIKE '%入門%' OR course_name LIKE '%基礎%';
```

解答4-6

```
SELECT * FROM students
WHERE student_name LIKE '山%' AND student_name NOT LIKE '%子';
```

まとめ

この章では、パターンマッチングを行うためのLIKE演算子と、その中で使用するワイルドカード文字（%と_）について学びました：

1. **LIKE演算子**：文字列パターンに一致するデータを検索するための演算子

2. **%（パーセント）**：0文字以上の任意の文字列に一致するワイルドカード
3. **_（アンダースコア）**：任意の1文字に一致するワイルドカード
4. **前方一致**：「パターン%」で「パターンで始まる」文字列に一致
5. **後方一致**：「%パターン」で「パターンで終わる」文字列に一致
6. **部分一致**：「%パターン%」で「パターンを含む」文字列に一致
7. **NOT LIKE**：指定したパターンに一致しないデータを検索
8. **エスケープ文字**：特殊文字（%や_）を通常の文字として扱うための方法
9. **複合条件との組み合わせ**：AND、ORなどと組み合わせたより複雑な条件

パターンマッチングは、特にテキストデータを扱う際に非常に便利な機能です。部分的な情報しか持っていない場合や、特定のパターンを持つデータを探す場合に活用できます。

次の章では、範囲指定のための「BETWEEN演算子」と「IN演算子」について学びます。

---

5. 範囲指定：BETWEEN、IN演算子

はじめに

これまでの章では、等号（=）や不等号（>、<）を使って条件を指定する方法や、LIKE演算子を使ったパターンマッチングを学びました。この章では、値の範囲を指定する「BETWEEN演算子」と、複数の値を一度に指定できる「IN演算子」について学びます。

これらの演算子を使うと、次のような検索がより簡単になります：

- 「80点から90点の間の成績」
- 「2025年4月から6月の間のスケジュール」
- 「特定の教師IDリストに該当する講座」

BETWEEN演算子：範囲を指定する

BETWEEN演算子は、ある値が指定した範囲内にあるかどうかを調べるために使います。

> **用語解説**：
>
> - **BETWEEN**：「～の間に」という意味の演算子で、ある値が指定した最小値と最大値の間（両端の値を含む）にあるかどうかを判定します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 BETWEEN 最小値 AND 最大値;
```

この構文は次の条件と同じ意味です：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 >= 最小値 AND カラム名 <= 最大値;
```

例1：数値範囲の指定

例えば、成績（grades）テーブルから、80点から90点の間の成績を取得するには：

```
SELECT * FROM grades
WHERE score BETWEEN 80 AND 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
308	6	小テスト1	89.0	100.0	2025-05-15
…	…	…	…	…	…

例2：日付範囲の指定

日付にもBETWEEN演算子が使えます。例えば、2025年5月10日から2025年5月20日までに提出された成績を取得するには：

```
SELECT * FROM grades
WHERE submission_date BETWEEN '2025-05-10' AND '2025-05-20';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	レポート1	45.0	50.0	2025-05-10
302	1	レポート1	48.0	50.0	2025-05-10
301	1	中間テスト	85.5	100.0	2025-05-20
…	…	…	…	…	…

NOT BETWEEN：範囲外を指定する

NOT BETWEENを使うと、指定した範囲の外にある値を持つレコードを取得できます。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 NOT BETWEEN 最小値 AND 最大値;
```

これは次の条件と同じです：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 < 最小値 OR カラム名 > 最大値;
```

例：範囲外の値を取得

例えば、80点未満または90点より高い成績を取得するには：

```
SELECT * FROM grades
WHERE score NOT BETWEEN 80 AND 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
303	1	中間テスト	78.5	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
…	…	…	…	…	…

IN演算子：複数の値を指定する

IN演算子は、ある値が指定した複数の値のリストのいずれかに一致するかどうかを調べるために使います。

> **用語解説**：
>
> - **IN**：「～の中に含まれる」という意味の演算子で、ある値が指定したリストの中に含まれているかどうかを判定します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 IN (値1, 値2, 値3, ...);
```

この構文は次のOR条件の組み合わせと同じ意味です：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 = 値1 OR カラム名 = 値2 OR カラム名 = 値3 OR ...;
```

例1：数値リストの指定

例えば、教師ID（teacher_id）が101、103、105のいずれかである講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id IN (101, 103, 105);
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
3	Cプログラミング演習	101
5	データベース設計と実装	105
10	プロジェクト管理手法	103
…	…	…

例2：文字列リストの指定

文字列のリストにも適用できます。例えば、特定の教室ID（classroom_id）のみの教室情報を取得するには：

```
SELECT * FROM classrooms
WHERE classroom_id IN ('101A', '202D', '301E');
```

実行結果：

classroom_id	classroom_name	capacity	building	facilities
101A	1号館コンピュータ実習室A	30	1号館	パソコン30台、プロジェクター
202D	2号館コンピュータ実習室D	25	2号館	パソコン25台、プロジェクター、3Dプリンター
301E	3号館講義室E	80	3号館	プロジェクター、マイク設備、録画設備

NOT IN：リストに含まれない値を指定する

NOT IN演算子を使うと、指定したリストに含まれない値を持つレコードを取得できます。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 NOT IN (値1, 値2, 値3, ...);
```

これは次の条件と同じです：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 <> 値1 AND カラム名 <> 値2 AND カラム名 <> 値3 AND ...;
```

例：リストに含まれない値を取得

例えば、教師IDが101、102、103以外の教師が担当する講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id NOT IN (101, 102, 103);
```

実行結果：

course_id	course_name	teacher_id
4	Webアプリケーション開発	104
5	データベース設計と実装	105
6	ネットワークセキュリティ	107
…	…	…

IN演算子でのサブクエリの利用（基本）

IN演算子の括弧内には、直接値を書く代わりに、サブクエリ（別のSELECT文）を指定することもできます。これにより、動的に値のリストを生成できます。

> **用語解説**：
>
> - **サブクエリ**：SQL文の中に含まれる別のSQL文のことで、外側のSQL文（メインクエリ）に値や条件を提供します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 IN (SELECT カラム名 FROM 別テーブル WHERE 条件);
```

例：サブクエリを使ったIN条件

例えば、教師名（teacher_name）に「田」を含む教師が担当している講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id IN (
```

```
    SELECT teacher_id FROM teachers
    WHERE teacher_name LIKE '%田%'
);
```

実行結果：

course_id	course_name	teacher_id
2	UNIX入門	102
10	プロジェクト管理手法	103
…	…	…

この例では、まず「teachers」テーブルから名前に「田」を含む教師のIDを取得し、それらのIDを持つ講座を「courses」テーブルから取得しています。

BETWEEN演算子とIN演算子の組み合わせ

BETWEEN演算子とIN演算子は、論理演算子（AND、OR）と組み合わせて、さらに複雑な条件を作ることができます。

例：BETWEENとINの組み合わせ

例えば、「教師IDが101、103、105のいずれかで、かつ、2025年5月15日から2025年6月15日の間に実施される授業」を取得するには：

```
SELECT * FROM course_schedule
WHERE teacher_id IN (101, 103, 105)
  AND schedule_date BETWEEN '2025-05-15' AND '2025-06-15';
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
38	1	2025-05-20	1	102B	101	scheduled
42	10	2025-05-23	3	201C	103	scheduled
45	5	2025-05-28	2	402H	105	scheduled
…	…	…	…	…	…	…

練習問題

問題5-1

grades（成績）テーブルから、得点（score）が85点から95点の間にある成績を取得するSQLを書いてください。

問題5-2

course_schedule（授業カレンダー）テーブルから、2025年5月1日から2025年5月31日までの授業スケジュールを取得するSQLを書いてください。

問題5-3

courses（講座）テーブルから、教師ID（teacher_id）が104、106、108のいずれかである講座の情報を取得するSQLを書いてください。

問題5-4

classrooms（教室）テーブルから、教室ID（classroom_id）が「101A」、「201C」、「301E」、「401G」以外の教室情報を取得するSQLを書いてください。

問題5-5

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」または「実技試験」で、かつ得点（score）が80点から90点の間にない成績を取得するSQLを書いてください。

問題5-6

course_schedule（授業カレンダー）テーブルから、教室ID（classroom_id）が「101A」、「202D」のいずれかで、かつ2025年5月15日から2025年5月30日の間に実施される授業スケジュールを取得するSQLを書いてください。

解答

解答5-1

```
SELECT * FROM grades
WHERE score BETWEEN 85 AND 95;
```

解答5-2

```
SELECT * FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31';
```

解答5-3

```
SELECT * FROM courses
WHERE teacher_id IN (104, 106, 108);
```

解答5-4

```
SELECT * FROM classrooms
WHERE classroom_id NOT IN ('101A', '201C', '301E', '401G');
```

解答5-5

```
SELECT * FROM grades
WHERE grade_type IN ('中間テスト', '実技試験')
  AND score NOT BETWEEN 80 AND 90;
```

解答5-6

```
SELECT * FROM course_schedule
WHERE classroom_id IN ('101A', '202D')
  AND schedule_date BETWEEN '2025-05-15' AND '2025-05-30';
```

まとめ

この章では、範囲指定のための「BETWEEN演算子」と複数値指定のための「IN演算子」について学びました：

1. **BETWEEN演算子**：値が指定した範囲内（両端を含む）にあるかどうかをチェック
2. **NOT BETWEEN**：値が指定した範囲外にあるかどうかをチェック
3. **IN演算子**：値が指定したリストのいずれかに一致するかどうかをチェック
4. **NOT IN**：値が指定したリストのいずれにも一致しないかどうかをチェック
5. **サブクエリとIN**：動的に生成された値のリストを使用する方法
6. **複合条件**：BETWEEN、IN、論理演算子を組み合わせたより複雑な条件

これらの演算子を使うことで、複数の条件を指定する場合に、SQLをより簡潔に書くことができます。特に、多くの値を指定する場合や範囲条件を指定する場合に便利です。

次の章では、「NULL値の処理：IS NULL、IS NOT NULL」について学びます。

---

6. NULL値の処理：IS NULL、IS NOT NULL

はじめに

データベースの世界では、データがない状態を表すために「NULL」という特別な値が使われます。NULLは「空」や「0」や「空白文字」とは異なる、「値が存在しない」または「不明」であることを表す特殊な概念です。

例えば、学校データベースでは次のようなシナリオがあります：

- まだ成績が付けられていない（NULL）

- コメントが入力されていない（NULL）
- 授業がキャンセルされたため教室が割り当てられていない（NULL）

この章では、NULL値を正しく処理するための「IS NULL」と「IS NOT NULL」演算子について学びます。

NULLとは何か？

NULL値には、いくつかの特徴があります：

1. **値がない**：NULLは値がないことを表します。0でも空文字列（''）でもなく、値そのものが存在しないことを示します。
2. **不明**：データが不明であることを表す場合もあります。
3. **未設定**：まだ値が設定されていないことを表す場合もあります。
4. **比較できない**：NULLは通常の比較演算子（=, <, >など）で比較できません。

> **用語解説**：
>
> - **NULL**：データベースにおいて「値がない」または「不明」を表す特殊な値です。0や空文字とは異なります。

NULL値と通常の比較演算子

通常の比較演算子（=, <>, >, <, >=, <=）では、NULL値を正しく検出できません。例えば：

```
-- この条件はNULL値に対して常にFALSEを返す
SELECT * FROM テーブル名 WHERE カラム名 = NULL;

-- この条件もNULL値に対して常にFALSEを返す
SELECT * FROM テーブル名 WHERE カラム名 <> NULL;
```

これは、NULL値との等価比較は「不明」と評価されるためです。つまり、NULL = NULLでさえFALSEではなく「不明」になります。

IS NULL演算子

NULL値を持つレコードを検索するには、「IS NULL」演算子を使います。

> **用語解説**：
>
> - **IS NULL**：カラムの値がNULLかどうかを調べる演算子です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE カラム名 IS NULL;
```

例：IS NULLの使用

例えば、コメントが入力されていない（NULL）出席レコードを検索するには：

```
SELECT * FROM attendance WHERE comment IS NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	NULL
1	306	present	NULL
1	307	present	NULL
...	...	...	NULL

IS NOT NULL演算子

逆に、NULL値を持たないレコード（つまり、何らかの値を持つレコード）を検索するには、「IS NOT NULL」演算子を使います。

> **用語解説**：
>
> - **IS NOT NULL**：カラムの値がNULLでないかどうかを調べる演算子です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE カラム名 IS NOT NULL;
```

例：IS NOT NULLの使用

例えば、コメントが入力されている（NOT NULL）出席レコードを検索するには：

```
SELECT * FROM attendance WHERE comment IS NOT NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	302	late	15分遅刻
1	303	absent	事前連絡あり
1	308	late	5分遅刻
...	...	...	...

NULL値の論理的な扱い

NULL値は論理演算（AND、OR、NOT）でも特殊な扱いを受けます。

- **NULL AND TRUE** → NULL（不明）
- **NULL AND FALSE** → FALSE
- **NULL OR TRUE** → TRUE
- **NULL OR FALSE** → NULL（不明）
- **NOT NULL** → NULL（不明）

この特殊な振る舞いが、バグや誤った結果の原因になることがあります。

NULL値と結合条件

テーブル結合（JOINなど、後の章で学習）の際も、NULL値は特殊な扱いを受けます。NULL値同士は「等しい」とは判定されないため、通常の結合条件ではNULL値を持つレコードは結合されません。

IS NULLとIS NOT NULLを使った複合条件

IS NULLとIS NOT NULLも、他の条件と組み合わせて使用できます。

例：複合条件でのIS NULLの使用

例えば、「出席状態が "absent"（欠席）で、コメントがNULLでない（理由が入力されている）レコード」を検索するには：

```
SELECT * FROM attendance
WHERE status = 'absent' AND comment IS NOT NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	303	absent	事前連絡あり
1	317	absent	体調不良
…	…	…	…

NVL/IFNULL/COALESCE関数：NULL値の置換

NULL値を別の値に置き換えるための関数が用意されています。データベースによって関数名が異なることがありますが、機能は似ています：

- MySQL/MariaDB: **IFNULL(expr, replace_value)**
- Oracle: **NVL(expr, replace_value)**
- SQL Server: **ISNULL(expr, replace_value)**
- 標準SQL: **COALESCE(expr1, expr2, ..., exprN)** - 最初のNULLでない式を返します

例：IFNULL関数の使用（MySQL）

例えば、コメントがNULLの場合は「特記事項なし」と表示するには：

```
SELECT schedule_id, student_id, status,
       IFNULL(comment, '特記事項なし') AS comment
FROM attendance;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	特記事項なし
1	302	late	15分遅刻
1	303	absent	事前連絡あり
...	...	...	...

NULLを使う際の注意点

1. **除外の罠**：`WHERE カラム名 <> 値` だけでは、NULL値を持つレコードは含まれません。すべてのレコードを対象にするには：

   ```
   WHERE カラム名 <> 値 OR カラム名 IS NULL
   ```

2. **集計関数**：COUNT(*)はすべての行を数えますが、COUNT(カラム名)はそのカラムがNULLでない行だけを数えます。

3. **インデックス**：多くのデータベースでは、NULL値にもインデックスを適用できますが、データベースによって動作が異なる場合があります。

4. **一意性制約**：一般的に、UNIQUE制約ではNULL値は重複としてカウントされません（複数のNULL値が許可されます）。

練習問題

問題6-1

attendance（出席）テーブルから、コメント（comment）がNULLの出席レコードをすべて取得するSQLを書いてください。

問題6-2

course_schedule（授業カレンダー）テーブルから、状態（status）が「cancelled」で、かつ教室ID（classroom_id）がNULLでないレコードを取得するSQLを書いてください。

問題6-3

grades（成績）テーブルから、提出日（submission_date）がNULLの成績レコードを取得するSQLを書いてください。

問題6-4

attendance（出席）テーブルから、出席状態（status）が「present」か「late」で、かつコメント（comment）がNULLのレコードを取得するSQLを書いてください。

問題6-5

以下のSQLで教師（teachers）テーブルから「佐藤」という名前を持つ教師を検索する場合、NULL値を持つレコードも含めるにはどう修正すべきですか？

```
SELECT * FROM teachers WHERE teacher_name <> '佐藤花子';
```

問題6-6

attendance（出席）テーブルのすべてのレコードを取得し、コメント（comment）がNULLの場合は「記録なし」と表示するSQLを書いてください。

解答

解答6-1

```
SELECT * FROM attendance WHERE comment IS NULL;
```

解答6-2

```
SELECT * FROM course_schedule
WHERE status = 'cancelled' AND classroom_id IS NOT NULL;
```

解答6-3

```
SELECT * FROM grades WHERE submission_date IS NULL;
```

解答6-4

```
SELECT * FROM attendance
WHERE (status = 'present' OR status = 'late') AND comment IS NULL;
```

または

```

```

```
SELECT * FROM attendance
WHERE status IN ('present', 'late') AND comment IS NULL;
```

解答6-5

```
SELECT * FROM teachers
WHERE teacher_name <> '佐藤花子' OR teacher_name IS NULL;
```

解答6-6

```
SELECT schedule_id, student_id, status,
       IFNULL(comment, '記録なし') AS comment
FROM attendance;
```

まとめ

この章では、データベースにおけるNULL値の概念と、NULL値を扱うための演算子や関数について学びました：

1. **NULL値の概念**：値がない、不明、未設定を表す特殊な値
2. **IS NULL演算子**：NULL値を持つレコードを検索する方法
3. **IS NOT NULL演算子**：NULL値を持たないレコードを検索する方法
4. **NULL値の論理的扱い**：論理演算（AND、OR、NOT）におけるNULLの振る舞い
5. **複合条件**：IS NULL/IS NOT NULLと他の条件の組み合わせ
6. **NULL値の置換**：IFNULL/NVL/COALESCE関数の使用方法
7. **注意点**：NULL値を扱う際の一般的な落とし穴

NULL値の正確な理解と適切な処理は、SQLプログラミングの重要な部分です。不適切なNULL処理は、予期しない結果やバグの原因になります。

次の章では、クエリ結果の並び替えを行うための「ORDER BY：結果の並び替え」について学びます。

---

7. ORDER BY：結果の並び替え

はじめに

これまでの章では、データベースから条件に合ったレコードを取得する方法を学んできました。しかし実際の業務では、取得したデータを見やすく整理する必要があります。例えば：

- 成績を高い順に表示したい
- 学生を名前の五十音順に並べたい
- 日付の新しい順にスケジュールを確認したい

このようなデータの「並び替え」を行うためのSQLコマンドが「ORDER BY」です。この章では、クエリ結果を特定の順序で並べる方法を学びます。

ORDER BYの基本

ORDER BY句は、SELECT文の結果を指定したカラムの値に基づいて並び替えるために使います。

> **用語解説：**
>
> - **ORDER BY**：「〜の順に並べる」という意味のSQLコマンドで、クエリ結果の並び順を指定します。

基本構文

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] ORDER BY 並び替えカラム;
```

ORDER BY句は通常、SELECT文の最後に記述します。

例1：単一カラムでの並び替え

例えば、学生（students）テーブルから、学生名（student_name）の五十音順（辞書順）でデータを取得するには：

```
SELECT * FROM students ORDER BY student_name;
```

実行結果：

student_id	student_name
309	相沢吉夫
303	柴崎春花
306	河田咲奈
305	河口菜恵子
…	…

デフォルトの並び順

ORDER BYを使わない場合、結果の順序は保証されません。多くの場合、データがデータベースに保存された順序で返されますが、これは信頼できるものではありません。

昇順と降順の指定

ORDER BY句では、並び順を「昇順」か「降順」のどちらかで指定できます。

> **用語解説：**

> - **昇順（ASC）**：小さい値から大きい値へ（A→Z、1→9）の順に並べます。
> - **降順（DESC）**：大きい値から小さい値へ（Z→A、9→1）の順に並べます。

構文

```
SELECT カラム名 FROM テーブル名 ORDER BY 並び替えカラム [ASC|DESC];
```

ASC（昇順）がデフォルトのため、省略可能です。

例2：降順での並び替え

例えば、成績（grades）テーブルから、得点（score）の高い順（降順）に成績を取得するには：

```
SELECT * FROM grades ORDER BY score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
…	…	…	…	…	…

複数カラムでの並び替え

複数のカラムを使って並び替えることもできます。最初に指定したカラムで並び替え、値が同じレコードがある場合は次のカラムで並び替えます。

構文

```
SELECT カラム名 FROM テーブル名
ORDER BY 並び替えカラム1 [ASC|DESC], 並び替えカラム2 [ASC|DESC], ...;
```

例3：複数カラムでの並び替え

例えば、成績（grades）テーブルから、課題タイプ（grade_type）の五十音順に並べ、同じ課題タイプ内では得点（score）の高い順に成績を取得するには：

```
SELECT * FROM grades
ORDER BY grade_type ASC, score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	2	実技試験	88.0	100.0	2025-05-18
321	2	実技試験	85.5	100.0	2025-05-18
…	…	…	…	…	…
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…
311	1	レポート1	49.0	50.0	2025-05-08
302	1	レポート1	48.0	50.0	2025-05-10
…	…	…	…	…	…

例4：昇順と降順の混合

各カラムごとに並び順を指定することもできます。例えば、講座ID（course_id）の昇順、評価タイプ（grade_type）の昇順、得点（score）の降順で並べるには：

```
SELECT * FROM grades
ORDER BY course_id ASC, grade_type ASC, score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	レポート1	49.0	50.0	2025-05-08
…	…	…	…	…	…
311	1	中間テスト	95.0	100.0	2025-05-20
…	…	…	…	…	…
301	2	実技試験	88.0	100.0	2025-05-18
…	…	…	…	…	…

NULLの扱い

ORDER BYでNULL値を並び替える場合、データベース製品によって動作が異なります。多くのデータベースでは、NULL値は最小値または最大値として扱われます。

- MySQL/MariaDBでは、NULL値は昇順（ASC）の場合は最小値として（最初に表示）、降順（DESC）の場合は最大値として（最後に表示）扱われます。

一部のデータベース（PostgreSQLなど）では、NULL値の位置を明示的に指定するための「NULLS FIRST」「NULLS LAST」構文がサポートされています。

例5：NULL値の扱い

例えば、出席（attendance）テーブルからコメント（comment）でソートすると、NULLが最初に来ます：

```
SELECT * FROM attendance ORDER BY comment;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	NULL
1	306	present	NULL
...	...	...	NULL
1	308	late	5分遅刻
1	323	late	電車遅延
...	...	...	...

カラム番号を使った並び替え

カラム名の代わりに、SELECT文の結果セットにおけるカラムの位置（番号）を使って並び替えることもできます。最初のカラムは1、2番目のカラムは2、という具合です。

構文

```
SELECT カラム名1, カラム名2, ... FROM テーブル名 ORDER BY カラム位置;
```

例6：カラム番号を使った並び替え

例えば、学生（students）テーブルから学生ID（student_id）と名前（student_name）を取得し、名前（2番目のカラム）で並べ替えるには：

```
SELECT student_id, student_name FROM students ORDER BY 2;
```

この場合、「ORDER BY 2」は「ORDER BY student_name」と同じ意味になります。

> **注意**：カラム番号を使う方法は、カラムの順序を変更すると問題が起きるため、実際の業務では使用を避けた方が良いとされています。

式や関数を使った並び替え

ORDER BY句で式や関数を使うことにより、計算結果に基づいて並び替えることもできます。

例7：式を使った並び替え

例えば、成績（grades）テーブルから、得点の達成率（score/max_score）の高い順に並べるには：

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score/max_score)*100 AS 達成率
FROM grades
ORDER BY (score/max_score) DESC;
```

または

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score/max_score)*100 AS 達成率
FROM grades
ORDER BY 達成率 DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	達成率
311	1	レポート1	49.0	50.0	98.0
320	1	レポート1	48.5	50.0	97.0
311	1	中間テスト	95.0	100.0	95.0
…	…	…	…	…	…

例8：関数を使った並び替え

文字列関数を使って並び替えることもできます。例えば、月名で並べることを考えましょう：

```
SELECT schedule_date, MONTH(schedule_date) AS month
FROM course_schedule
ORDER BY MONTH(schedule_date);
```

実行結果：

schedule_date	month
2025-04-07	4
2025-04-08	4
…	…

schedule_date	month
2025-05-01	5
2025-05-02	5
…	…
2025-06-01	6
…	…

CASE式を使った条件付き並び替え

さらに高度な並び替えとして、CASE式を使って条件に応じた並び順を定義することもできます。

例9：CASE式を使った並び替え

例えば、出席（attendance）テーブルから、出席状況（status）を「欠席→遅刻→出席」の順に優先して表示するには：

```
SELECT * FROM attendance
ORDER BY CASE
           WHEN status = 'absent' THEN 1
           WHEN status = 'late' THEN 2
           WHEN status = 'present' THEN 3
           ELSE 4
         END;
```

実行結果：

schedule_id	student_id	status	comment
1	303	absent	事前連絡あり
1	317	absent	体調不良
…	…	…	…
1	302	late	15分遅刻
1	308	late	5分遅刻
…	…	…	…
1	301	present	NULL
1	306	present	NULL
…	…	…	…

練習問題

問題7-1

students（学生）テーブルから、すべての学生情報を学生名（student_name）の降順（逆五十音順）で取得するSQLを書いてください。

問題7-2

grades（成績）テーブルから、得点（score）が85点以上の成績を得点の高い順に取得するSQLを書いてください。

問題7-3

course_schedule（授業カレンダー）テーブルから、2025年5月の授業スケジュールを日付（schedule_date）の昇順で取得するSQLを書いてください。

問題7-4

teachers（教師）テーブルから、教師IDと名前を取得し、名前（teacher_name）の五十音順で並べるSQLを書いてください。

問題7-5

grades（成績）テーブルから、講座ID（course_id）ごとに、成績を評価タイプ（grade_type）の五十音順に、同じ評価タイプ内では得点（score）の高い順に並べて取得するSQLを書いてください。

問題7-6

attendance（出席）テーブルから、すべての出席情報を出席状況（status）が「absent」「late」「present」の順番で、同じ状態内ではコメント（comment）の有無（NULLが後）で並べて取得するSQLを書いてください。

解答

解答7-1

```sql
SELECT * FROM students ORDER BY student_name DESC;
```

解答7-2

```sql
SELECT * FROM grades WHERE score >= 85 ORDER BY score DESC;
```

解答7-3

```sql
SELECT * FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31'
ORDER BY schedule_date;
```

解答7-4

```
SELECT teacher_id, teacher_name FROM teachers ORDER BY teacher_name;
```

解答7-5

```
SELECT * FROM grades
ORDER BY course_id, grade_type, score DESC;
```

解答7-6

```
SELECT * FROM attendance
ORDER BY
  CASE
    WHEN status = 'absent' THEN 1
    WHEN status = 'late' THEN 2
    WHEN status = 'present' THEN 3
    ELSE 4
  END,
  CASE
    WHEN comment IS NULL THEN 2
    ELSE 1
  END;
```

まとめ

この章では、クエリ結果を特定の順序で並べるための「ORDER BY」句について学びました：

1. **基本的な並び替え**：指定したカラムの値に基づいて結果を並べる方法
2. **昇順と降順**：ASC（昇順）とDESC（降順）の指定方法
3. **複数カラムでの並び替え**：優先順位の高いカラムから順に指定する方法
4. **NULL値の扱い**：NULL値が並び替えでどのように扱われるか
5. **カラム番号**：カラム名の代わりに位置で指定する方法（あまり推奨されない）
6. **式や関数**：計算結果に基づいて並べる方法
7. **CASE式**：条件付きの複雑な並び替え

ORDER BY句は、データを見やすく整理するために非常に重要です。特に大量のデータを扱う場合、適切な並び順はデータの理解を大きく助けます。

次の章では、取得する結果の件数を制限する「LIMIT句：結果件数の制限とページネーション」について学びます。

---

8. LIMIT句：結果件数の制限とページネーション

はじめに

これまでの章では、条件に合うデータを取得し、それを特定の順序で並べる方法を学びました。しかし実際のアプリケーションでは、大量のデータがあるときに、その一部だけを表示したいことがよくあります。例えば：

- 成績上位10件だけを表示したい
- Webページで一度に20件ずつ表示したい（ページネーション）
- 最新の5件のお知らせだけを取得したい

このような「結果の件数を制限する」ためのSQLコマンドが「LIMIT句」です。この章では、クエリ結果の件数を制限する方法と、ページネーションの実装方法を学びます。

LIMIT句の基本

LIMIT句は、SELECT文の結果から指定した件数だけを取得するために使います。

> **用語解説**：
>
> - **LIMIT**：「制限する」という意味のSQLコマンドで、取得する行数を制限します。

基本構文（MySQL/MariaDB）

MySQLやMariaDBでのLIMIT句の基本構文は次のとおりです：

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT 件数;
```

LIMIT句は通常、SELECT文の最後に記述します（ORDER BYの後）。

例1：単純なLIMIT

例えば、学生（students）テーブルから最初の5人だけを取得するには：

```
SELECT * FROM students LIMIT 5;
```

実行結果：

student_id	student_name
301	黒沢春馬
302	新垣愛留
303	柴崎春花
304	森下風凛
305	河口菜恵子

ORDER BYとLIMITの組み合わせ

通常、LIMIT句はORDER BY句と組み合わせて使用します。これにより、「上位N件」「最新N件」などの操作が可能になります。

例2：ORDER BYとLIMITの組み合わせ

例えば、成績（grades）テーブルから得点（score）の高い順に上位3件を取得するには：

```
SELECT * FROM grades ORDER BY score DESC LIMIT 3;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20

例3：最新のレコードを取得

日付でソートして最新のデータを取得することもよくあります。例えば、最新の3つの授業スケジュールを取得するには：

```
SELECT * FROM course_schedule
ORDER BY schedule_date DESC LIMIT 3;
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
95	28	2026-12-21	3	202D	119	scheduled
94	1	2026-12-21	1	102B	101	scheduled
93	14	2026-12-15	4	202D	110	scheduled

OFFSETとページネーション

Webアプリケーションなどでは、大量のデータを「ページ」に分けて表示することがよくあります（ページネーション）。この機能を実現するためには、「OFFSET」（オフセット）という機能が必要です。

> **用語解説**：
>
> - **OFFSET**：「ずらす」という意味で、結果セットの先頭から指定した数だけ行をスキップします。
> - **ページネーション**：大量のデータを複数のページに分割して表示する技術です。

基本構文（MySQL/MariaDB）

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT 件数 OFFSET スキップ数;
```

または、短縮形として：

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT スキップ数, 件数;
```

例4：OFFSETを使ったスキップ

例えば、学生（students）テーブルから6番目から10番目までの学生を取得するには：

```
SELECT * FROM students LIMIT 5 OFFSET 5;
```

または：

```
SELECT * FROM students LIMIT 5, 5;
```

実行結果：

student_id	student_name
306	河田咲奈
307	織田柚夏
308	永田悦子
309	相沢吉夫
310	吉川伽羅

例5：ページネーションの実装

ページネーションを実装する場合、通常は以下の式を使ってOFFSETを計算します：

```
OFFSET = (ページ番号 - 1) × ページあたりの件数
```

例えば、1ページあたり10件表示で、3ページ目のデータを取得するには：

```
SELECT * FROM students ORDER BY student_id LIMIT 10 OFFSET 20;
```

または：

```
SELECT * FROM students ORDER BY student_id LIMIT 20, 10;
```

実行結果：

student_id	student_name
321	井上竜也
322	木村結衣
323	林正義
324	清水香織
325	山田翔太
326	葉山陽太
327	青山凛
328	沢村大和
329	白石優月
330	月岡星奈

LIMIT句を使用する際の注意点

1. ORDER BYの重要性

LIMIT句を使用する場合、通常はORDER BY句も一緒に使うべきです。ORDER BYがなければ、どのレコードが取得されるかは保証されません。

```
-- 良い例：結果が予測可能
SELECT * FROM students ORDER BY student_id LIMIT 5;

-- 悪い例：結果が不確定
SELECT * FROM students LIMIT 5;
```

2. パフォーマンスへの影響

大規模なテーブルで大きなOFFSET値を使用すると、パフォーマンスが低下する可能性があります。これは、データベースがOFFSET分のレコードを読み込んでから破棄する必要があるためです。

3. データベース製品による構文の違い

LIMIT句の構文はデータベース製品によって異なります：

- **MySQL/MariaDB/SQLite**：`LIMIT 件数 OFFSET スキップ数` または `LIMIT スキップ数, 件数`
- **PostgreSQL**：`LIMIT 件数 OFFSET スキップ数`
- **Oracle**：`OFFSET スキップ数 ROWS FETCH NEXT 件数 ROWS ONLY`
- **SQL Server**：`OFFSET スキップ数 ROWS FETCH NEXT 件数 ROWS ONLY` または旧バージョンでは `TOP`句

この章では主にMySQL/MariaDBの構文を使用します。

実践的なページネーションの実装

実際のアプリケーションでページネーションを実装する場合、以下のようなコードになります（疑似コード）：

```
ページ番号 = URLから取得またはデフォルト値（例：1）
1ページあたりの件数 = 設定値（例：10）
総レコード数 = SELECTで取得（COUNT(*)を使用）
総ページ数 = CEILING(総レコード数 ÷ 1ページあたりの件数)
OFFSET = (ページ番号 - 1) × 1ページあたりの件数

SQLクエリ = "SELECT * FROM テーブル ORDER BY カラム LIMIT " + 1ページあたりの件数 + " OFFSET
" + OFFSET
```

例6：総レコード数と総ページ数の取得

総レコード数を取得するには：

```sql
SELECT COUNT(*) AS total_records FROM students;
```

実行結果：

total_records
100

この場合、1ページあたり10件表示なら、総ページ数は10ページ（CEILING(100 ÷ 10)）になります。

練習問題

問題8-1

grades（成績）テーブルから、得点（score）の高い順に上位5件の成績レコードを取得するSQLを書いてください。

問題8-2

course_schedule（授業カレンダー）テーブルから、日付（schedule_date）の新しい順に3件のスケジュールを取得するSQLを書いてください。

問題8-3

students（学生）テーブルを学生ID（student_id）の昇順で並べ、11番目から15番目までの学生（5件）を取得するSQLを書いてください。

問題8-4

teachers（教師）テーブルから、教師名（teacher_name）の五十音順で6番目から10番目までの教師情報を取得するSQLを書いてください。

問題8-5

1ページあたり20件表示で、grades（成績）テーブルの3ページ目のデータを得点（score）の高い順に取得するSQLを書いてください。

問題8-6

course_schedule（授業カレンダー）テーブルから、状態（status）が「scheduled」のスケジュールを日付（schedule_date）の昇順で並べ、先頭から10件スキップして次の5件を取得するSQLを書いてください。

解答

解答8-1

```
SELECT * FROM grades ORDER BY score DESC LIMIT 5;
```

解答8-2

```
SELECT * FROM course_schedule ORDER BY schedule_date DESC LIMIT 3;
```

解答8-3

```
SELECT * FROM students ORDER BY student_id LIMIT 5 OFFSET 10;
```

または

```
SELECT * FROM students ORDER BY student_id LIMIT 10, 5;
```

解答8-4

```
SELECT * FROM teachers ORDER BY teacher_name LIMIT 5 OFFSET 5;
```

または

```
SELECT * FROM teachers ORDER BY teacher_name LIMIT 5, 5;
```

解答8-5

```
SELECT * FROM grades ORDER BY score DESC LIMIT 20 OFFSET 40;
```

または

```
SELECT * FROM grades ORDER BY score DESC LIMIT 40, 20;
```

解答8-6

```
SELECT * FROM course_schedule
WHERE status = 'scheduled'
ORDER BY schedule_date
LIMIT 5 OFFSET 10;
```

または

```
SELECT * FROM course_schedule
WHERE status = 'scheduled'
ORDER BY schedule_date
LIMIT 10, 5;
```

まとめ

この章では、クエリ結果の件数を制限するためのLIMIT句と、ページネーションの実装方法について学びました：

1. **LIMIT句の基本**：指定した件数だけのレコードを取得する方法
2. **ORDER BYとの組み合わせ**：順序付けされた結果から一部だけを取得する方法
3. **OFFSET**：結果の先頭から指定した数だけレコードをスキップする方法
4. **ページネーション**：大量のデータを複数のページに分けて表示する実装方法
5. **注意点**：LIMIT句を使用する際の留意事項
6. **データベース製品による違い**：異なるデータベースでの構文の違い

LIMIT句は特にWebアプリケーションの開発で重要な機能です。大量のデータを効率よく表示するためのページネーション機能を実装するために欠かせません。また、トップN（上位N件）やボトムN（下位N件）のデータを取得する際にも使われます。

次の章では、データの集計分析を行うための「集計関数：COUNT、SUM、AVG、MAX、MIN」について学びます。

SQL学習テキスト 完全版

このテキストは、SQLの基礎から応用まで体系的に学習できるように構成されています。 学校データベースを使った実践的な例題と練習問題を通じて、実務で使えるSQLスキルを身につけることができます。

目次

---

13. JOIN基本：テーブル結合の概念

はじめに

これまでの章では、単一のテーブルからデータを取得する方法を学んできました。しかし実際のデータベースでは、情報は複数のテーブルに分散して保存されています。

例えば、学校データベースでは：

- 学生の基本情報は「students」テーブルに
- 講座の情報は「courses」テーブルに
- 成績データは「grades」テーブルに

このように別々のテーブルに保存されたデータを組み合わせて取得するための機能が「JOIN（結合）」です。この章では、複数のテーブルを結合して必要な情報を取得する基本的な方法を学びます。

JOINとは

JOIN（結合）とは、2つ以上のテーブルを関連付けて、それらのテーブルから情報を組み合わせて取得するためのSQL機能です。

> **用語解説**：
>
> - **JOIN**：「結合する」という意味のSQLコマンドで、複数のテーブルを関連付けてデータを取得するために使います。
> - **結合キー**：テーブル間の関連付けに使用されるカラムのことで、通常は主キーと外部キーの関係にあります。

テーブル結合の必要性

なぜテーブル結合が必要なのでしょうか？いくつかの理由があります：

1. **データの正規化**：データベース設計では、情報の重複を避けるためにデータを複数のテーブルに分割します（これを「正規化」と呼びます）。
2. **データの一貫性**：例えば、教師の名前を1か所（teachersテーブル）だけで管理することで、名前変更時の更新が容易になります。
3. **効率的なデータ管理**：関連するデータをグループ化して管理できます。

例えば、「どの学生がどの講座でどのような成績を取ったか」という情報を取得するには、students、courses、gradesの3つのテーブルを結合する必要があります。

基本的なJOINの種類

SQLでは主に4種類のJOINが使われます：

1. **JOIN**：両方のテーブルで一致するレコードだけを取得(INNER JOIN)
2. **LEFT JOIN**：左テーブルのすべてのレコードと、右テーブルの一致するレコード
3. **RIGHT JOIN**：右テーブルのすべてのレコードと、左テーブルの一致するレコード
4. **FULL JOIN**：両方のテーブルのすべてのレコード（MySQLでは直接サポートされていません）

この章では主にJOINについて学び、次章で他の種類のJOINを学びます。

JOIN（内部結合）

JOINは、最も基本的な結合方法で、両方のテーブルで一致するレコードのみを取得します。

> **用語解説**：
>
> - **JOIN**：「内部結合」とも呼ばれ、結合条件に一致するレコードのみを返します。条件に一致しないレコードは結果に含まれません。

基本構文

```
SELECT カラム名
FROM テーブル1
JOIN テーブル2 ON テーブル1.結合カラム = テーブル2.結合カラム;
```

「ON」の後には結合条件を指定します。これは通常、一方のテーブルの主キーと他方のテーブルの外部キーを一致させる条件です。

> **用語解説**：
>
> - **ON**：「～の条件で」という意味で、JOINの条件を指定するためのキーワードです。

JOINの実践例

例1：講座とその担当教師の情報を取得

講座（courses）テーブルと教師（teachers）テーブルを結合して、各講座の名前と担当教師の名前を取得してみましょう：

```
SELECT c.course_id, c.course_name, t.teacher_name
FROM courses c
JOIN teachers t ON c.teacher_id = t.teacher_id;
```

この例では：

- cとtはそれぞれcourses、teachersテーブルの「テーブル別名」（短縮名）です。
- ON c.teacher_id = t.teacher_idが結合条件です。
- 講座テーブルのteacher_id（外部キー）と教師テーブルのteacher_id（主キー）が一致するレコードが結合されます。

実行結果：

course_id	course_name	teacher_name
1	ITのための基礎知識	寺内鞍
2	UNIX入門	田尻朋美
3	Cプログラミング演習	寺内鞍
4	Webアプリケーション開発	藤本理恵
…	…	…

例2：学生と受講講座の情報を取得

学生（students）テーブルと受講（student_courses）テーブルを結合して、特定の講座を受講している学生の一覧を取得してみましょう：

```
SELECT s.student_id, s.student_name, sc.course_id
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id = '1';
```

この例では、学生テーブルと受講テーブルを学生IDで結合し、講座ID = 1の学生だけを取得しています。

実行結果：

student_id	student_name	course_id
301	黒沢春馬	1
302	新垣愛留	1
303	柴崎春花	1

student_id	student_name	course_id
306	河田咲奈	1
...	...	...

例3：3つのテーブルの結合

より複雑な例として、3つのテーブルを結合してみましょう。学生、講座、成績の情報を組み合わせて表示します：

```
SELECT s.student_name, c.course_name, g.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 = '中間テスト'
ORDER BY g.score DESC;
```

この例では：

1. まず学生テーブルと成績テーブルを結合
2. その結果と講座テーブルを結合
3. 中間テストの成績だけを抽出
4. 点数の高い順にソート

実行結果：

student_name	course_name	score
鈴木健太	ITのための基礎知識	95.0
松本さくら	ITのための基礎知識	93.5
新垣愛留	ITのための基礎知識	92.0
...	...	...

JOIN時のカラム指定

テーブルを結合する際に、特に同じカラム名が両方のテーブルに存在する場合は、カラム名の前にテーブル名（または別名）を付けることで、どのテーブルのカラムを参照しているかを明確にする必要があります。

例えば：

```
SELECT students.student_id, students.student_name
FROM students;
```

これは次のように短縮できます：

```
SELECT s.student_id, s.student_name
FROM students s;
```

ここでsはstudentsテーブルの別名です。

テーブルの別名（エイリアス）

長いテーブル名は別名を使って短くすることができます。これにより、SQLが読みやすくなります。

> **用語解説：**
>
> - **テーブル別名（エイリアス）**：テーブルに一時的につける短い名前で、クエリ内でテーブルを参照するときに使用します。

構文：

```
SELECT t.カラム名
FROM テーブル名 AS t;
```

「AS」キーワードは省略可能です：

```
SELECT t.カラム名
FROM テーブル名 t;
```

例4：別名を使った結合

```
SELECT s.student_name, c.course_name, t.teacher_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
JOIN teachers t ON c.teacher_id = t.teacher_id
WHERE s.student_id = 301;
```

この例では4つのテーブルを結合して、学生ID=301の学生が受講しているすべての講座とその担当教師の情報を取得しています。

実行結果：

student_name	course_name	teacher_name
黒沢春馬	ITのための基礎知識	寺内鞍
黒沢春馬	UNIX入門	田尻朋美
黒沢春馬	クラウドコンピューティング	吉岡由佳

student_name	course_name	teacher_name
...	...	...

結合条件の書き方

結合条件には複数の方法があります：

1. 等価結合（Equi-Join）

最も一般的な結合方法で、カラムの値が等しいことを条件に使います：

```
... ON テーブル1.カラム = テーブル2.カラム
```

2. 非等価結合（Non-Equi Join）

「等しい」以外の条件（<, >, <=, >=など）を使う結合方法です：

```
... ON テーブル1.カラム > テーブル2.カラム
```

3. 複合条件結合

複数の条件を組み合わせる結合方法です：

```
... ON テーブル1.カラム1 = テーブル2.カラム1
    AND テーブル1.カラム2 = テーブル2.カラム2
```

JOINの特徴と注意点

JOINを使用する際には、以下の点に注意が必要です：

1. **一致しないレコードは含まれない**：結合条件に一致しないレコードは結果に含まれません。
2. **NULL値の扱い**：JOIN条件で使用するカラムにNULL値があると、その行は結果に含まれません。
3. **パフォーマンス**：大きなテーブル同士を結合すると、処理に時間がかかることがあります。

練習問題

問題13-1

courses（講座）テーブルとteachers（教師）テーブルを結合して、講座ID（course_id）、講座名（course_name）、担当教師名（teacher_name）を取得するSQLを書いてください。

問題13-2

students（学生）テーブルとstudent_courses（受講）テーブルを結合して、講座ID（course_id）が2の講座を受講している学生の学生ID（student_id）と学生名（student_name）を取得するSQLを書いてください。

問題13-3

course_schedule（授業カレンダー）テーブルとclassrooms（教室）テーブルを結合して、2025年5月20日に行われる授業のスケジュールIDと教室名（classroom_name）を取得するSQLを書いてください。

問題13-4

courses（講座）テーブル、teachers（教師）テーブル、course_schedule（授業カレンダー）テーブルの3つを結合して、2025年5月22日に行われる授業の講座名（course_name）と担当教師名（teacher_name）を取得するSQLを書いてください。

問題13-5

students（学生）テーブル、grades（成績）テーブル、courses（講座）テーブルを結合して、学生ID（student_id）が301の学生の成績情報（講座名、成績タイプ、点数）を取得するSQLを書いてください。

問題13-6

courses（講座）テーブル、course_schedule（授業カレンダー）テーブル、classrooms（教室）テーブル、class_periods（授業時間）テーブルの4つを結合して、2025年5月21日の授業スケジュール（講座名、教室名、開始時間、終了時間）を時間順に取得するSQLを書いてください。

解答

解答13-1

```sql
SELECT c.course_id, c.course_name, t.teacher_name
FROM courses c
JOIN teachers t ON c.teacher_id = t.teacher_id;
```

解答13-2

```sql
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';
```

解答13-3

```sql
SELECT cs.schedule_id, cl.classroom_name
FROM course_schedule cs
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
WHERE cs.schedule_date = '2025-05-20';
```

解答13-4

```
SELECT c.course_name, t.teacher_name
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
JOIN teachers t ON cs.teacher_id = t.teacher_id
WHERE cs.schedule_date = '2025-05-22';
```

解答13-5

```
SELECT c.course_name, g.grade_type, g.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 s.student_id = 301;
```

解答13-6

```
SELECT c.course_name, cl.classroom_name, cp.start_time, cp.end_time
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
JOIN class_periods cp ON cs.period_id = cp.period_id
WHERE cs.schedule_date = '2025-05-21'
ORDER BY cp.start_time;
```

まとめ

この章では、複数のテーブルを結合して情報を取得するための基本的なJOIN操作について学びました：

1. **JOIN（結合）の概念**：複数のテーブルの情報を組み合わせる方法
2. **JOIN（内部結合）**：両方のテーブルで一致するレコードのみを取得
3. **結合条件**：ON句を使って結合条件を指定する方法
4. **テーブル別名**：テーブルに短い名前を付けて使う方法
5. **複数テーブルの結合**：3つ以上のテーブルを結合する方法
6. **結合条件の書き方**：等価結合、非等価結合、複合条件結合

JOINは実際のデータベース操作で非常に重要な機能です。データベースの性能を維持するために、関連情報は複数のテーブルに分散して保存されることが一般的で、必要な情報を取得するためにはこれらのテーブルを結合する必要があります。

次の章では、「テーブル別名：AS句とその省略」について詳しく学び、より効率的なJOINクエリの書き方を学びます。

14. テーブル別名：AS句とその省略

はじめに

前章では複数のテーブルを結合する基本的な方法について学びました。テーブル結合を含むSQLクエリは、複数のテーブル名を扱うため長く複雑になりがちです。また、同じテーブル名やカラム名を繰り返し記述することも多くなります。

SQLでは、このような繰り返しを避け、クエリを簡潔に書くために「テーブル別名（エイリアス）」という機能が用意されています。この章では、テーブルやカラムに一時的な名前（別名）を付ける方法と、それを効果的に使用する方法について学びます。

テーブル別名とは

テーブル別名（エイリアス）とは、SQLクエリの中で使用する仮の短い名前のことです。特に複数のテーブルを結合する場合や同じテーブルを複数回参照する場合に便利です。

> **用語解説**：
>
> - **テーブル別名（エイリアス）**：テーブルに一時的につける短い別名です。クエリ内でテーブルを参照するときに使用します。
> - **AS句**：「～として」という意味のSQLキーワードで、テーブルやカラムに別名を付けるために使います。

AS句を使ったテーブル別名の指定

テーブル別名を指定するには、FROMやJOIN句の中でテーブル名の後にASキーワードとともに別名を書きます。

基本構文

```
SELECT 列リスト
FROM テーブル名 AS 別名
[JOIN 別のテーブル AS 別名 ON 結合条件];
```

例1：AS句を使ったテーブル別名の指定

例えば、コースとその担当教師を取得するクエリでASを使用して別名を付けます：

```
SELECT c.course_id, c.course_name, t.teacher_name
FROM courses AS c
JOIN teachers AS t ON c.teacher_id = t.teacher_id;
```

この例では：

- `courses`テーブルに`c`という別名

- `teachers`テーブルに`t`という別名 を付けています。

AS句の省略

SQLではAS句を省略することができます。これにより、クエリをさらに簡潔に書くことができます。

基本構文（AS省略）

```
SELECT 列リスト
FROM テーブル名 別名
[JOIN 別のテーブル 別名 ON 結合条件];
```

例2：AS句を省略したテーブル別名の指定

先ほどの例からASを省略してみましょう：

```
SELECT c.course_id, c.course_name, t.teacher_name
FROM courses c
JOIN teachers t ON c.teacher_id = t.teacher_id;
```

この書き方は前の例と全く同じ結果を返します。ASを省略してもテーブル名の直後に別名を書くことで同じ効果が得られます。

実行結果：

course_id	course_name	teacher_name
1	ITのための基礎知識	寺内鞍
2	UNIX入門	田尻朋美
3	Cプログラミング演習	寺内鞍
…	…	…

テーブル別名を使う利点

テーブル別名を使用することには、いくつかの利点があります：

1. **クエリの短縮**：長いテーブル名を短い別名で参照できるため、SQLの記述量が減ります。
2. **可読性の向上**：特に複数のテーブルを結合する複雑なクエリでは、どのテーブルのカラムを参照しているかが明確になります。
3. **同一テーブルの複数回使用**：後述する「自己結合」のように、同じテーブルを複数回使う場合に区別するために必須です。
4. **タイピングの労力削減**：繰り返し長いテーブル名を入力する必要がなくなります。

例3：複数テーブル結合での別名の活用

4つのテーブルを結合する複雑なクエリでテーブル別名を活用してみましょう：

```
SELECT s.student_name, c.course_name, cs.schedule_date, cl.classroom_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
JOIN course_schedule cs ON c.course_id = cs.course_id
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
WHERE s.student_id = 301 AND cs.schedule_date >= '2025-05-01';
```

このクエリでは以下の別名を使用しています：

- `students`テーブル → `s`
- `student_courses`テーブル → `sc`
- `courses`テーブル → `c`
- `course_schedule`テーブル → `cs`
- `classrooms`テーブル → `cl`

もし別名を使わないと、以下のような長く読みにくいクエリになります：

```
SELECT students.student_name, courses.course_name, course_schedule.schedule_date,
classrooms.classroom_name
FROM students
JOIN student_courses ON students.student_id = student_courses.student_id
JOIN courses ON student_courses.course_id = courses.course_id
JOIN course_schedule ON courses.course_id = course_schedule.course_id
JOIN classrooms ON course_schedule.classroom_id = classrooms.classroom_id
WHERE students.student_id = 301 AND course_schedule.schedule_date >= '2025-05-01';
```

カラム別名の指定

テーブルだけでなく、カラム（列）にも別名を付けることができます。カラム別名を使うと、結果セットの列見出しをわかりやすい名前に変更できます。

基本構文

```
SELECT
    カラム名 AS 別名,
    計算式 AS 別名
FROM テーブル名;
```

例4：カラム別名の指定

```
SELECT
    student_id AS 学生番号,
    student_name AS 氏名,
    CONCAT(student_id, ': ', student_name) AS 学生情報
FROM students
WHERE student_id < 305;
```

実行結果：

学生番号	氏名	学生情報
301	黒沢春馬	301: 黒沢春馬
302	新垣愛留	302: 新垣愛留
303	柴崎春花	303: 柴崎春花
304	森下風凛	304: 森下風凛

カラム別名でもASは省略可能

カラム別名の場合もAS句は省略可能です：

```
SELECT
    student_id 学生番号,
    student_name 氏名,
    CONCAT(student_id, ': ', student_name) 学生情報
FROM students
WHERE student_id < 305;
```

スペースを含む別名

テーブル名やカラム名に含まれるスペースは通常問題になりますが、別名にスペースを含めたい場合は二重引用符（"）または（データベースによっては）バッククォート（`）で囲むことで指定できます。

例5：スペースを含む別名

```
SELECT
    student_id AS "学生 ID",
    student_name AS "学生 氏名"
FROM students
WHERE student_id < 305;
```

実行結果：

学生 ID	学生 氏名

学生 ID	学生 氏名
301	黒沢春馬
302	新垣愛留
303	柴崎春花
304	森下風凛

OrderByとテーブル別名

一般的に、ORDER BY句ではSELECT句で指定したカラム別名を使用できます：

例6：ORDER BY句での別名の使用

```
SELECT
    student_id AS 学生番号,
    student_name AS 氏名
FROM students
WHERE student_id < 310
ORDER BY 氏名;
```

実行結果（氏名の五十音順）：

学生番号	氏名
309	相沢吉夫
305	河口菜恵子
306	河田咲奈
301	黒沢春馬
304	森下風凛
302	新垣愛留
303	柴崎春花
307	織田柚夏
308	永田悦子

別名の命名規則とベストプラクティス

テーブル別名やカラム別名を付ける際の一般的なルールとベストプラクティスを紹介します：

1. **テーブル別名は短く**：通常、1〜3文字程度の短い名前が好まれます（例：students → s）。
2. **意味のある別名**：テーブルの内容を表す頭文字や略語を使うと良いでしょう（例：teachers → t、course_schedule → cs）。

3. **一貫性**：プロジェクト内で同じテーブルには同じ別名を使うと可読性が向上します。
4. **カラム別名は説明的に**：カラム別名は、その内容がわかりやすい名前にします。特に計算列や関数の結果には説明的な名前を付けましょう。
5. **日本語の別名**：日本語環境では、カラム別名に日本語を使うとレポートが読みやすくなることがあります。

例7：ベストプラクティスに基づく別名

```
SELECT
    c.course_name AS 講座名,
    t.teacher_name AS 担当教員,
    COUNT(sc.student_id) AS 受講者数
FROM courses c
JOIN teachers t ON c.teacher_id = t.teacher_id
JOIN student_courses sc ON c.course_id = sc.course_id
GROUP BY c.course_id, c.course_name, t.teacher_name
ORDER BY 受講者数 DESC, 講座名;
```

実行結果：

講座名	担当教員	受講者数
ITのための基礎知識	寺内鞍	12
データサイエンスとビジネス応用	星野涼子	11
クラウドネイティブアーキテクチャ	吉岡由佳	10
…	…	…

テーブル別名を使用したJOINの応用

ここまで学んだテーブル別名の知識を使って、より実践的なJOINクエリを見てみましょう。

例8：出席状況と成績情報の結合

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    a.status AS 出席状況,
    g.score AS 得点,
    g.score / g.max_score * 100 AS 達成率
FROM students s
JOIN attendance a ON s.student_id = a.student_id
JOIN course_schedule cs ON a.schedule_id = cs.schedule_id
JOIN courses c ON cs.course_id = c.course_id
LEFT JOIN grades g ON s.student_id = g.student_id AND cs.course_id = g.course_id
```

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

このクエリでは：

- 5つのテーブルを結合
- すべてのテーブルに別名を使用
- 結果のカラムにもわかりやすい別名を付与
- 成績は存在しない可能性があるためLEFT JOINを使用

練習問題

問題14-1

courses（講座）テーブルとteachers（教師）テーブルを結合して、講座名（course_name）と担当教師名（teacher_name）を「講座」と「担当者」という別名をつけて取得するSQLを書いてください。テーブル別名も使用してください。

問題14-2

students（学生）テーブルとgrades（成績）テーブルを結合して、学生名、成績タイプ、点数を「学生」「評価種別」「点数」という別名をつけて取得するSQLを書いてください。点数が90点以上のレコードだけを抽出し、点数の高い順にソートしてください。

問題14-3

course_schedule（授業カレンダー）テーブル、courses（講座）テーブル、classrooms（教室）テーブルを結合して、2025年5月21日の授業予定を「時間割」という形で取得するSQLを書いてください。結果には「講座名」「教室」「状態」という別名を付け、テーブル別名も使用してください。

問題14-4

学生の出席状況を確認するため、students（学生）テーブル、attendance（出席）テーブル、course_schedule（授業カレンダー）テーブルを結合して、学生ID=301の出席記録を取得するSQLを書いてください。結果には「日付」「状態」「コメント」という別名をつけ、日付順にソートしてください。

問題14-5

成績の集計情報を取得するため、students（学生）テーブル、grades（成績）テーブル、courses（講座）テーブルを結合し、学生ごとの平均点を計算するSQLを書いてください。結果には「学生名」「受講講座数」「平均点」という別名をつけ、平均点が高い順にソートしてください。

問題14-6

course_schedule（授業カレンダー）テーブル、teachers（教師）テーブル、teacher_unavailability（講師スケジュール管理）テーブルを結合して、講師が不在の日に予定されていた授業（status = 'cancelled'）を取得するSQLを書いてください。結果には「日付」「講師名」「不在理由」という別名をつけ、日付順にソートしてください。

解答

解答14-1

```
SELECT
    c.course_name AS 講座,
    t.teacher_name AS 担当者
FROM courses c
JOIN teachers t ON c.teacher_id = t.teacher_id;
```

解答14-2

```
SELECT
    s.student_name AS 学生,
    g.grade_type AS 評価種別,
    g.score AS 点数
FROM students s
JOIN grades g ON s.student_id = g.student_id
WHERE g.score >= 90
ORDER BY 点数 DESC;
```

解答14-3

```
SELECT
    c.course_name AS 講座名,
    cl.classroom_name AS 教室,
    cs.status AS 状態
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
WHERE cs.schedule_date = '2025-05-21'
ORDER BY cs.period_id;
```

解答14-4

```
SELECT
    cs.schedule_date AS 日付,
    a.status AS 状態,
    a.comment AS コメント
FROM students s
JOIN attendance a ON s.student_id = a.student_id
JOIN course_schedule cs ON a.schedule_id = cs.schedule_id
WHERE s.student_id = 301
ORDER BY 日付;
```

解答14-5

```sql
SELECT
    s.student_name AS 学生名,
    COUNT(DISTINCT g.course_id) AS 受講講座数,
    AVG(g.score) AS 平均点
FROM students s
JOIN grades g ON s.student_id = g.student_id
JOIN courses c ON g.course_id = c.course_id
GROUP BY s.student_id, s.student_name
ORDER BY 平均点 DESC;
```

解答14-6

```sql
SELECT
    cs.schedule_date AS 日付,
    t.teacher_name AS 講師名,
    tu.reason AS 不在理由
FROM course_schedule cs
JOIN teachers t ON cs.teacher_id = t.teacher_id
JOIN teacher_unavailability tu ON t.teacher_id = tu.teacher_id
WHERE cs.status = 'cancelled'
  AND cs.schedule_date BETWEEN tu.start_date AND tu.end_date
ORDER BY 日付;
```

まとめ

この章では、SQLクエリを簡潔で読みやすくするためのテーブル別名とカラム別名について学びました：

1. **テーブル別名の基本**：AS句を使ってテーブルに短い別名を付ける方法
2. **AS句の省略**：テーブル別名を指定する際にASキーワードを省略する書き方
3. **テーブル別名の利点**：クエリの短縮、可読性の向上、同一テーブルの複数回使用
4. **カラム別名**：SELECT句の結果セットの列に別名を付ける方法
5. **スペースを含む別名**：引用符を使って空白を含む別名を指定する方法
6. **ORDER BYでの別名使用**：ソート条件にカラム別名を使用する方法
7. **命名規則とベストプラクティス**：効果的な別名の付け方

テーブル別名とカラム別名はSQLの読みやすさと保守性を大きく向上させる重要な機能です。特に複数のテーブルを結合する複雑なクエリでは、適切な別名を使うことでコードの量が減り、理解しやすくなります。

次の章では、「結合の種類：JOIN、LEFT JOIN、RIGHT JOIN」について学び、さまざまな結合方法とその使い分けについて理解を深めていきます。

---

15. 結合の種類：INNER JOIN、LEFT JOIN、RIGHT JOIN

はじめに

これまでの章で、テーブルを結合する基本的な方法であるINNER JOINと、クエリを簡潔にするためのテーブル別名について学びました。しかし実際のデータベース操作では、テーブル間の関係はさまざまであり、結合方法もそれに合わせて選ぶ必要があります。

例えば：

- 「すべての学生と、存在すれば成績情報も取得したい」
- 「課題を提出していない学生も含めて一覧を取得したい」
- 「担当講座のない教師も含めて教師一覧を表示したい」

このような要件に対応するためには、さまざまな種類の結合方法を理解する必要があります。この章では、主要な結合の種類（INNER JOIN、LEFT JOIN、RIGHT JOIN）とその使い分けについて学びます。

結合の種類とは

SQLでは、テーブルの結合方法として主に以下の種類があります：

1. **INNER JOIN（内部結合）**：両方のテーブルで一致するレコードのみを返します。
2. **LEFT JOIN（左外部結合）**：左テーブルのすべてのレコードと、右テーブルの一致するレコードを返します。
3. **RIGHT JOIN（右外部結合）**：右テーブルのすべてのレコードと、左テーブルの一致するレコードを返します。
4. **FULL JOIN（完全外部結合）**：両方のテーブルのすべてのレコードを返します（MySQLでは直接サポートされていません）。

> **用語解説**：
>
> - **内部結合**：両方のテーブルで条件に一致するレコードだけを返す結合方法。
> - **外部結合**：一方のテーブルのレコードがもう一方のテーブルに一致するレコードがなくても結果に含める結合方法。
> - **左テーブル**：FROM句で最初に指定されるテーブル。
> - **右テーブル**：JOIN句で指定されるテーブル。

INNER JOIN（内部結合）の復習

INNER JOINは最も基本的な結合タイプで、13章で学んだ通り、両方のテーブルで結合条件に一致するレコードのみを返します。

基本構文

```
SELECT カラム名
FROM テーブル1
INNER JOIN テーブル2 ON 結合条件;
```

例1：INNER JOINの基本

学生と彼らが提出した成績情報を結合してみましょう：

```
SELECT s.student_id, s.student_name, g.course_id, g.score
FROM students s
INNER JOIN grades g ON s.student_id = g.student_id
WHERE g.grade_type = '中間テスト'
ORDER BY s.student_id
LIMIT 5;
```

実行結果：

student_id	student_name	course_id	score
301	黒沢春馬	1	85.5
302	新垣愛留	1	92.0
303	柴崎春花	1	78.5
306	河田咲奈	1	88.0
307	織田柚夏	1	76.5

この結果には、成績テーブルに中間テストの記録がある学生だけが含まれています。成績記録のない学生は結果に含まれません。

LEFT JOIN（左外部結合）

LEFT JOINは、左側のテーブル（FROM句のテーブル）のすべてのレコードを返し、右側のテーブル（JOIN句のテーブル）からは一致するレコードだけを返します。右側のテーブルに一致するレコードがない場合、そのカラムはNULLで埋められます。

> **用語解説**：
>
> - **LEFT JOIN**：左テーブルのすべてのレコードと、右テーブルの一致するレコードを返す結合方法です。
> - **NULL埋め**：一致するレコードがない場合、結果セット内の該当カラムはNULL値で埋められます。

基本構文

```
SELECT カラム名
FROM テーブル1
LEFT JOIN テーブル2 ON 結合条件;
```

例2：LEFT JOINの基本

すべての学生と、存在すれば彼らの成績情報を取得してみましょう：

```
SELECT s.student_id, s.student_name, g.course_id, g.score
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id AND g.grade_type = '中間テスト'
ORDER BY s.student_id
LIMIT 10;
```

実行結果：

student_id	student_name	course_id	score
301	黒沢春馬	1	85.5
302	新垣愛留	1	92.0
303	柴崎春花	1	78.5
304	森下風凛	NULL	NULL
305	河口菜恵子	NULL	NULL
306	河田咲奈	1	88.0
307	織田柚夏	1	76.5
308	永田悦子	1	91.0
309	相沢吉夫	NULL	NULL
310	吉川伽羅	1	82.5

この結果には、すべての学生が含まれています。成績記録がない学生（student_id = 304, 305, 309）の場合、course_idとscoreはNULLになっています。

例3：未提出者を見つける

LEFT JOINを使って、特定の課題を提出していない学生を見つけることができます：

```
SELECT s.student_id, s.student_name
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id AND g.grade_type = 'レポート1'
WHERE g.student_id IS NULL AND s.student_id < 320
ORDER BY s.student_id;
```

実行結果：

student_id	student_name
304	森下風凛
305	河口菜恵子
309	相沢吉夫

student_id	student_name
313	佐藤大輔
314	中村彩香
316	渡辺美咲
319	加藤悠真

このクエリは、grades テーブルにレポート1の記録がない学生を探しています。WHEREで「g.student_id IS NULL」を指定することで、結合後にNULLになったレコード（＝提出していない学生）だけを抽出しています。

RIGHT JOIN（右外部結合）

RIGHT JOINは、LEFT JOINの逆で、右側のテーブル（JOIN句のテーブル）のすべてのレコードを返し、左側のテーブル（FROM句のテーブル）からは一致するレコードだけを返します。

> **用語解説：**
>
> - **RIGHT JOIN**：右テーブルのすべてのレコードと、左テーブルの一致するレコードを返す結合方法です。

基本構文

```
SELECT カラム名
FROM テーブル1
RIGHT JOIN テーブル2 ON 結合条件;
```

例4：RIGHT JOINの基本

講座テーブルを基準に、その講座を受講している学生を取得してみましょう：

```
SELECT c.course_id, c.course_name, sc.student_id
FROM student_courses sc
RIGHT JOIN courses c ON sc.course_id = c.course_id AND sc.student_id = 301
ORDER BY c.course_id
LIMIT 10;
```

実行結果：

course_id	course_name	student_id
1	ITのための基礎知識	301
2	UNIX入門	301
3	Cプログラミング演習	NULL

course_id	course_name	student_id
4	Webアプリケーション開発	NULL
5	データベース設計と実装	NULL
6	ネットワークセキュリティ	NULL
7	AI・機械学習入門	NULL
8	モバイルアプリ開発	NULL
9	クラウドコンピューティング	301
10	プロジェクト管理手法	NULL

このクエリの結果には、すべての講座が含まれています。学生ID=301（黒沢春馬）が受講している講座（course_id = 1, 2, 9）では、student_idが表示され、それ以外の講座ではNULLになっています。

LEFT JOINとRIGHT JOINの変換

LEFT JOINとRIGHT JOINは、テーブルの順序を入れ替えることで相互に変換できます。一般的には、LEFT JOINの方が直感的に理解しやすいため、多くの場合はLEFT JOINが好まれます。

次の2つのクエリは同等です：

```
-- LEFT JOINを使用
SELECT *
FROM テーブル1
LEFT JOIN テーブル2 ON 結合条件;

-- RIGHT JOINを使用（テーブルの順序を入れ替え）
SELECT *
FROM テーブル2
RIGHT JOIN テーブル1 ON 結合条件;
```

複数テーブルでの外部結合

外部結合は複数のテーブルを結合する場合にも使用できます。

例5：3つのテーブルを使った外部結合

学生、受講テーブル、講座テーブルを結合して、すべての学生と彼らが受講している講座（あれば）を取得してみましょう：

```
SELECT s.student_id, s.student_name, c.course_id, c.course_name
FROM students s
LEFT JOIN student_courses sc ON s.student_id = sc.student_id
LEFT JOIN courses c ON sc.course_id = c.course_id
WHERE s.student_id BETWEEN 301 AND 305
ORDER BY s.student_id, c.course_id;
```

実行結果：

student_id	student_name	course_id	course_name
301	黒沢春馬	1	ITのための基礎知識
301	黒沢春馬	2	UNIX入門
301	黒沢春馬	9	クラウドコンピューティング
301	黒沢春馬	16	クラウドネイティブアーキテクチャ
…	…	…	…
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
302	新垣愛留	10	プロジェクト管理手法
…	…	…	…
303	柴崎春花	1	ITのための基礎知識
303	柴崎春花	4	Webアプリケーション開発
303	柴崎春花	10	プロジェクト管理手法
…	…	…	…
304	森下風凛	5	データベース設計と実装
304	森下風凛	8	モバイルアプリ開発
304	森下風凛	12	サイバーセキュリティ対策
…	…	…	…
305	河口菜恵子	4	Webアプリケーション開発
305	河口菜恵子	7	AI・機械学習入門
305	河口菜恵子	11	データ分析と可視化
…	…	…	…

このクエリでは2つのLEFT JOINを使用しています。最初のLEFT JOINは学生と受講テーブル、2つ目のLEFT JOINは受講テーブルと講座テーブルを結合しています。

INNER JOINとLEFT JOINの使い分け

INNER JOINとLEFT JOIN（外部結合）は、用途に応じて使い分ける必要があります。以下のような基準で選択すると良いでしょう：

INNER JOINを使う場合

- 両方のテーブルに対応するレコードが存在する場合のみデータを取得したい

- 関連付けられていないデータは不要な場合
- データの存在を確認したい場合

LEFT JOIN（外部結合）を使う場合

- 主テーブルのすべてのレコードを表示したい
- 関連データがあるかどうかにかかわらず、主テーブルの情報は必要な場合
- 欠損データ（未提出、未登録など）を見つけたい場合
- レポート作成時に集計漏れを防ぎたい場合

NULL値の処理に注意

外部結合を使用する場合、NULL値の処理に注意が必要です。特にWHERE句でフィルタリングする場合、通常の比較演算子（=, <, >など）はNULL値に対して常にFALSEを返します。

NULL値を検出するには「IS NULL」演算子を使用し、NULL値を除外するには「IS NOT NULL」演算子を使用します。

例6：NULL値の処理

```
-- 受講している講座がない学生を検索（NULLを検出）
SELECT s.student_id, s.student_name
FROM students s
LEFT JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id IS NULL;

-- 少なくとも1つの講座を受講している学生を検索（NULLを除外）
SELECT DISTINCT s.student_id, s.student_name
FROM students s
LEFT JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_id IS NOT NULL;
```

ON句とWHERE句の違い

LEFT JOINやRIGHT JOINを使用する場合、条件をON句に書くかWHERE句に書くかで結果が大きく変わることがあります。

- **ON句の条件**：結合操作の一部として適用され、外部結合の場合もNULL行を保持します。
- **WHERE句の条件**：結合後のすべての行に適用され、条件を満たさない行は結果から除外されます。

例7：ON句とWHERE句の違い

```
-- ON句に条件を書いた場合（外部結合の特性が保たれる）
SELECT s.student_id, s.student_name, g.score
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id AND g.grade_type = '中間テスト'
ORDER BY s.student_id
LIMIT 5;
```

```sql
-- WHERE句に条件を書いた場合（内部結合と同等になる）
SELECT s.student_id, s.student_name, g.score
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id
WHERE g.grade_type = '中間テスト'
ORDER BY s.student_id
LIMIT 5;
```

最初のクエリでは、すべての学生が結果に含まれ、中間テストのスコアがある場合はその値が表示され、ない場合はNULLになります。

2つ目のクエリでは、WHERE句でg.grade_type = '中間テスト'という条件を指定しているため、中間テストのスコアがない学生は結果から除外されます（実質的にINNER JOINと同等になります）。

練習問題

問題15-1

courses（講座）テーブルとteachers（教師）テーブルを使い、すべての講座とその担当教師（いれば）の情報を取得するSQLを書いてください。LEFT JOINを使用してください。

問題15-2

students（学生）テーブルとgrades（成績）テーブルを使い、講座ID（course_id）= 1の中間テストを受けていない学生を特定するSQLを書いてください。

問題15-3

teachers（教師）テーブルとcourses（講座）テーブルを使い、担当講座がない教師を特定するSQLを書いてください。

問題15-4

course_schedule（授業カレンダー）テーブルとattendance（出席）テーブルを使い、2025年5月20日の各授業に対する出席学生数と欠席学生数を集計するSQLを書いてください。

問題15-5

students（学生）テーブル、student_courses（受講）テーブル、courses（講座）テーブルを使い、各学生が受講している講座の数を取得するSQLを書いてください。受講していない学生も0として表示してください。

問題15-6

course_schedule（授業カレンダー）テーブル、teachers（教師）テーブル、teacher_unavailability（講師スケジュール管理）テーブルを使い、今後の授業予定（schedule_date >= '2025-05-20'）について、担当教師が不在期間と重なっているかどうかをチェックするSQLを書いてください。不在期間と重なっている場合は「要注意」、そうでない場合は「OK」と表示してください。

解答

解答15-1

```
SELECT c.course_id, c.course_name, t.teacher_id, t.teacher_name
FROM courses c
LEFT JOIN teachers t ON c.teacher_id = t.teacher_id
ORDER BY c.course_id;
```

解答15-2

```
SELECT s.student_id, s.student_name
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id
                   AND g.course_id = '1'
                   AND g.grade_type = '中間テスト'
WHERE g.student_id IS NULL
ORDER BY s.student_id;
```

解答15-3

```
SELECT t.teacher_id, t.teacher_name
FROM teachers t
LEFT JOIN courses c ON t.teacher_id = c.teacher_id
WHERE c.teacher_id IS NULL;
```

解答15-4

```
SELECT
    cs.schedule_id,
    c.course_name,
    SUM(CASE WHEN a.status = 'present' THEN 1 ELSE 0 END) AS 出席数,
    SUM(CASE WHEN a.status = 'absent' THEN 1 ELSE 0 END) AS 欠席数,
    SUM(CASE WHEN a.status = 'late' THEN 1 ELSE 0 END) AS 遅刻数,
    COUNT(a.student_id) AS 総数
FROM course_schedule cs
LEFT JOIN attendance a ON cs.schedule_id = a.schedule_id
LEFT JOIN courses c ON cs.course_id = c.course_id
WHERE cs.schedule_date = '2025-05-20'
GROUP BY cs.schedule_id, c.course_name
ORDER BY cs.schedule_id;
```

解答15-5

```
SELECT
    s.student_id,
    s.student_name,
```

```
    COUNT(sc.course_id) AS 受講講座数
FROM students s
LEFT JOIN student_courses sc ON s.student_id = sc.student_id
GROUP BY s.student_id, s.student_name
ORDER BY s.student_id;
```

解答15-6

```
SELECT
    cs.schedule_id,
    cs.schedule_date,
    t.teacher_name,
    c.course_name,
    CASE
        WHEN tu.teacher_id IS NOT NULL THEN '要注意 - ' || tu.reason
        ELSE 'OK'
    END AS 状態
FROM course_schedule cs
INNER JOIN teachers t ON cs.teacher_id = t.teacher_id
INNER JOIN courses c ON cs.course_id = c.course_id
LEFT JOIN teacher_unavailability tu ON cs.teacher_id = tu.teacher_id
                                    AND cs.schedule_date BETWEEN tu.start_date AND
tu.end_date
WHERE cs.schedule_date >= '2025-05-20'
ORDER BY cs.schedule_date, cs.period_id;
```

まとめ

この章では、データベースのさまざまな結合方法について学びました：

1. **INNER JOIN（内部結合）**：両方のテーブルで条件に一致するレコードのみを返す
2. **LEFT JOIN（左外部結合）**：左テーブルのすべてのレコードと、右テーブルの一致するレコードを返す
3. **RIGHT JOIN（右外部結合）**：右テーブルのすべてのレコードと、左テーブルの一致するレコードを返す
4. **結合方法の使い分け**：目的に応じてINNER JOINと外部結合を使い分ける基準
5. **NULL値の処理**：外部結合結果のNULL値を適切に処理する方法
6. **ON句とWHERE句の違い**：条件の記述場所による結果の違い

適切な結合方法を選択することで、より柔軟で正確なデータの抽出が可能になります。特に、データの欠損（未提出、未登録など）を検出したい場合や、すべてのレコードを漏れなく処理したい場合には、外部結合が非常に役立ちます。

次の章では、「自己結合：同一テーブル内での関連付け」について学び、同じテーブル内でのデータ間の関係を扱う方法を学びます。

---

16. 自己結合：同一テーブル内での関連付け

はじめに

これまでの章では、異なるテーブル同士を結合する方法について学びました。しかし実際のデータベース設計では、同じテーブル内にデータ同士の関連がある場合があります。例えば：

- 社員テーブルで「上司」も同じ社員テーブル内の別の社員である
- 部品表で「部品」と「その部品の構成部品」が同じテーブルに格納されている
- 家系図データで「親」と「子」が同じ人物テーブルに格納されている

このような「同一テーブル内のレコード同士の関連」を取り扱うための結合方法が「自己結合（セルフジョイン）」です。この章では、テーブル自身と結合して情報を取得する方法について学びます。

自己結合（セルフジョイン）とは

自己結合とは、テーブルを自分自身と結合する技術です。テーブル内のあるレコードと、同じテーブル内の別のレコードとの関係を表現するために使用されます。

> **用語解説**：
>
> - **自己結合（セルフジョイン）**：同じテーブルを異なる別名で参照し、テーブル自身と結合する手法です。
> - **再帰的関係**：同じエンティティ（テーブル）内での親子関係や階層構造などの関係のこと。

自己結合の基本

自己結合を行うには、同じテーブルを異なる別名で参照する必要があります。これは通常、テーブル別名（エイリアス）を使用して実現します。

基本構文

```
SELECT a.カラム1, a.カラム2, b.カラム1, b.カラム2
FROM テーブル名 a
JOIN テーブル名 b ON a.関連カラム = b.主キー;
```

ここで、`a`と`b`は同じテーブルに対する異なる別名です。

自己結合の実践例

学校データベースでは、明示的な再帰的関係を持つテーブルはありませんが、自己結合の概念を理解するために簡単な例を見てみましょう。

例1：同じ教師が担当する講座を検索

```
SELECT
    c1.course_id AS 講座ID,
    c1.course_name AS 講座名,
    c2.course_id AS 関連講座ID,
```

```
    c2.course_name AS 関連講座名,
    t.teacher_name AS 担当教師
FROM courses c1
JOIN courses c2 ON c1.teacher_id = c2.teacher_id AND c1.course_id < c2.course_id
JOIN teachers t ON c1.teacher_id = t.teacher_id
ORDER BY t.teacher_name, c1.course_id, c2.course_id;
```

このクエリは、同じ教師が担当している講座の組み合わせを検索しています。

- `c1`と`c2`は同じcoursesテーブルの別名
- `c1.teacher_id = c2.teacher_id`で同じ教師を担当している講座を結合
- `c1.course_id < c2.course_id`でペアの重複を避けています（例：講座1と講座2、講座2と講座1が両方表示されることを防ぐ）

実行結果：

講座ID	講座名	関連講座ID	関連講座名	担当教師
4	Webアプリケーション開発	8	モバイルアプリ開発	藤本理恵
4	Webアプリケーション開発	22	フルスタック開発マスタークラス	藤本理恵
8	モバイルアプリ開発	22	フルスタック開発マスタークラス	藤本理恵
1	ITのための基礎知識	3	Cプログラミング演習	寺内鞍
1	ITのための基礎知識	29	コードリファクタリングとクリーンコード	寺内鞍
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	寺内鞍
…	…	…	…	…

例2：同じ日に複数の授業がある教師を検索

```
SELECT
    cs1.schedule_date AS 日付,
    t.teacher_name AS 教師名,
    cs1.period_id AS 時限1,
    c1.course_name AS 講座1,
    cs2.period_id AS 時限2,
    c2.course_name AS 講座2
FROM course_schedule cs1
JOIN course_schedule cs2 ON cs1.teacher_id = cs2.teacher_id
                        AND cs1.schedule_date = cs2.schedule_date
                        AND cs1.period_id < cs2.period_id
JOIN teachers t ON cs1.teacher_id = t.teacher_id
```

```
JOIN courses c1 ON cs1.course_id = c1.course_id
JOIN courses c2 ON cs2.course_id = c2.course_id
WHERE cs1.schedule_date = '2025-05-21'
ORDER BY cs1.schedule_date, t.teacher_name, cs1.period_id, cs2.period_id;
```

このクエリは、同じ日に複数の授業を担当する教師と、その授業の組み合わせを検索しています。

- `cs1`と`cs2`は同じcourse_scheduleテーブルの別名
- `cs1.teacher_id = cs2.teacher_id AND cs1.schedule_date = cs2.schedule_date`で同じ教師が同じ日に担当している授業を結合
- `cs1.period_id < cs2.period_id`でペアの重複を避けています

実行結果（例）：

日付	教師名	時限1	講座1	時限2	講座2
2025-05-21	星野涼子	1	高度データ可視化技術	3	データサイエンスとビジネス応用
2025-05-21	寺内鞍	2	コードリファクタリングとクリーンコード	4	Cプログラミング演習
…	…	…	…	…	…

自己結合の応用：階層構造の表現

自己結合は、階層構造のデータを表現する際に特に役立ちます。例えば、組織図、カテゴリ階層、部品表などです。

ここでは、学校データベースにはない例として、架空の「社員」テーブルを使って階層構造を表現する例を示します：

例3：架空の社員テーブルを使った階層構造の表現

```
-- 架空の社員テーブル
CREATE TABLE employees (
  employee_id INT PRIMARY KEY,
  employee_name VARCHAR(100),
  manager_id INT,
  FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
);

-- サンプルデータ
INSERT INTO employees VALUES (1, '山田太郎', NULL);  -- 社長（上司なし）
INSERT INTO employees VALUES (2, '佐藤次郎', 1);     -- 山田の部下
INSERT INTO employees VALUES (3, '鈴木三郎', 1);     -- 山田の部下
INSERT INTO employees VALUES (4, '高橋四郎', 2);     -- 佐藤の部下
INSERT INTO employees VALUES (5, '田中五郎', 2);     -- 佐藤の部下
INSERT INTO employees VALUES (6, '伊藤六郎', 3);     -- 鈴木の部下
```

```sql
-- 社員と直属の上司を取得
SELECT
    e.employee_id,
    e.employee_name AS 社員名,
    m.employee_name AS 上司名
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id
ORDER BY e.employee_id;
```

結果：

employee_id	社員名	上司名
1	山田太郎	NULL
2	佐藤次郎	山田太郎
3	鈴木三郎	山田太郎
4	高橋四郎	佐藤次郎
5	田中五郎	佐藤次郎
6	伊藤六郎	鈴木三郎

この例では、社員テーブルの`manager_id`が同じテーブルの`employee_id`を参照しています（自己参照）。自己結合を使用して、各社員の上司の名前を取得しています。

例4：架空の社員テーブルを使った部下の一覧表示

```sql
-- ある上司の直属の部下を取得
SELECT
    m.employee_id AS 上司ID,
    m.employee_name AS 上司名,
    e.employee_id AS 部下ID,
    e.employee_name AS 部下名
FROM employees m
JOIN employees e ON m.employee_id = e.manager_id
WHERE m.employee_id = 2
ORDER BY e.employee_id;
```

結果：

上司ID	上司名	部下ID	部下名
2	佐藤次郎	4	高橋四郎
2	佐藤次郎	5	田中五郎

この例では、社員テーブルを`m`（上司）と`e`（部下）として自己結合し、上司ID=2の部下を取得しています。

学校データベースでの自己結合活用例

実際の学校データベースでは、自己結合を利用できる具体的なシナリオを見てみましょう。

例5：同じ教室を使用する授業の組み合わせ

```
SELECT
    cs1.schedule_date AS 日付,
    cs1.classroom_id AS 教室,
    cs1.period_id AS 時限1,
    c1.course_name AS 講座1,
    t1.teacher_name AS 教師1,
    cs2.period_id AS 時限2,
    c2.course_name AS 講座2,
    t2.teacher_name AS 教師2
FROM course_schedule cs1
JOIN course_schedule cs2 ON cs1.classroom_id = cs2.classroom_id
                        AND cs1.schedule_date = cs2.schedule_date
                        AND cs1.period_id < cs2.period_id
JOIN courses c1 ON cs1.course_id = c1.course_id
JOIN courses c2 ON cs2.course_id = c2.course_id
JOIN teachers t1 ON cs1.teacher_id = t1.teacher_id
JOIN teachers t2 ON cs2.teacher_id = t2.teacher_id
WHERE cs1.schedule_date = '2025-05-21'
ORDER BY cs1.classroom_id, cs1.period_id;
```

このクエリは、同じ日に同じ教室で行われる授業の組み合わせを検索しています。教室の利用状況や消毒・清掃のスケジュール計画などに役立ちます。

例6：同じ学生が受講している講座の組み合わせ

```
SELECT
    sc1.student_id,
    s.student_name,
    c1.course_name AS 講座1,
    c2.course_name AS 講座2
FROM student_courses sc1
JOIN student_courses sc2 ON sc1.student_id = sc2.student_id AND sc1.course_id <
sc2.course_id
JOIN students s ON sc1.student_id = s.student_id
JOIN courses c1 ON sc1.course_id = c1.course_id
JOIN courses c2 ON sc2.course_id = c2.course_id
WHERE sc1.student_id = 301
ORDER BY c1.course_name, c2.course_name;
```

このクエリは、学生ID=301の学生が受講しているすべての講座の組み合わせを検索しています。学生の履修パターンの分析や時間割の調整などに役立ちます。

INNER JOINとLEFT JOINの自己結合

自己結合では、INNER JOIN、LEFT JOIN、RIGHT JOINなど、すべての結合タイプを使用できます。結合タイプの選択は、取得したいデータの性質に依存します。

例7：部下のいない社員を見つける（LEFT JOIN）

```
-- 架空の社員テーブルを使用
SELECT
    m.employee_id,
    m.employee_name,
    COUNT(e.employee_id) AS 部下の数
FROM employees m
LEFT JOIN employees e ON m.employee_id = e.manager_id
GROUP BY m.employee_id, m.employee_name
HAVING COUNT(e.employee_id) = 0
ORDER BY m.employee_id;
```

このクエリは、部下のいない社員（つまり、誰も自分を上司として参照していない社員）を検索しています。結果として、employee_id = 4, 5, 6の社員（高橋四郎、田中五郎、伊藤六郎）が表示されるでしょう。

自己結合の注意点

自己結合を使用する際には、以下の点に注意が必要です：

1. **テーブル別名の明確化**：同じテーブルを複数回参照するため、わかりやすいテーブル別名を使用し、混乱を避けましょう。
2. **結合条件の正確性**：自己結合では結合条件が不適切だと、結果が指数関数的に増えることがあります。必要に応じて絞り込み条件を追加しましょう。
3. **重複の排除**：例1や例2で使用した`id1 < id2`のような条件を使って、組み合わせの重複を避けることが重要です。
4. **パフォーマンス**：自己結合は、特に大きなテーブルでは処理が重くなる可能性があります。必要に応じてインデックスを使用して最適化しましょう。

練習問題

問題16-1

教師（teachers）テーブルを自己結合して、教師IDが連続する教師のペア（例：ID 101と102, 102と103）を取得するSQLを書いてください。

問題16-2

course_schedule（授業カレンダー）テーブルを自己結合して、2025年5月15日に同じ教室で行われる授業のペアを時限の昇順で取得するSQLを書いてください。course_id、period_id、classroom_idの3つを表示してください。

問題16-3

生徒（students）テーブルを自己結合して、学生名（student_name）のファーストネーム（名前の最初の文字）が同じ学生のペアを取得するSQLを書いてください。ヒント：SUBSTRING関数を使用します。

問題16-4

course_schedule（授業カレンダー）テーブルを自己結合して、同じ日に同じ講師が担当する授業のペアを見つけるSQLを書いてください。期間が離れていない授業（period_idの差が1または2）のみを対象とします。

問題16-5

grades（成績）テーブルを自己結合して、同じ学生の異なる課題タイプ（grade_type）間で点数差が10点以上あるケースを検出するSQLを書いてください。

問題16-6

以下のような架空の「職階」テーブルを作成し、直接の上下関係（部下と上司）だけでなく、組織階層全体を表示するSQLを書いてください。

```
-- 架空の職階テーブル
CREATE TABLE job_hierarchy (
  level_id INT PRIMARY KEY,
  level_name VARCHAR(50),
  reports_to INT,
  FOREIGN KEY (reports_to) REFERENCES job_hierarchy(level_id)
);

-- データ挿入
INSERT INTO job_hierarchy VALUES (1, '学長', NULL);
INSERT INTO job_hierarchy VALUES (2, '副学長', 1);
INSERT INTO job_hierarchy VALUES (3, '学部長', 2);
INSERT INTO job_hierarchy VALUES (4, '学科長', 3);
INSERT INTO job_hierarchy VALUES (5, '教授', 4);
INSERT INTO job_hierarchy VALUES (6, '准教授', 5);
INSERT INTO job_hierarchy VALUES (7, '講師', 6);
INSERT INTO job_hierarchy VALUES (8, '助教', 7);
```

解答

解答16-1

```
SELECT
    t1.teacher_id AS 教師ID1,
    t1.teacher_name AS 教師名1,
    t2.teacher_id AS 教師ID2,
    t2.teacher_name AS 教師名2
FROM teachers t1
```

```
JOIN teachers t2 ON t1.teacher_id + 1 = t2.teacher_id
ORDER BY t1.teacher_id;
```

解答16-2

```
SELECT
    cs1.course_id AS 講座ID1,
    cs1.period_id AS 時限1,
    cs2.course_id AS 講座ID2,
    cs2.period_id AS 時限2,
    cs1.classroom_id AS 教室
FROM course_schedule cs1
JOIN course_schedule cs2 ON cs1.classroom_id = cs2.classroom_id
                        AND cs1.schedule_date = cs2.schedule_date
                        AND cs1.period_id < cs2.period_id
WHERE cs1.schedule_date = '2025-05-15'
ORDER BY cs1.classroom_id, cs1.period_id, cs2.period_id;
```

解答16-3

```
SELECT
    s1.student_id AS 学生ID1,
    s1.student_name AS 学生名1,
    s2.student_id AS 学生ID2,
    s2.student_name AS 学生名2,
    SUBSTRING(s1.student_name, 1, 1) AS 共通文字
FROM students s1
JOIN students s2 ON SUBSTRING(s1.student_name, 1, 1) = SUBSTRING(s2.student_name,
1, 1)
                AND s1.student_id < s2.student_id
ORDER BY 共通文字, s1.student_id, s2.student_id;
```

解答16-4

```
SELECT
    cs1.schedule_date AS 日付,
    t.teacher_name AS 教師名,
    cs1.period_id AS 時限1,
    cs1.course_id AS 講座ID1,
    cs2.period_id AS 時限2,
    cs2.course_id AS 講座ID2,
    ABS(cs1.period_id - cs2.period_id) AS 時限差
FROM course_schedule cs1
JOIN course_schedule cs2 ON cs1.teacher_id = cs2.teacher_id
                        AND cs1.schedule_date = cs2.schedule_date
                        AND cs1.schedule_id < cs2.schedule_id
```

```
JOIN teachers t ON cs1.teacher_id = t.teacher_id
WHERE ABS(cs1.period_id - cs2.period_id) IN (1, 2)
ORDER BY cs1.schedule_date, t.teacher_name, cs1.period_id;
```

解答16-5

```
SELECT
    g1.student_id,
    s.student_name,
    g1.grade_type AS 評価タイプ1,
    g1.score AS 点数1,
    g2.grade_type AS 評価タイプ2,
    g2.score AS 点数2,
    ABS(g1.score - g2.score) AS 点数差
FROM grades g1
JOIN grades g2 ON g1.student_id = g2.student_id
               AND g1.course_id = g2.course_id
               AND g1.grade_type < g2.grade_type
JOIN students s ON g1.student_id = s.student_id
WHERE ABS(g1.score - g2.score) >= 10
ORDER BY 点数差 DESC, g1.student_id;
```

解答16-6

```
-- 組織階層全体を表示
WITH RECURSIVE hierarchy AS (
  -- 基点（学長）
  SELECT
    level_id,
    level_name,
    reports_to,
    0 AS depth,
    CAST(level_name AS CHAR(200)) AS path
  FROM job_hierarchy
  WHERE reports_to IS NULL

  UNION ALL

  -- 再帰部分
  SELECT
    j.level_id,
    j.level_name,
    j.reports_to,
    h.depth + 1,
    CONCAT(h.path, ' > ', j.level_name)
  FROM job_hierarchy j
  JOIN hierarchy h ON j.reports_to = h.level_id
)
SELECT
```

```
    level_id,
    level_name,
    reports_to,
    depth,
    CONCAT(REPEAT('    ', depth), level_name) AS 階層表示,
    path AS 組織経路
FROM hierarchy
ORDER BY depth, level_id;
```

注意：最後の問題は再帰共通テーブル式（Recursive CTE）を使用しています。これはMySQL 8.0以降でサポートされています。再帰CTEはまだ学習していない高度な内容ですが、階層構造を扱う効果的な方法として参考になります。

まとめ

この章では、同一テーブル内でのレコード間の関連を扱うための「自己結合」について学びました：

1. **自己結合の概念**：テーブルを自分自身と結合して、同一テーブル内のレコード間の関係を表現する方法
2. **基本的な構文**：テーブル別名を使用して同じテーブルを複数回参照する方法
3. **実践的な例**：同じ教師が担当する講座、同じ日の複数の授業など、実用的な自己結合のケース
4. **階層構造の表現**：上司と部下、組織階層など、再帰的な関係の表現方法
5. **さまざまな結合タイプ**：INNER JOINやLEFT JOINなど、自己結合でも利用可能な結合の種類
6. **注意点**：テーブル別名の明確化、結合条件の正確性、重複の排除、パフォーマンスの考慮

自己結合は、階層構造の表現や、同一エンティティ内での関連を取り扱う強力な手法です。特に組織図、部品表、カテゴリ階層、家系図など、再帰的な関係を含むデータモデルで頻繁に活用されます。

次の章では、「複数テーブル結合：3つ以上のテーブルの連結」について学び、より複雑なデータ関係を扱う方法を学びます。

---

17. 複数テーブル結合：3つ以上のテーブルの連結

はじめに

これまでの章では、2つのテーブルを結合する方法や、同じテーブルを自己結合する方法について学びました。しかし、実際のデータベース操作では、3つ以上のテーブルを一度に結合して情報を取得する必要があることが多くあります。

例えば、以下のようなケースを考えてみましょう：

- 「学生の名前、受講している講座名、その担当教師名、教室情報を一度に取得したい」
- 「授業スケジュール、講座情報、教室情報、時間情報をまとめて表示したい」
- 「成績データに学生名、講座名、提出情報を関連付けて分析したい」

これらを実現するには、3つ以上のテーブルを連結する必要があります。この章では、複数のテーブルを効果的に結合する方法について学びます。

複数テーブル結合の基本

3つ以上のテーブルを結合する基本的な考え方は、2つのテーブルの結合を拡張したものです。2つのテーブルを結合した結果に、さらに別のテーブルを結合していきます。

> **用語解説**：
>
> - **複数結合**：3つ以上のテーブルを一度に連結して情報を取得する操作のことです。
> - **結合チェーン**：複数のJOIN句を連続して記述し、テーブルをつなげていく方法です。

基本構文

```
SELECT カラムリスト
FROM テーブル1
JOIN テーブル2 ON 結合条件1
JOIN テーブル3 ON 結合条件2
JOIN テーブル4 ON 結合条件3
...;
```

JOINの種類（INNER JOIN、LEFT JOIN、RIGHT JOINなど）は、各結合ごとに適切に選択できます。

3つのテーブルを結合する例

まずは、3つのテーブルを結合する基本的な例を見てみましょう。

例1：学生、講座、教師の情報を連結する

```
SELECT
    s.student_id,
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    t.teacher_name AS 担当教師
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN courses c ON sc.course_id = c.course_id
JOIN teachers t ON c.teacher_id = t.teacher_id
WHERE s.student_id = 301
ORDER BY c.course_id;
```

このクエリでは：

1. 学生テーブル(`students`)を基点に
2. 受講テーブル(`student_courses`)を結合し
3. 講座テーブル(`courses`)を結合し
4. 教師テーブル(`teachers`)を結合しています

実行結果：

student_id	学生名	講座名	担当教師
301	黒沢春馬	ITのための基礎知識	寺内鞍
301	黒沢春馬	UNIX入門	田尻朋美
301	黒沢春馬	クラウドコンピューティング	吉岡由佳
301	黒沢春馬	クラウドネイティブアーキテクチャ	吉岡由佳
…	…	…	…

4つ以上のテーブルを結合する例

より複雑な情報を取得するために、4つ以上のテーブルを結合してみましょう。

例2：授業スケジュール詳細の取得

```
SELECT
    cs.schedule_date AS 日付,
    cp.start_time AS 開始時間,
    cp.end_time AS 終了時間,
    c.course_name AS 講座名,
    t.teacher_name AS 担当教師,
    cl.classroom_name AS 教室,
    cl.building AS 建物,
    cs.status AS 状態
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
JOIN teachers t ON cs.teacher_id = t.teacher_id
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
JOIN class_periods cp ON cs.period_id = cp.period_id
WHERE cs.schedule_date = '2025-05-21'
ORDER BY cp.start_time, cl.classroom_id;
```

このクエリでは、5つのテーブルを結合して授業スケジュールの詳細情報を取得しています：

1. 授業カレンダーテーブル(`course_schedule`)を基点に
2. 講座テーブル(`courses`)を結合し
3. 教師テーブル(`teachers`)を結合し
4. 教室テーブル(`classrooms`)を結合し
5. 授業時間テーブル(`class_periods`)を結合しています

実行結果（例）：

日付	開始時間	終了時間	講座名	担当教師	教室	建物	状態
2025-05-21	09:00:00	10:30:00	高度データ可視化技術	星野涼子	402Hコンピュータ実習室	4号館	scheduled

日付	開始時間	終了時間	講座名	担当教師	教室	建物	状態
2025-05-21	09:00:00	10:30:00	サイバーセキュリティ対策	深山誠一	301E講義室	3号館	scheduled
2025-05-21	10:40:00	12:10:00	コードリファクタリングとクリーンコード	寺内鞍	101Aコンピュータ実習室	1号館	scheduled
...	...	...	...	...	...	...	...

複数テーブル結合のバリエーション

複数テーブルの結合では、さまざまな結合タイプを組み合わせることができます。

例3：INNER JOINとLEFT JOINの組み合わせ

```
SELECT
    s.student_id,
    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 点数,
    a.status AS 出席状況
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN courses c ON sc.course_id = c.course_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'
LEFT JOIN attendance a ON s.student_id = a.student_id
WHERE s.student_id = 301
ORDER BY c.course_id;
```

このクエリでは、以下のように異なる結合タイプを組み合わせています：

- students、student_courses、coursesテーブルには**INNER JOIN**を使用（関連データが必ず存在するため）
- gradesテーブルには**LEFT JOIN**を使用（中間テストの成績がない可能性があるため）
- attendanceテーブルにも**LEFT JOIN**を使用（出席情報がない可能性があるため）

結合順序とパフォーマンス

複数テーブルを結合する場合、結合の順序はSQLの実行計画に影響しますが、通常はデータベースのオプティマイザが最適な実行順序を決定します。ただし、以下の点に注意すると効率的なクエリを書くことができます：

1. **フィルタリングを早めに行う**：WHERE句での絞り込みを早い段階で行うことで、結合対象のレコード数を減らせます。

2. **小さいテーブルから大きいテーブルへ**：一般的に、小さいテーブルを基点に、より大きいテーブルを結合していく方が効率的です。

3. **結合条件の最適化**：インデックスが設定されているカラムを結合条件に使用すると、パフォーマンスが向上します。

例4：効率的な結合順序とフィルタリングの例

```
SELECT
    cs.schedule_date,
    c.course_name,
    COUNT(a.student_id) AS 出席学生数
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 AND a.status = 'present'
WHERE cs.schedule_date BETWEEN '2025-05-01' AND '2025-05-31'
GROUP BY cs.schedule_id, cs.schedule_date, c.course_name
ORDER BY cs.schedule_date, c.course_name;
```

このクエリでは、以下の効率化を行っています：

- 日付範囲による絞り込みを早い段階で行っている（`course_schedule`テーブルへのWHERE句）
- 比較的小さい`course_schedule`テーブルを基点にしている
- 出席状態のフィルタリングを結合条件に含めている（`a.status = 'present'`）

テーブル別名の重要性

複数テーブルの結合では、テーブル別名（エイリアス）の使用が特に重要になります。テーブル別名を使うことで：

1. コードが簡潔になる
2. 同じカラム名が複数のテーブルに存在する場合の曖昧さが解消される
3. SQLの可読性が向上する

例5：明確なテーブル別名の使用

```
SELECT
    s.student_name AS 学生名,
    c.course_name AS 講座名,
    t.teacher_name AS 教師名,
    cs.schedule_date AS 日付,
    cp.start_time AS 開始時間,
    cl.classroom_name AS 教室
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN courses c ON sc.course_id = c.course_id
JOIN teachers t ON c.teacher_id = t.teacher_id
JOIN course_schedule cs ON c.course_id = cs.course_id
JOIN class_periods cp ON cs.period_id = cp.period_id
```

```
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
WHERE s.student_id = 301
  AND cs.schedule_date >= '2025-05-01'
ORDER BY cs.schedule_date, cp.start_time;
```

このクエリでは、7つのテーブルに明確な別名を付けています：

- students → s
- student_courses → sc
- courses → c
- teachers → t
- course_schedule → cs
- class_periods → cp
- classrooms → cl

複雑な多テーブル結合の実用例

実際の業務でよく使われる、より複雑な多テーブル結合の例を見てみましょう。

例6：学生の成績と出席状況の総合分析

```
SELECT
    s.student_id,
    s.student_name AS 学生名,
    c.course_id,
    c.course_name AS 講座名,
    t.teacher_name AS 担当教師,
    -- 中間テストの点数
    MAX(CASE WHEN g.grade_type = '中間テスト' THEN g.score ELSE NULL END) AS 中間テス
ト,
    -- レポートの点数
    MAX(CASE WHEN g.grade_type = 'レポート1' THEN g.score ELSE NULL END) AS レポート,
    -- 出席情報の集計
    COUNT(DISTINCT cs.schedule_id) AS 授業回数,
    SUM(CASE WHEN a.status = 'present' THEN 1 ELSE 0 END) AS 出席回数,
    SUM(CASE WHEN a.status = 'late' THEN 1 ELSE 0 END) AS 遅刻回数,
    SUM(CASE WHEN a.status = 'absent' THEN 1 ELSE 0 END) AS 欠席回数,
    -- 出席率の計算
    ROUND(
        SUM(CASE
            WHEN a.status = 'present' THEN 1
            WHEN a.status = 'late' THEN 0.5
            ELSE 0
        END) / COUNT(DISTINCT cs.schedule_id) * 100,
    1) AS 出席率
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
JOIN courses c ON sc.course_id = c.course_id
JOIN teachers t ON c.teacher_id = t.teacher_id
LEFT JOIN grades g ON s.student_id = g.student_id AND c.course_id = g.course_id
```

```
LEFT JOIN course_schedule cs ON c.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
GROUP BY s.student_id, s.student_name, c.course_id, c.course_name, t.teacher_name
ORDER BY s.student_id, c.course_id;
```

このクエリは、複数のテーブルを結合して各学生の成績と出席状況を総合的に分析しています。特徴として：

- 複数テーブルの結合
- CASE式を使った条件付き集計
- GROUP BY句による集計
- 計算式を使った派生列（出席率）

練習問題

問題17-1

学生（students）、受講（student_courses）、講座（courses）の3つのテーブルを結合して、学生ID=302の学生が受講しているすべての講座名とその講座IDを取得するSQLを書いてください。

問題17-2

講座（courses）、授業カレンダー（course_schedule）、教室（classrooms）、授業時間（class_periods）の4つのテーブルを結合して、2025年5月22日のすべての授業スケジュールを、開始時間順に取得するSQLを書いてください。結果には講座名、教室名、開始時間、終了時間、担当教師名を含めてください。

問題17-3

学生（students）テーブル、成績（grades）テーブル、講座（courses）テーブルを結合して、「ITのための基礎知識」（course_id=1）の講座を受講している学生の中間テスト成績を、点数の高い順に取得するSQLを書いてください。結果には学生名、得点、および講座名を含めてください。

問題17-4

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、教室（classrooms）テーブルを結合して、教師ID=106（星野涼子）が2025年5月に担当するすべての授業の詳細を取得するSQLを書いてください。結果には授業日、講座名、教室名を含めてください。

問題17-5

学生（students）テーブル、受講（student_courses）テーブル、講座（courses）テーブル、成績（grades）テーブルを結合して、各学生の受講講座数と成績の平均点を計算するSQLを書いてください。成績がない場合も学生と講座は表示してください。結果を平均点の高い順に並べてください。

問題17-6

授業カレンダー（course_schedule）テーブル、講座（courses）テーブル、教師（teachers）テーブル、教師スケジュール管理（teacher_unavailability）テーブルを結合して、2025年5月20日以降に予定されている授業

のうち、担当教師が不在期間と重なっている授業を特定するSQLを書いてください。結果には授業日、講座名、教師名、不在理由を含めてください。

解答

解答17-1

```
SELECT
    s.student_id,
    s.student_name AS 学生名,
    c.course_id,
    c.course_name AS 講座名
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 s.student_id = 302
ORDER BY c.course_id;
```

解答17-2

```
SELECT
    c.course_name AS 講座名,
    cl.classroom_name AS 教室名,
    cp.start_time AS 開始時間,
    cp.end_time AS 終了時間,
    t.teacher_name AS 担当教師名
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
JOIN class_periods cp ON cs.period_id = cp.period_id
JOIN teachers t ON cs.teacher_id = t.teacher_id
WHERE cs.schedule_date = '2025-05-22'
ORDER BY cp.start_time, cl.classroom_id;
```

解答17-3

```
SELECT
    s.student_name AS 学生名,
    g.score AS 得点,
    c.course_name AS 講座名
FROM students s
JOIN grades g ON s.student_id = g.student_id
JOIN courses c ON g.course_id = c.course_id
WHERE c.course_id = '1'
  AND g.grade_type = '中間テスト'
ORDER BY g.score DESC;
```

解答17-4

```
SELECT
    cs.schedule_date AS 授業日,
    c.course_name AS 講座名,
    cl.classroom_name AS 教室名,
    cp.start_time AS 開始時間,
    cp.end_time AS 終了時間
FROM teachers t
JOIN course_schedule cs ON t.teacher_id = cs.teacher_id
JOIN courses c ON cs.course_id = c.course_id
JOIN classrooms cl ON cs.classroom_id = cl.classroom_id
JOIN class_periods cp ON cs.period_id = cp.period_id
WHERE t.teacher_id = 106
  AND cs.schedule_date BETWEEN '2025-05-01' AND '2025-05-31'
ORDER BY cs.schedule_date, cp.start_time;
```

解答17-5

```
SELECT
    s.student_id,
    s.student_name AS 学生名,
    COUNT(DISTINCT sc.course_id) AS 受講講座数,
    AVG(g.score) AS 平均点
FROM students s
LEFT JOIN student_courses sc ON s.student_id = sc.student_id
LEFT JOIN courses c ON sc.course_id = c.course_id
LEFT JOIN grades g ON s.student_id = g.student_id AND c.course_id = g.course_id
GROUP BY s.student_id, s.student_name
ORDER BY 平均点 DESC NULLS LAST;
```

注：NULLS LASTはデータベースによってはサポートされていない場合があります。その場合は以下のように書き換えます：

```
ORDER BY CASE WHEN AVG(g.score) IS NULL THEN 0 ELSE 1 END DESC, AVG(g.score) DESC
```

解答17-6

```
SELECT
    cs.schedule_date AS 授業日,
    c.course_name AS 講座名,
    t.teacher_name AS 教師名,
    tu.reason AS 不在理由
FROM course_schedule cs
JOIN courses c ON cs.course_id = c.course_id
```

```
JOIN teachers t ON cs.teacher_id = t.teacher_id
JOIN teacher_unavailability tu ON cs.teacher_id = tu.teacher_id
WHERE cs.schedule_date >= '2025-05-20'
  AND cs.schedule_date BETWEEN tu.start_date AND tu.end_date
ORDER BY cs.schedule_date, cs.period_id;
```

まとめ

この章では、3つ以上のテーブルを結合して複雑な情報を取得する方法について学びました：

1. **複数テーブル結合の基本構文**：JOINをチェーンさせて3つ以上のテーブルを連結する方法
2. **さまざまな結合タイプの組み合わせ**：INNER JOIN、LEFT JOIN、RIGHT JOINを状況に応じて組み合わせる方法
3. **結合順序とパフォーマンスの考慮点**：効率的なクエリを作成するためのヒント
4. **テーブル別名の重要性**：複数テーブル結合での可読性と明確さの向上
5. **複雑な結合の実用例**：実際の業務で使われるような多テーブル結合の例

複数テーブルの結合は、関連するデータを効率的に取得し、有意義な情報を抽出するための重要なスキルです。特に正規化されたデータベースでは、必要な情報を得るためには複数のテーブルを結合することが必須となります。

次の章では、「サブクエリ：WHERE句内のサブクエリ」について学び、クエリの中に別のクエリを埋め込む高度なテクニックを習得していきます。

SQL学習テキスト 完全版

このテキストは、SQLの基礎から応用まで体系的に学習できるように構成されています。 学校データベースを使った実践的な例題と練習問題を通じて、実務で使えるSQLスキルを身につけることができます。

目次

---

1. SELECT基本：単一テーブルから特定カラムを取得する

はじめに

データベースからデータを取り出す作業は、料理人が大きな冷蔵庫から必要な材料だけを取り出すようなものです。SQLでは、この「取り出す」作業を「SELECT文」で行います。

SELECT文はSQLの中で最も基本的で、最もよく使われる命令です。この章では、単一のテーブル（データの表）から必要な情報だけを取り出す方法を学びます。

基本構文

SELECT文の最も基本的な形は次のとおりです：

```
SELECT カラム名 FROM テーブル名;
```

この文は「テーブル名というテーブルからカラム名という列のデータを取り出してください」という意味です。

> **用語解説**：
>
> - **SELECT**：「選択する」という意味のSQLコマンドで、データを取り出すときに使います。
> - **カラム**：テーブルの縦の列のことで、同じ種類のデータが並んでいます（例：名前のカラム、年齢のカラムなど）。
> - **FROM**：「～から」という意味で、どのテーブルからデータを取るかを指定します。
> - **テーブル**：データベース内の表のことで、行と列で構成されています。

実践例：単一カラムの取得

学校データベースの中の「teachers」テーブル（教師テーブル）から、教師の名前だけを取得してみましょう。

```
SELECT teacher_name FROM teachers;
```

実行結果：

teacher_name
寺内鞍
田尻朋美
内村海凪
藤本理恵
黒木大介
星野涼子
深山誠一
吉岡由佳

teacher_name
山田太郎
佐藤花子
…

これは「teachers」テーブルの「teacher_name」という列（先生の名前）だけを取り出しています。

複数のカラムを取得する

料理に複数の材料が必要なように、データを取り出すときも複数の列が必要なことがよくあります。複数のカラムを取得するには、カラム名をカンマ（,）で区切って指定します。

```
SELECT カラム名1, カラム名2, カラム名3 FROM テーブル名;
```

例えば、教師の番号（ID）と名前を一緒に取得してみましょう：

```
SELECT teacher_id, teacher_name FROM teachers;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍
102	田尻朋美
103	内村海凪
104	藤本理恵
105	黒木大介
106	星野涼子
…	…

すべてのカラムを取得する

テーブルのすべての列を取得したい場合は、アスタリスク（*）を使います。これは「すべての列」を意味するワイルドカードです。

```
SELECT * FROM テーブル名;
```

例：

```
SELECT * FROM teachers;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍
102	田尻朋美
103	内村海凪
…	…

> **注意**：SELECT *は便利ですが、実際の業務では必要なカラムだけを指定する方が良いとされています。これは、データ量が多いときに処理速度が遅くなるのを防ぐためです。

カラムに別名をつける（AS句）

取得したカラムに分かりやすい名前（別名）をつけることができます。これは「AS」句を使います。

```
SELECT カラム名 AS 別名 FROM テーブル名;
```

> **用語解説**：
>
> - **AS**：「～として」という意味で、カラムに別名をつけるときに使います。この別名は結果を表示するときだけ使われます。

例えば、教師IDを「番号」、教師名を「名前」として表示してみましょう：

```
SELECT teacher_id AS 番号, teacher_name AS 名前 FROM teachers;
```

実行結果：

番号	名前
101	寺内鞍
102	田尻朋美
103	内村海凪
…	…

ASは省略することも可能です：

```
SELECT teacher_id 番号, teacher_name 名前 FROM teachers;
```

計算式を使う

SELECT文では、カラムの値を使った計算もできます。例えば、成績テーブルから点数と満点を取得して、達成率（パーセント）を計算してみましょう。

```
SELECT student_id, course_id, grade_type,
       score, max_score,
       (score / max_score) * 100 AS 達成率
FROM grades;
```

実行結果：

student_id	course_id	grade_type	score	max_score	達成率
301	1	中間テスト	85.5	100.0	85.5
302	1	中間テスト	92.0	100.0	92.0
…	…	…	…	…	…

重複を除外する（DISTINCT）

同じ値が複数ある場合に、重複を除いて一意の値だけを表示するには「DISTINCT」キーワードを使います。

> **用語解説**：
>
> - **DISTINCT**：「異なる」「区別された」という意味で、重複する値を除外して一意の値だけを取得します。

例えば、どの講座にどの教師が担当しているかを重複なしで確認してみましょう：

```
SELECT DISTINCT teacher_id FROM courses;
```

実行結果：

teacher_id
101
102
103
104
…

これにより、courses（講座）テーブルで使われている教師IDが重複なく表示されます。

文字列の結合

文字列を結合するには、MySQLでは「CONCAT」関数を使います。例えば、教師のIDと名前を組み合わせて表示してみましょう：

```
SELECT CONCAT('教師ID:', teacher_id, ' 名前:', teacher_name) AS 教師情報
FROM teachers;
```

実行結果：

教師情報
教師ID:101 名前:寺内鞍
教師ID:102 名前:田尻朋美

...

> **用語解説**：
>
> - **CONCAT**：複数の文字列を一つにつなげる関数です。

SELECT文と終了記号

SQLの文は通常、セミコロン（;）で終わります。これは「この命令はここで終わりです」という合図です。

複数のSQL文を一度に実行する場合は、それぞれの文の最後にセミコロンをつけます。

練習問題

問題1-1

students（学生）テーブルから、すべての学生の名前（student_name）を取得するSQLを書いてください。

問題1-2

classrooms（教室）テーブルから、教室ID（classroom_id）と教室名（classroom_name）を取得するSQLを書いてください。

問題1-3

courses（講座）テーブルから、すべての列（カラム）を取得するSQLを書いてください。

問題1-4

class_periods（授業時間）テーブルから、時限ID（period_id）、開始時間（start_time）、終了時間（end_time）を取得し、開始時間には「開始」、終了時間には「終了」という別名をつけるSQLを書いてください。

問題1-5

gradesテーブルから、学生ID（student_id）、講座ID（course_id）、評価タイプ（grade_type）、得点（score）、満点（max_score）、そして得点を満点で割って100を掛けた値を「パーセント」という別名で取得するSQLを書いてください。

問題1-6

course_scheduleテーブルから、schedule_date（予定日）カラムだけを重複なしで取得するSQLを書いてください。

解答

解答1-1

```
SELECT student_name FROM students;
```

解答1-2

```
SELECT classroom_id, classroom_name FROM classrooms;
```

解答1-3

```
SELECT * FROM courses;
```

解答1-4

```
SELECT period_id, start_time AS 開始, end_time AS 終了 FROM class_periods;
```

解答1-5

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score / max_score) * 100 AS パーセント
FROM grades;
```

解答1-6

```
SELECT DISTINCT schedule_date FROM course_schedule;
```

まとめ

この章では、SQLのSELECT文の基本を学びました：

1. 単一カラムの取得: `SELECT カラム名 FROM テーブル名;`
2. 複数カラムの取得: `SELECT カラム名1, カラム名2 FROM テーブル名;`
3. すべてのカラムの取得: `SELECT * FROM テーブル名;`
4. カラムに別名をつける: `SELECT カラム名 AS 別名 FROM テーブル名;`
5. 計算式を使う: `SELECT カラム名, (計算式) AS 別名 FROM テーブル名;`
6. 重複を除外する: `SELECT DISTINCT カラム名 FROM テーブル名;`
7. 文字列の結合: `SELECT CONCAT(文字列1, カラム名, 文字列2) FROM テーブル名;`

これらの基本操作を使いこなせるようになれば、データベースから必要な情報を効率よく取り出せるようになります。次の章では、WHERE句を使って条件に合ったデータだけを取り出す方法を学びます。

---

2. WHERE句：条件に合ったレコードを絞り込む

はじめに

前章では、テーブルからデータを取得する基本的な方法を学びました。しかし実際の業務では、すべてのデータではなく、特定の条件に合ったデータだけを取得したいことがほとんどです。

例えば、「全生徒の情報」ではなく「特定の学科の生徒だけ」や「成績が80点以上の学生だけ」といった形で、データを絞り込みたい場合があります。

このような場合に使用するのが「WHERE句」です。WHERE句は、SELECTコマンドの後に追加して使い、条件に合致するレコード（行）だけを取得します。

基本構文

WHERE句の基本的な形は次のとおりです：

```
SELECT カラム名 FROM テーブル名 WHERE 条件式;
```

> **用語解説**：
>
> - **WHERE**：「～の場所で」「～の条件で」という意味のSQLコマンドで、条件に合うデータだけを抽出するために使います。
> - **条件式**：データが満たすべき条件を指定するための式です。例えば「age > 20」（年齢が20より大きい）などです。
> - **レコード**：テーブルの横の行のことで、1つのデータの集まりを表します。

基本的な比較演算子

WHERE句では、様々な比較演算子を使って条件を指定できます：

演算子	意味	例
=	等しい	age = 25

演算子	意味	例
<>	等しくない（≠と同じ）	gender <> 'male'
>	より大きい	score > 80
<	より小さい	price < 1000
>=	以上	height >= 170
<=	以下	weight <= 70

実践例：基本的な条件での絞り込み

例1：等しい（=）

例えば、教師ID（teacher_id）が101の教師のみを取得するには：

```
SELECT * FROM teachers WHERE teacher_id = 101;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍

例2：より大きい（>）

成績（grades）テーブルから、90点を超える成績だけを取得するには：

```
SELECT * FROM grades WHERE score > 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

例3：等しくない（<>）

講座IDが3ではない講座に関する成績を取得するには：

```
SELECT * FROM grades WHERE course_id <> '3';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
301	2	実技試験	88.0	100.0	2025-05-18
...	...	...	...	...	...

文字列の比較

テキスト（文字列）を条件にする場合は、シングルクォーテーション（'）またはダブルクォーテーション（"）で囲みます。MySQLではどちらも使えますが、多くの場合シングルクォーテーションが推奨されます。

```
SELECT * FROM テーブル名 WHERE テキストカラム = 'テキスト値';
```

例えば、教師名（teacher_name）が「田尻朋美」の教師を検索するには：

```
SELECT * FROM teachers WHERE teacher_name = '田尻朋美';
```

実行結果：

teacher_id	teacher_name
102	田尻朋美

日付の比較

日付の比較も同様にシングルクォーテーションで囲みます。日付の形式はデータベースの設定によって異なりますが、一般的にはISO形式（YYYY-MM-DD）が使われます。

```
SELECT * FROM テーブル名 WHERE 日付カラム = '日付';
```

例えば、2025年5月20日に提出された成績を検索するには：

```
SELECT * FROM grades WHERE submission_date = '2025-05-20';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
303	1	中間テスト	78.5	100.0	2025-05-20
...	...	...	...	...	...

また、日付同士の大小関係も比較できます：

```
SELECT * FROM grades WHERE submission_date < '2025-05-15';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	レポート1	45.0	50.0	2025-05-10
302	1	レポート1	48.0	50.0	2025-05-10
304	5	ER図作成課題	27.5	30.0	2025-05-14
...	...	...	...	...	...

複数の条件を指定する（次章の内容）

WHERE句では、複数の条件を組み合わせることもできます。この詳細は次章「論理演算子：AND、OR、NOTを使った複合条件」で説明します。

例えば、講座IDが1で、かつ、得点が90以上の成績を取得するには：

```
SELECT * FROM grades WHERE course_id = '1' AND score >= 90;
```

この例の詳細な説明は次章で行います。

練習問題

問題2-1

students（学生）テーブルから、学生ID（student_id）が「310」の学生情報を取得するSQLを書いてください。

問題2-2

classrooms（教室）テーブルから、収容人数（capacity）が30人より多い教室の情報をすべて取得するSQLを書いてください。

問題2-3

courses（講座）テーブルから、教師ID（teacher_id）が「105」の講座情報を取得するSQLを書いてください。

問題2-4

course_schedule（授業カレンダー）テーブルから、「2025-05-15」の授業スケジュールをすべて取得するSQLを書いてください。

問題2-5

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」で、得点（score）が80点未満の成績を取得するSQLを書いてください。

問題2-6

teachers（教師）テーブルから、教師ID（teacher_id）が「101」ではない教師の名前を取得するSQLを書いてください。

解答

解答2-1

```sql
SELECT * FROM students WHERE student_id = 310;
```

解答2-2

```sql
SELECT * FROM classrooms WHERE capacity > 30;
```

解答2-3

```sql
SELECT * FROM courses WHERE teacher_id = 105;
```

解答2-4

```sql
SELECT * FROM course_schedule WHERE schedule_date = '2025-05-15';
```

解答2-5

```sql
SELECT * FROM grades WHERE grade_type = '中間テスト' AND score < 80;
```

解答2-6

```
SELECT teacher_name FROM teachers WHERE teacher_id <> 101;
```

まとめ

この章では、WHERE句を使って条件に合ったレコードを取得する方法を学びました：

1. 基本的な比較演算子（=, <>, >, <, >=, <=）の使い方
2. 数値による条件絞り込み
3. 文字列（テキスト）による条件絞り込み
4. 日付による条件絞り込み

WHERE句は、大量のデータから必要な情報だけを取り出すための非常に重要な機能です。実際のデータベース操作では、この条件絞り込みを頻繁に使います。

次の章では、複数の条件を組み合わせるための「論理演算子（AND、OR、NOT）」について学びます。

---

3. 論理演算子：AND、OR、NOTを使った複合条件

はじめに

前章では、WHERE句を使って単一の条件でデータを絞り込む方法を学びました。しかし実際の業務では、より複雑な条件でデータを絞り込む必要があることがよくあります。

例えば：

- 「成績が80点以上**かつ**出席率が90%以上の学生」
- 「数学**または**英語の成績が優秀な学生」
- 「課題を**まだ提出していない**学生」

このような複合条件を指定するために使うのが論理演算子です。主な論理演算子は次の3つです：

- **AND**：両方の条件を満たす（かつ）
- **OR**：いずれかの条件を満たす（または）
- **NOT**：条件を満たさない（～ではない）

AND演算子

AND演算子は、指定した**すべての条件を満たす**レコードだけを取得したいときに使います。

> **用語解説**：
>
> - **AND**：「かつ」「そして」という意味の論理演算子です。複数の条件をすべて満たすデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 条件1 AND 条件2;
```

例：ANDを使った複合条件

例えば、中間テストで90点以上かつ満点が100点の成績レコードを取得するには：

```
SELECT * FROM grades
WHERE score >= 90 AND max_score = 100;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

この例では、「score >= 90」と「max_score = 100」の両方の条件を満たすレコードだけが取得されます。

3つ以上の条件の組み合わせ

ANDを使って3つ以上の条件を組み合わせることもできます：

```
SELECT * FROM grades
WHERE score >= 90 AND max_score = 100 AND grade_type = '中間テスト';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

OR演算子

OR演算子は、指定した条件の**いずれか一つでも満たす**レコードを取得したいときに使います。

> **用語解説**：
>
> - **OR**：「または」「もしくは」という意味の論理演算子です。複数の条件のうち少なくとも1つを満たすデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 条件1 OR 条件2;
```

例：ORを使った複合条件

例えば、教師IDが101または102の講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id = 101 OR teacher_id = 102;
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
2	UNIX入門	102
3	Cプログラミング演習	101
29	コードリファクタリングとクリーンコード	101
40	ソフトウェアアーキテクチャパターン	102
…	…	…

この例では、「teacher_id = 101」または「teacher_id = 102」のいずれかの条件を満たすレコードが取得されます。

NOT演算子

NOT演算子は、指定した条件を**満たさない**レコードを取得したいときに使います。

> **用語解説**：
>
> - **NOT**：「～ではない」という意味の論理演算子です。条件を否定して、その条件を満たさないデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE NOT 条件;
```

例：NOTを使った否定条件

例えば、完了（completed）状態ではない授業スケジュールを取得するには：

```
SELECT * FROM course_schedule
WHERE NOT status = 'completed';
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
45	11	2025-05-20	5	401G	106	scheduled
46	12	2025-05-21	1	301E	107	scheduled
50	2	2025-05-23	3	101A	102	cancelled
…	…	…	…	…	…	…

この例では、statusが「completed」ではないレコード（scheduled状態やcancelled状態）が取得されます。

NOT演算子は、「～ではない」という否定の条件を作るために使われます。例えば次の2つの書き方は同じ意味になります：

```
SELECT * FROM teachers WHERE NOT teacher_id = 101;
SELECT * FROM teachers WHERE teacher_id <> 101;
```

複合論理条件（AND、OR、NOTの組み合わせ）

AND、OR、NOTを組み合わせて、より複雑な条件を指定することもできます。

例：ANDとORの組み合わせ

例えば、「成績が90点以上で中間テストである」または「成績が45点以上でレポートである」レコードを取得するには：

```
SELECT * FROM grades
WHERE (score >= 90 AND grade_type = '中間テスト')
   OR (score >= 45 AND grade_type = 'レポート1');
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
302	1	レポート1	48.0	50.0	2025-05-10
308	1	レポート1	47.0	50.0	2025-05-09

student_id	course_id	grade_type	score	max_score	submission_date
...	...	...	...	...	...

優先順位と括弧の使用

論理演算子を組み合わせる場合、演算子の優先順位に注意が必要です。基本的に**ANDはORよりも優先順位が高い**です。つまり、ANDが先に処理されます。

例えば：

```
WHERE 条件1 OR 条件2 AND 条件3
```

これは次のように解釈されます：

```
WHERE 条件1 OR (条件2 AND 条件3)
```

意図した条件と異なる場合は、**括弧（）を使って明示的にグループ化**することが重要です：

```
WHERE (条件1 OR 条件2) AND 条件3
```

例：括弧を使った条件のグループ化

教師IDが101または102で、かつ、講座名に「プログラミング」という単語が含まれる講座を取得するには：

```sql
SELECT * FROM courses
WHERE (teacher_id = 101 OR teacher_id = 102)
  AND course_name LIKE '%プログラミング%';
```

実行結果：

course_id	course_name	teacher_id
3	Cプログラミング演習	101
...	...	...

この例では、括弧を使って「teacher_id = 101 OR teacher_id = 102」の部分をグループ化し、その条件と「course_name LIKE '%プログラミング%'」の条件をANDで結合しています。

練習問題

問題3-1

grades（成績）テーブルから、課題タイプ（grade_type）が「中間テスト」かつ点数（score）が85点以上のレコードを取得するSQLを書いてください。

問題3-2

classrooms（教室）テーブルから、収容人数（capacity）が40人以下または建物（building）が「1号館」の教室を取得するSQLを書いてください。

問題3-3

teachers（教師）テーブルから、教師ID（teacher_id）が101、102、103ではない教師の情報を取得するSQLを書いてください。

問題3-4

course_schedule（授業カレンダー）テーブルから、2025年5月20日の授業で、時限（period_id）が1か2で、かつ状態（status）が「scheduled」の授業を取得するSQLを書いてください。

問題3-5

students（学生）テーブルから、学生名（student_name）に「田」または「山」を含む学生を取得するSQLを書いてください。

問題3-6

grades（成績）テーブルから、提出日（submission_date）が2025年5月15日以降で、かつ（「中間テスト」で90点以上または「レポート1」で45点以上）の成績を取得するSQLを書いてください。

解答

解答3-1

```sql
SELECT * FROM grades
WHERE grade_type = '中間テスト' AND score >= 85;
```

解答3-2

```sql
SELECT * FROM classrooms
WHERE capacity <= 40 OR building = '1号館';
```

解答3-3

```sql
SELECT * FROM teachers
WHERE NOT (teacher_id = 101 OR teacher_id = 102 OR teacher_id = 103);
```

または

```
SELECT * FROM teachers
WHERE teacher_id <> 101 AND teacher_id <> 102 AND teacher_id <> 103;
```

解答3-4

```
SELECT * FROM course_schedule
WHERE schedule_date = '2025-05-20'
  AND (period_id = 1 OR period_id = 2)
  AND status = 'scheduled';
```

解答3-5

```
SELECT * FROM students
WHERE student_name LIKE '%田%' OR student_name LIKE '%山%';
```

解答3-6

```
SELECT * FROM grades
WHERE submission_date >= '2025-05-15'
  AND ((grade_type = '中間テスト' AND score >= 90)
       OR (grade_type = 'レポート1' AND score >= 45));
```

まとめ

この章では、論理演算子（AND、OR、NOT）を使って複合条件を作る方法を学びました：

1. **AND**：すべての条件を満たすレコードを取得（条件1 AND 条件2）
2. **OR**：いずれかの条件を満たすレコードを取得（条件1 OR 条件2）
3. **NOT**：指定した条件を満たさないレコードを取得（NOT 条件）
4. **複合条件**：AND、OR、NOTを組み合わせたより複雑な条件
5. **括弧（）**：条件をグループ化して優先順位を明示的に指定

これらの論理演算子を使いこなすことで、より複雑で細かな条件でデータを絞り込むことができるようになります。実際のデータベース操作では、複数の条件を組み合わせることが頻繁にあるため、この章で学んだ内容は非常に重要です。

次の章では、テキストデータに対する検索を行う「パターンマッチング」について学びます。

---

4. パターンマッチング：LIKE演算子と%、_ワイルドカード

はじめに

前章までは、データの完全一致や数値の比較といった条件での絞り込みを学びました。しかし実際の業務では、もっと柔軟な検索が必要なケースがあります。例えば：

- 「山」で始まる名前の学生を検索したい
- 「プログラミング」という単語を含む講座名を探したい
- 電話番号の一部だけ覚えているデータを探したい

このような「部分一致」や「パターン一致」の検索を行うためのSQLの機能が「パターンマッチング」です。この章では、パターンマッチングを行うための「LIKE演算子」と「ワイルドカード文字」について学びます。

LIKE演算子の基本

LIKE演算子は、文字列のパターンマッチングを行うための演算子です。WHERE句と組み合わせて使用します。

> **用語解説**：
>
> - **LIKE**：「～のような」という意味の演算子で、パターンに一致する文字列を検索します。
> - **パターンマッチング**：完全一致ではなく、一定のパターンに合致するデータを検索する方法です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 文字列カラム LIKE 'パターン';
```

パターンには、通常の文字に加えて、特別な意味を持つ「ワイルドカード文字」を使用できます。

ワイルドカード文字

SQLでは主に2つのワイルドカード文字があります：

1. **%（パーセント）**：0文字以上の任意の文字列に一致します。
2. _**（アンダースコア）**：任意の1文字に一致します。

> **用語解説**：
>
> - **ワイルドカード**：任意の文字や文字列に一致する特殊な文字記号です。トランプのジョーカーのように、様々な値に代用できます。

LIKE演算子の使い方：実践例

例1：%（パーセント）を使ったパターンマッチング

「〜で始まる」パターン：前方一致

例えば、「山」で始まる学生名を検索するには：

```
SELECT * FROM students WHERE student_name LIKE '山%';
```

実行結果：

student_id	student_name
312	山本裕子
325	山田翔太
…	…

ここでの「山%」は「山で始まり、その後に0文字以上の任意の文字が続く」という意味です。

「〜で終わる」パターン：後方一致

例えば、「子」で終わる教師名を検索するには：

```
SELECT * FROM teachers WHERE teacher_name LIKE '%子';
```

実行結果：

teacher_id	teacher_name
102	田尻朋美
106	星野涼子
108	吉岡由佳
110	佐藤花子
…	…

「〜を含む」パターン：部分一致

例えば、「プログラミング」という単語を含む講座名を検索するには：

```
SELECT * FROM courses WHERE course_name LIKE '%プログラミング%';
```

実行結果：

course_id	course_name	teacher_id

course_id	course_name	teacher_id
3	Cプログラミング演習	101
14	IoTデバイスプログラミング実践	110
…	…	…

例2：_（アンダースコア）を使ったパターンマッチング

アンダースコアは任意の1文字に一致します。例えば、教室IDが「10_A」パターン（最初の2文字が「10」、3文字目が任意の1文字、最後が「A」）の教室を検索するには：

```
SELECT * FROM classrooms WHERE classroom_id LIKE '10_A';
```

実行結果：

classroom_id	classroom_name	capacity	building	facilities
101A	1号館コンピュータ実習室A	30	1号館	パソコン30台、プロジェクター
…	…	…	…	…

例3：%と_の組み合わせ

ワイルドカード文字は組み合わせて使うこともできます。例えば、「2文字目が田」の学生を検索するには：

```
SELECT * FROM students WHERE student_name LIKE '_田%';
```

実行結果：

student_id	student_name
321	井上竜也
384	櫻井翼
…	…

NOT LIKEを使った否定形のパターンマッチング

特定のパターンに一致しないレコードを検索したい場合は、「NOT LIKE」を使います。

> **用語解説**：
>
> - **NOT LIKE**：「～のパターンに一致しない」という意味で、指定したパターンに一致しないデータを検索します。

例えば、講座名に「入門」を含まない講座を検索するには：

```
SELECT * FROM courses WHERE course_name NOT LIKE '%入門%';
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
3	Cプログラミング演習	101
...	...	...

エスケープ文字の使用

もし検索したいパターンに「%」や「_」自体が含まれている場合は、それらを特別な文字としてではなく、通常の文字として扱うために「エスケープ文字」を使います。

> **用語解説**：
>
> - **エスケープ文字**：特別な意味を持つ文字を通常の文字として扱うための印です。

MySQLでは、バックスラッシュ（\）をエスケープ文字として使用できます。例えば、「50%」という値そのものを検索するには：

```
SELECT * FROM テーブル名 WHERE カラム LIKE '50\%';
```

または、ESCAPE句を使って明示的にエスケープ文字を指定することもできます：

```
SELECT * FROM テーブル名 WHERE カラム LIKE '50!%' ESCAPE '!';
```

この例では「!」をエスケープ文字として指定しています。

大文字と小文字の区別

MySQLのデフォルト設定では、LIKE演算子は大文字と小文字を区別しません（大文字小文字を同じものとして扱います）。

例えば、次の2つのクエリは同じ結果を返します：

```
SELECT * FROM courses WHERE course_name LIKE '%web%';
SELECT * FROM courses WHERE course_name LIKE '%Web%';
```

もし大文字と小文字を区別した検索が必要な場合は、「BINARY」キーワードを使用します：

```

```

```
SELECT * FROM courses WHERE course_name LIKE BINARY '%Web%';
```

この場合、「Web」は「web」とは一致しません。

複合条件との組み合わせ

LIKE演算子は、これまで学んだAND、OR、NOTなどの論理演算子と組み合わせて使うこともできます。

例えば、「田」で始まる名前で、かつ教師IDが102から105の間の教師を検索するには：

```
SELECT * FROM teachers
WHERE teacher_name LIKE '田%'
  AND teacher_id BETWEEN 102 AND 105;
```

実行結果：

teacher_id	teacher_name
102	田尻朋美
...	...

練習問題

問題4-1

students（学生）テーブルから、学生名（student_name）が「佐藤」で始まる学生の情報をすべて取得するSQLを書いてください。

問題4-2

courses（講座）テーブルから、講座名（course_name）に「データ」という単語を含む講座の情報を取得するSQLを書いてください。

問題4-3

classrooms（教室）テーブルから、教室名（classroom_name）が「コンピュータ実習室」で終わる教室の情報を取得するSQLを書いてください。

問題4-4

teachers（教師）テーブルから、教師名（teacher_name）の2文字目が「木」である教師の情報を取得するSQLを書いてください。

問題4-5

courses（講座）テーブルから、講座名（course_name）に「入門」または「基礎」を含む講座を取得するSQLを書いてください。

問題4-6

students（学生）テーブルから、学生名（student_name）が「山」で始まり、かつ「子」で終わらない学生を取得するSQLを書いてください。

解答

解答4-1

```
SELECT * FROM students WHERE student_name LIKE '佐藤%';
```

解答4-2

```
SELECT * FROM courses WHERE course_name LIKE '%データ%';
```

解答4-3

```
SELECT * FROM classrooms WHERE classroom_name LIKE '%コンピュータ実習室';
```

解答4-4

```
SELECT * FROM teachers WHERE teacher_name LIKE '_木%';
```

解答4-5

```
SELECT * FROM courses
WHERE course_name LIKE '%入門%' OR course_name LIKE '%基礎%';
```

解答4-6

```
SELECT * FROM students
WHERE student_name LIKE '山%' AND student_name NOT LIKE '%子';
```

まとめ

この章では、パターンマッチングを行うためのLIKE演算子と、その中で使用するワイルドカード文字（%と_）について学びました：

1. **LIKE演算子**：文字列パターンに一致するデータを検索するための演算子

2. **%（パーセント）**：0文字以上の任意の文字列に一致するワイルドカード
3. _**（アンダースコア）**：任意の1文字に一致するワイルドカード
4. **前方一致**：「パターン%」で「パターンで始まる」文字列に一致
5. **後方一致**：「%パターン」で「パターンで終わる」文字列に一致
6. **部分一致**：「%パターン%」で「パターンを含む」文字列に一致
7. **NOT LIKE**：指定したパターンに一致しないデータを検索
8. **エスケープ文字**：特殊文字（%や_）を通常の文字として扱うための方法
9. **複合条件との組み合わせ**：AND、ORなどと組み合わせたより複雑な条件

パターンマッチングは、特にテキストデータを扱う際に非常に便利な機能です。部分的な情報しか持っていない場合や、特定のパターンを持つデータを探す場合に活用できます。

次の章では、範囲指定のための「BETWEEN演算子」と「IN演算子」について学びます。

---

5. 範囲指定：BETWEEN、IN演算子

はじめに

これまでの章では、等号（=）や不等号（>、<）を使って条件を指定する方法や、LIKE演算子を使ったパターンマッチングを学びました。この章では、値の範囲を指定する「BETWEEN演算子」と、複数の値を一度に指定できる「IN演算子」について学びます。

これらの演算子を使うと、次のような検索がより簡単になります：

- 「80点から90点の間の成績」
- 「2025年4月から6月の間のスケジュール」
- 「特定の教師IDリストに該当する講座」

BETWEEN演算子：範囲を指定する

BETWEEN演算子は、ある値が指定した範囲内にあるかどうかを調べるために使います。

> **用語解説**：
>
> - **BETWEEN**：「～の間に」という意味の演算子で、ある値が指定した最小値と最大値の間（両端の値を含む）にあるかどうかを判定します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 BETWEEN 最小値 AND 最大値;
```

この構文は次の条件と同じ意味です：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 >= 最小値 AND カラム名 <= 最大値;
```

例1：数値範囲の指定

例えば、成績（grades）テーブルから、80点から90点の間の成績を取得するには：

```
SELECT * FROM grades
WHERE score BETWEEN 80 AND 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
308	6	小テスト1	89.0	100.0	2025-05-15
…	…	…	…	…	…

例2：日付範囲の指定

日付にもBETWEEN演算子が使えます。例えば、2025年5月10日から2025年5月20日までに提出された成績を取得するには：

```
SELECT * FROM grades
WHERE submission_date BETWEEN '2025-05-10' AND '2025-05-20';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	レポート1	45.0	50.0	2025-05-10
302	1	レポート1	48.0	50.0	2025-05-10
301	1	中間テスト	85.5	100.0	2025-05-20
…	…	…	…	…	…

NOT BETWEEN：範囲外を指定する

NOT BETWEENを使うと、指定した範囲の外にある値を持つレコードを取得できます。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 NOT BETWEEN 最小値 AND 最大値;
```

これは次の条件と同じです：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 < 最小値 OR カラム名 > 最大値;
```

例：範囲外の値を取得

例えば、80点未満または90点より高い成績を取得するには：

```
SELECT * FROM grades
WHERE score NOT BETWEEN 80 AND 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
303	1	中間テスト	78.5	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
…	…	…	…	…	…

IN演算子：複数の値を指定する

IN演算子は、ある値が指定した複数の値のリストのいずれかに一致するかどうかを調べるために使います。

> **用語解説**：
>
> - **IN**：「～の中に含まれる」という意味の演算子で、ある値が指定したリストの中に含まれているかどうかを判定します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 IN (値1, 値2, 値3, ...);
```

この構文は次のOR条件の組み合わせと同じ意味です：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 = 値1 OR カラム名 = 値2 OR カラム名 = 値3 OR ...;
```

例1：数値リストの指定

例えば、教師ID（teacher_id）が101、103、105のいずれかである講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id IN (101, 103, 105);
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
3	Cプログラミング演習	101
5	データベース設計と実装	105
10	プロジェクト管理手法	103
…	…	…

例2：文字列リストの指定

文字列のリストにも適用できます。例えば、特定の教室ID（classroom_id）のみの教室情報を取得するには：

```
SELECT * FROM classrooms
WHERE classroom_id IN ('101A', '202D', '301E');
```

実行結果：

classroom_id	classroom_name	capacity	building	facilities
101A	1号館コンピュータ実習室A	30	1号館	パソコン30台、プロジェクター
202D	2号館コンピュータ実習室D	25	2号館	パソコン25台、プロジェクター、3Dプリンター
301E	3号館講義室E	80	3号館	プロジェクター、マイク設備、録画設備

NOT IN：リストに含まれない値を指定する

NOT IN演算子を使うと、指定したリストに含まれない値を持つレコードを取得できます。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 NOT IN (値1, 値2, 値3, ...);
```

これは次の条件と同じです：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 <> 値1 AND カラム名 <> 値2 AND カラム名 <> 値3 AND ...;
```

例：リストに含まれない値を取得

例えば、教師IDが101、102、103以外の教師が担当する講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id NOT IN (101, 102, 103);
```

実行結果：

course_id	course_name	teacher_id
4	Webアプリケーション開発	104
5	データベース設計と実装	105
6	ネットワークセキュリティ	107
…	…	…

IN演算子でのサブクエリの利用（基本）

IN演算子の括弧内には、直接値を書く代わりに、サブクエリ（別のSELECT文）を指定することもできます。これにより、動的に値のリストを生成できます。

> **用語解説**：
>
> - **サブクエリ**：SQL文の中に含まれる別のSQL文のことで、外側のSQL文（メインクエリ）に値や条件を提供します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 IN (SELECT カラム名 FROM 別テーブル WHERE 条件);
```

例：サブクエリを使ったIN条件

例えば、教師名（teacher_name）に「田」を含む教師が担当している講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id IN (
```

```
    SELECT teacher_id FROM teachers
    WHERE teacher_name LIKE '%田%'
);
```

実行結果：

course_id	course_name	teacher_id
2	UNIX入門	102
10	プロジェクト管理手法	103
…	…	…

この例では、まず「teachers」テーブルから名前に「田」を含む教師のIDを取得し、それらのIDを持つ講座を「courses」テーブルから取得しています。

BETWEEN演算子とIN演算子の組み合わせ

BETWEEN演算子とIN演算子は、論理演算子（AND、OR）と組み合わせて、さらに複雑な条件を作ることができます。

例：BETWEENとINの組み合わせ

例えば、「教師IDが101、103、105のいずれかで、かつ、2025年5月15日から2025年6月15日の間に実施される授業」を取得するには：

```
SELECT * FROM course_schedule
WHERE teacher_id IN (101, 103, 105)
  AND schedule_date BETWEEN '2025-05-15' AND '2025-06-15';
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
38	1	2025-05-20	1	102B	101	scheduled
42	10	2025-05-23	3	201C	103	scheduled
45	5	2025-05-28	2	402H	105	scheduled
…	…	…	…	…	…	…

練習問題

問題5-1

grades（成績）テーブルから、得点（score）が85点から95点の間にある成績を取得するSQLを書いてください。

問題5-2

course_schedule（授業カレンダー）テーブルから、2025年5月1日から2025年5月31日までの授業スケジュールを取得するSQLを書いてください。

問題5-3

courses（講座）テーブルから、教師ID（teacher_id）が104、106、108のいずれかである講座の情報を取得するSQLを書いてください。

問題5-4

classrooms（教室）テーブルから、教室ID（classroom_id）が「101A」、「201C」、「301E」、「401G」以外の教室情報を取得するSQLを書いてください。

問題5-5

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」または「実技試験」で、かつ得点（score）が80点から90点の間にない成績を取得するSQLを書いてください。

問題5-6

course_schedule（授業カレンダー）テーブルから、教室ID（classroom_id）が「101A」、「202D」のいずれかで、かつ2025年5月15日から2025年5月30日の間に実施される授業スケジュールを取得するSQLを書いてください。

解答

解答5-1

```
SELECT * FROM grades
WHERE score BETWEEN 85 AND 95;
```

解答5-2

```
SELECT * FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31';
```

解答5-3

```
SELECT * FROM courses
WHERE teacher_id IN (104, 106, 108);
```

解答5-4

```sql
SELECT * FROM classrooms
WHERE classroom_id NOT IN ('101A', '201C', '301E', '401G');
```

解答5-5

```sql
SELECT * FROM grades
WHERE grade_type IN ('中間テスト', '実技試験')
  AND score NOT BETWEEN 80 AND 90;
```

解答5-6

```sql
SELECT * FROM course_schedule
WHERE classroom_id IN ('101A', '202D')
  AND schedule_date BETWEEN '2025-05-15' AND '2025-05-30';
```

まとめ

この章では、範囲指定のための「BETWEEN演算子」と複数値指定のための「IN演算子」について学びました：

1. **BETWEEN演算子**：値が指定した範囲内（両端を含む）にあるかどうかをチェック
2. **NOT BETWEEN**：値が指定した範囲外にあるかどうかをチェック
3. **IN演算子**：値が指定したリストのいずれかに一致するかどうかをチェック
4. **NOT IN**：値が指定したリストのいずれにも一致しないかどうかをチェック
5. **サブクエリとIN**：動的に生成された値のリストを使用する方法
6. **複合条件**：BETWEEN、IN、論理演算子を組み合わせたより複雑な条件

これらの演算子を使うことで、複数の条件を指定する場合に、SQLをより簡潔に書くことができます。特に、多くの値を指定する場合や範囲条件を指定する場合に便利です。

次の章では、「NULL値の処理：IS NULL、IS NOT NULL」について学びます。

---

6. NULL値の処理：IS NULL、IS NOT NULL

はじめに

データベースの世界では、データがない状態を表すために「NULL」という特別な値が使われます。NULLは「空」や「0」や「空白文字」とは異なる、「値が存在しない」または「不明」であることを表す特殊な概念です。

例えば、学校データベースでは次のようなシナリオがあります：

- まだ成績が付けられていない（NULL）

- コメントが入力されていない（NULL）
- 授業がキャンセルされたため教室が割り当てられていない（NULL）

この章では、NULL値を正しく処理するための「IS NULL」と「IS NOT NULL」演算子について学びます。

NULLとは何か？

NULL値には、いくつかの特徴があります：

1. **値がない**：NULLは値がないことを表します。0でも空文字列（''）でもなく、値そのものが存在しないことを示します。
2. **不明**：データが不明であることを表す場合もあります。
3. **未設定**：まだ値が設定されていないことを表す場合もあります。
4. **比較できない**：NULLは通常の比較演算子（=, <, >など）で比較できません。

> **用語解説**：
>
> - **NULL**：データベースにおいて「値がない」または「不明」を表す特殊な値です。0や空文字とは異なります。

NULL値と通常の比較演算子

通常の比較演算子（=, <>, >, <, >=, <=）では、NULL値を正しく検出できません。例えば：

```
-- この条件はNULL値に対して常にFALSEを返す
SELECT * FROM テーブル名 WHERE カラム名 = NULL;

-- この条件もNULL値に対して常にFALSEを返す
SELECT * FROM テーブル名 WHERE カラム名 <> NULL;
```

これは、NULL値との等価比較は「不明」と評価されるためです。つまり、NULL = NULLでさえFALSEではなく「不明」になります。

IS NULL演算子

NULL値を持つレコードを検索するには、「IS NULL」演算子を使います。

> **用語解説**：
>
> - **IS NULL**：カラムの値がNULLかどうかを調べる演算子です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE カラム名 IS NULL;
```

例：IS NULLの使用

例えば、コメントが入力されていない（NULL）出席レコードを検索するには：

```
SELECT * FROM attendance WHERE comment IS NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	NULL
1	306	present	NULL
1	307	present	NULL
...	...	...	NULL

IS NOT NULL演算子

逆に、NULL値を持たないレコード（つまり、何らかの値を持つレコード）を検索するには、「IS NOT NULL」演算子を使います。

> **用語解説**：
>
> - **IS NOT NULL**：カラムの値がNULLでないかどうかを調べる演算子です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE カラム名 IS NOT NULL;
```

例：IS NOT NULLの使用

例えば、コメントが入力されている（NOT NULL）出席レコードを検索するには：

```
SELECT * FROM attendance WHERE comment IS NOT NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	302	late	15分遅刻
1	303	absent	事前連絡あり
1	308	late	5分遅刻
...	...	...	...

NULL値の論理的な扱い

NULL値は論理演算（AND、OR、NOT）でも特殊な扱いを受けます。

- **NULL AND TRUE** → NULL（不明）
- **NULL AND FALSE** → FALSE
- **NULL OR TRUE** → TRUE
- **NULL OR FALSE** → NULL（不明）
- **NOT NULL** → NULL（不明）

この特殊な振る舞いが、バグや誤った結果の原因になることがあります。

NULL値と結合条件

テーブル結合（JOINなど、後の章で学習）の際も、NULL値は特殊な扱いを受けます。NULL値同士は「等しい」とは判定されないため、通常の結合条件ではNULL値を持つレコードは結合されません。

IS NULLとIS NOT NULLを使った複合条件

IS NULLとIS NOT NULLも、他の条件と組み合わせて使用できます。

例：複合条件でのIS NULLの使用

例えば、「出席状態が "absent"（欠席）で、コメントがNULLでない（理由が入力されている）レコード」を検索するには：

```
SELECT * FROM attendance
WHERE status = 'absent' AND comment IS NOT NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	303	absent	事前連絡あり
1	317	absent	体調不良
…	…	…	…

NVL/IFNULL/COALESCE関数：NULL値の置換

NULL値を別の値に置き換えるための関数が用意されています。データベースによって関数名が異なることがありますが、機能は似ています：

- MySQL/MariaDB: **IFNULL(expr, replace_value)**
- Oracle: **NVL(expr, replace_value)**
- SQL Server: **ISNULL(expr, replace_value)**
- 標準SQL: **COALESCE(expr1, expr2, ..., exprN)** - 最初のNULLでない式を返します

例：IFNULL関数の使用（MySQL）

例えば、コメントがNULLの場合は「特記事項なし」と表示するには：

```
SELECT schedule_id, student_id, status,
       IFNULL(comment, '特記事項なし') AS comment
FROM attendance;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	特記事項なし
1	302	late	15分遅刻
1	303	absent	事前連絡あり
...	...	...	...

NULLを使う際の注意点

1. **除外の罠**：`WHERE カラム名 <> 値` だけでは、NULL値を持つレコードは含まれません。すべてのレコードを対象にするには：

   ```
   WHERE カラム名 <> 値 OR カラム名 IS NULL
   ```

2. **集計関数**：COUNT(*)はすべての行を数えますが、COUNT(カラム名)はそのカラムがNULLでない行だけを数えます。

3. **インデックス**：多くのデータベースでは、NULL値にもインデックスを適用できますが、データベースによって動作が異なる場合があります。

4. **一意性制約**：一般的に、UNIQUE制約ではNULL値は重複としてカウントされません（複数のNULL値が許可されます）。

練習問題

問題6-1

attendance（出席）テーブルから、コメント（comment）がNULLの出席レコードをすべて取得するSQLを書いてください。

問題6-2

course_schedule（授業カレンダー）テーブルから、状態（status）が「cancelled」で、かつ教室ID（classroom_id）がNULLでないレコードを取得するSQLを書いてください。

問題6-3

grades（成績）テーブルから、提出日（submission_date）がNULLの成績レコードを取得するSQLを書いてください。

問題6-4

attendance（出席）テーブルから、出席状態（status）が「present」か「late」で、かつコメント（comment）がNULLのレコードを取得するSQLを書いてください。

問題6-5

以下のSQLで教師（teachers）テーブルから「佐藤」という名前を持つ教師を検索する場合、NULL値を持つレコードも含めるにはどう修正すべきですか？

```
SELECT * FROM teachers WHERE teacher_name <> '佐藤花子';
```

問題6-6

attendance（出席）テーブルのすべてのレコードを取得し、コメント（comment）がNULLの場合は「記録なし」と表示するSQLを書いてください。

解答

解答6-1

```
SELECT * FROM attendance WHERE comment IS NULL;
```

解答6-2

```
SELECT * FROM course_schedule
WHERE status = 'cancelled' AND classroom_id IS NOT NULL;
```

解答6-3

```
SELECT * FROM grades WHERE submission_date IS NULL;
```

解答6-4

```
SELECT * FROM attendance
WHERE (status = 'present' OR status = 'late') AND comment IS NULL;
```

または

```

```

```
SELECT * FROM attendance
WHERE status IN ('present', 'late') AND comment IS NULL;
```

解答6-5

```
SELECT * FROM teachers
WHERE teacher_name <> '佐藤花子' OR teacher_name IS NULL;
```

解答6-6

```
SELECT schedule_id, student_id, status,
       IFNULL(comment, '記録なし') AS comment
FROM attendance;
```

まとめ

この章では、データベースにおけるNULL値の概念と、NULL値を扱うための演算子や関数について学びました：

1. **NULL値の概念**：値がない、不明、未設定を表す特殊な値
2. **IS NULL演算子**：NULL値を持つレコードを検索する方法
3. **IS NOT NULL演算子**：NULL値を持たないレコードを検索する方法
4. **NULL値の論理的扱い**：論理演算（AND、OR、NOT）におけるNULLの振る舞い
5. **複合条件**：IS NULL/IS NOT NULLと他の条件の組み合わせ
6. **NULL値の置換**：IFNULL/NVL/COALESCE関数の使用方法
7. **注意点**：NULL値を扱う際の一般的な落とし穴

NULL値の正確な理解と適切な処理は、SQLプログラミングの重要な部分です。不適切なNULL処理は、予期しない結果やバグの原因になります。

次の章では、クエリ結果の並び替えを行うための「ORDER BY：結果の並び替え」について学びます。

---

7. ORDER BY：結果の並び替え

はじめに

これまでの章では、データベースから条件に合ったレコードを取得する方法を学んできました。しかし実際の業務では、取得したデータを見やすく整理する必要があります。例えば：

- 成績を高い順に表示したい
- 学生を名前の五十音順に並べたい
- 日付の新しい順にスケジュールを確認したい

このようなデータの「並び替え」を行うためのSQLコマンドが「ORDER BY」です。この章では、クエリ結果を特定の順序で並べる方法を学びます。

ORDER BYの基本

ORDER BY句は、SELECT文の結果を指定したカラムの値に基づいて並び替えるために使います。

> **用語解説：**
>
> - **ORDER BY**：「～の順に並べる」という意味のSQLコマンドで、クエリ結果の並び順を指定します。

基本構文

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] ORDER BY 並び替えカラム;
```

ORDER BY句は通常、SELECT文の最後に記述します。

例1：単一カラムでの並び替え

例えば、学生（students）テーブルから、学生名（student_name）の五十音順（辞書順）でデータを取得するには：

```
SELECT * FROM students ORDER BY student_name;
```

実行結果：

student_id	student_name
309	相沢吉夫
303	柴崎春花
306	河田咲奈
305	河口菜恵子
…	…

デフォルトの並び順

ORDER BYを使わない場合、結果の順序は保証されません。多くの場合、データがデータベースに保存された順序で返されますが、これは信頼できるものではありません。

昇順と降順の指定

ORDER BY句では、並び順を「昇順」か「降順」のどちらかで指定できます。

> **用語解説：**

> - **昇順（ASC）**：小さい値から大きい値へ（A→Z、1→9）の順に並べます。
> - **降順（DESC）**：大きい値から小さい値へ（Z→A、9→1）の順に並べます。

構文

```
SELECT カラム名 FROM テーブル名 ORDER BY 並び替えカラム [ASC|DESC];
```

ASC（昇順）がデフォルトのため、省略可能です。

例2：降順での並び替え

例えば、成績（grades）テーブルから、得点（score）の高い順（降順）に成績を取得するには：

```
SELECT * FROM grades ORDER BY score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
…	…	…	…	…	…

複数カラムでの並び替え

複数のカラムを使って並び替えることもできます。最初に指定したカラムで並び替え、値が同じレコードがある場合は次のカラムで並び替えます。

構文

```
SELECT カラム名 FROM テーブル名
ORDER BY 並び替えカラム1 [ASC|DESC], 並び替えカラム2 [ASC|DESC], ...;
```

例3：複数カラムでの並び替え

例えば、成績（grades）テーブルから、課題タイプ（grade_type）の五十音順に並べ、同じ課題タイプ内では得点（score）の高い順に成績を取得するには：

```
SELECT * FROM grades
ORDER BY grade_type ASC, score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	2	実技試験	88.0	100.0	2025-05-18
321	2	実技試験	85.5	100.0	2025-05-18
…	…	…	…	…	…
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…
311	1	レポート1	49.0	50.0	2025-05-08
302	1	レポート1	48.0	50.0	2025-05-10
…	…	…	…	…	…

例4：昇順と降順の混合

各カラムごとに並び順を指定することもできます。例えば、講座ID（course_id）の昇順、評価タイプ（grade_type）の昇順、得点（score）の降順で並べるには：

```
SELECT * FROM grades
ORDER BY course_id ASC, grade_type ASC, score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	レポート1	49.0	50.0	2025-05-08
…	…	…	…	…	…
311	1	中間テスト	95.0	100.0	2025-05-20
…	…	…	…	…	…
301	2	実技試験	88.0	100.0	2025-05-18
…	…	…	…	…	…

NULLの扱い

ORDER BYでNULL値を並び替える場合、データベース製品によって動作が異なります。多くのデータベースでは、NULL値は最小値または最大値として扱われます。

- MySQL/MariaDBでは、NULL値は昇順（ASC）の場合は最小値として（最初に表示）、降順（DESC）の場合は最大値として（最後に表示）扱われます。

一部のデータベース（PostgreSQLなど）では、NULL値の位置を明示的に指定するための「NULLS FIRST」「NULLS LAST」構文がサポートされています。

例5：NULL値の扱い

例えば、出席（attendance）テーブルからコメント（comment）でソートすると、NULLが最初に来ます：

```
SELECT * FROM attendance ORDER BY comment;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	NULL
1	306	present	NULL
...	...	...	NULL
1	308	late	5分遅刻
1	323	late	電車遅延
...	...	...	...

カラム番号を使った並び替え

カラム名の代わりに、SELECT文の結果セットにおけるカラムの位置（番号）を使って並び替えることもできます。最初のカラムは1、2番目のカラムは2、という具合です。

構文

```
SELECT カラム名1, カラム名2, ... FROM テーブル名 ORDER BY カラム位置;
```

例6：カラム番号を使った並び替え

例えば、学生（students）テーブルから学生ID（student_id）と名前（student_name）を取得し、名前（2番目のカラム）で並べ替えるには：

```
SELECT student_id, student_name FROM students ORDER BY 2;
```

この場合、「ORDER BY 2」は「ORDER BY student_name」と同じ意味になります。

> **注意**：カラム番号を使う方法は、カラムの順序を変更すると問題が起きるため、実際の業務では使用を避けた方が良いとされています。

式や関数を使った並び替え

ORDER BY句で式や関数を使うことにより、計算結果に基づいて並び替えることもできます。

例7：式を使った並び替え

例えば、成績（grades）テーブルから、得点の達成率（score/max_score）の高い順に並べるには：

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score/max_score)*100 AS 達成率
FROM grades
ORDER BY (score/max_score) DESC;
```

または

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score/max_score)*100 AS 達成率
FROM grades
ORDER BY 達成率 DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	達成率
311	1	レポート1	49.0	50.0	98.0
320	1	レポート1	48.5	50.0	97.0
311	1	中間テスト	95.0	100.0	95.0
…	…	…	…	…	…

例8：関数を使った並び替え

文字列関数を使って並び替えることもできます。例えば、月名で並べることを考えましょう：

```
SELECT schedule_date, MONTH(schedule_date) AS month
FROM course_schedule
ORDER BY MONTH(schedule_date);
```

実行結果：

schedule_date	month
2025-04-07	4
2025-04-08	4
…	…

schedule_date	month
2025-05-01	5
2025-05-02	5
…	…
2025-06-01	6
…	…

CASE式を使った条件付き並び替え

さらに高度な並び替えとして、CASE式を使って条件に応じた並び順を定義することもできます。

例9：CASE式を使った並び替え

例えば、出席（attendance）テーブルから、出席状況（status）を「欠席→遅刻→出席」の順に優先して表示するには：

```
SELECT * FROM attendance
ORDER BY CASE
           WHEN status = 'absent' THEN 1
           WHEN status = 'late' THEN 2
           WHEN status = 'present' THEN 3
           ELSE 4
         END;
```

実行結果：

schedule_id	student_id	status	comment
1	303	absent	事前連絡あり
1	317	absent	体調不良
…	…	…	…
1	302	late	15分遅刻
1	308	late	5分遅刻
…	…	…	…
1	301	present	NULL
1	306	present	NULL
…	…	…	…

練習問題

問題7-1

students（学生）テーブルから、すべての学生情報を学生名（student_name）の降順（逆五十音順）で取得するSQLを書いてください。

問題7-2

grades（成績）テーブルから、得点（score）が85点以上の成績を得点の高い順に取得するSQLを書いてください。

問題7-3

course_schedule（授業カレンダー）テーブルから、2025年5月の授業スケジュールを日付（schedule_date）の昇順で取得するSQLを書いてください。

問題7-4

teachers（教師）テーブルから、教師IDと名前を取得し、名前（teacher_name）の五十音順で並べるSQLを書いてください。

問題7-5

grades（成績）テーブルから、講座ID（course_id）ごとに、成績を評価タイプ（grade_type）の五十音順に、同じ評価タイプ内では得点（score）の高い順に並べて取得するSQLを書いてください。

問題7-6

attendance（出席）テーブルから、すべての出席情報を出席状況（status）が「absent」「late」「present」の順番で、同じ状態内ではコメント（comment）の有無（NULLが後）で並べて取得するSQLを書いてください。

解答

解答7-1

```
SELECT * FROM students ORDER BY student_name DESC;
```

解答7-2

```
SELECT * FROM grades WHERE score >= 85 ORDER BY score DESC;
```

解答7-3

```
SELECT * FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31'
ORDER BY schedule_date;
```

解答7-4

```
SELECT teacher_id, teacher_name FROM teachers ORDER BY teacher_name;
```

解答7-5

```
SELECT * FROM grades
ORDER BY course_id, grade_type, score DESC;
```

解答7-6

```
SELECT * FROM attendance
ORDER BY
  CASE
    WHEN status = 'absent' THEN 1
    WHEN status = 'late' THEN 2
    WHEN status = 'present' THEN 3
    ELSE 4
  END,
  CASE
    WHEN comment IS NULL THEN 2
    ELSE 1
  END;
```

まとめ

この章では、クエリ結果を特定の順序で並べるための「ORDER BY」句について学びました：

1. **基本的な並び替え**：指定したカラムの値に基づいて結果を並べる方法
2. **昇順と降順**：ASC（昇順）とDESC（降順）の指定方法
3. **複数カラムでの並び替え**：優先順位の高いカラムから順に指定する方法
4. **NULL値の扱い**：NULL値が並び替えでどのように扱われるか
5. **カラム番号**：カラム名の代わりに位置で指定する方法（あまり推奨されない）
6. **式や関数**：計算結果に基づいて並べる方法
7. **CASE式**：条件付きの複雑な並び替え

ORDER BY句は、データを見やすく整理するために非常に重要です。特に大量のデータを扱う場合、適切な並び順はデータの理解を大きく助けます。

次の章では、取得する結果の件数を制限する「LIMIT句：結果件数の制限とページネーション」について学びます。

---

8. LIMIT句：結果件数の制限とページネーション

はじめに

これまでの章では、条件に合うデータを取得し、それを特定の順序で並べる方法を学びました。しかし実際のアプリケーションでは、大量のデータがあるときに、その一部だけを表示したいことがよくあります。例えば：

- 成績上位10件だけを表示したい
- Webページで一度に20件ずつ表示したい（ページネーション）
- 最新の5件のお知らせだけを取得したい

このような「結果の件数を制限する」ためのSQLコマンドが「LIMIT句」です。この章では、クエリ結果の件数を制限する方法と、ページネーションの実装方法を学びます。

LIMIT句の基本

LIMIT句は、SELECT文の結果から指定した件数だけを取得するために使います。

> **用語解説**：
>
> - **LIMIT**：「制限する」という意味のSQLコマンドで、取得する行数を制限します。

基本構文（MySQL/MariaDB）

MySQLやMariaDBでのLIMIT句の基本構文は次のとおりです：

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT 件数;
```

LIMIT句は通常、SELECT文の最後に記述します（ORDER BYの後）。

例1：単純なLIMIT

例えば、学生（students）テーブルから最初の5人だけを取得するには：

```
SELECT * FROM students LIMIT 5;
```

実行結果：

student_id	student_name
301	黒沢春馬
302	新垣愛留
303	柴崎春花
304	森下風凛
305	河口菜恵子

ORDER BYとLIMITの組み合わせ

通常、LIMIT句はORDER BY句と組み合わせて使用します。これにより、「上位N件」「最新N件」などの操作が可能になります。

例2：ORDER BYとLIMITの組み合わせ

例えば、成績（grades）テーブルから得点（score）の高い順に上位3件を取得するには：

```
SELECT * FROM grades ORDER BY score DESC LIMIT 3;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20

例3：最新のレコードを取得

日付でソートして最新のデータを取得することもよくあります。例えば、最新の3つの授業スケジュールを取得するには：

```
SELECT * FROM course_schedule
ORDER BY schedule_date DESC LIMIT 3;
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
95	28	2026-12-21	3	202D	119	scheduled
94	1	2026-12-21	1	102B	101	scheduled
93	14	2026-12-15	4	202D	110	scheduled

OFFSETとページネーション

Webアプリケーションなどでは、大量のデータを「ページ」に分けて表示することがよくあります（ページネーション）。この機能を実現するためには、「OFFSET」（オフセット）という機能が必要です。

> **用語解説**：
>
> - **OFFSET**：「ずらす」という意味で、結果セットの先頭から指定した数だけ行をスキップします。
> - **ページネーション**：大量のデータを複数のページに分割して表示する技術です。

基本構文（MySQL/MariaDB）

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT 件数 OFFSET スキップ数;
```

または、短縮形として：

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT スキップ数, 件数;
```

例4：OFFSETを使ったスキップ

例えば、学生（students）テーブルから6番目から10番目までの学生を取得するには：

```
SELECT * FROM students LIMIT 5 OFFSET 5;
```

または：

```
SELECT * FROM students LIMIT 5, 5;
```

実行結果：

student_id	student_name
306	河田咲奈
307	織田柚夏
308	永田悦子
309	相沢吉夫
310	吉川伽羅

例5：ページネーションの実装

ページネーションを実装する場合、通常は以下の式を使ってOFFSETを計算します：

```
OFFSET = (ページ番号 - 1) × ページあたりの件数
```

例えば、1ページあたり10件表示で、3ページ目のデータを取得するには：

```
SELECT * FROM students ORDER BY student_id LIMIT 10 OFFSET 20;
```

または：

```
SELECT * FROM students ORDER BY student_id LIMIT 20, 10;
```

実行結果：

student_id	student_name
321	井上竜也
322	木村結衣
323	林正義
324	清水香織
325	山田翔太
326	葉山陽太
327	青山凛
328	沢村大和
329	白石優月
330	月岡星奈

LIMIT句を使用する際の注意点

1. ORDER BYの重要性

LIMIT句を使用する場合、通常はORDER BY句も一緒に使うべきです。ORDER BYがなければ、どのレコードが取得されるかは保証されません。

```
-- 良い例：結果が予測可能
SELECT * FROM students ORDER BY student_id LIMIT 5;

-- 悪い例：結果が不確定
SELECT * FROM students LIMIT 5;
```

2. パフォーマンスへの影響

大規模なテーブルで大きなOFFSET値を使用すると、パフォーマンスが低下する可能性があります。これは、データベースがOFFSET分のレコードを読み込んでから破棄する必要があるためです。

3. データベース製品による構文の違い

LIMIT句の構文はデータベース製品によって異なります：

- **MySQL/MariaDB/SQLite**：`LIMIT 件数 OFFSET スキップ数` または `LIMIT スキップ数, 件数`
- **PostgreSQL**：`LIMIT 件数 OFFSET スキップ数`
- **Oracle**：`OFFSET スキップ数 ROWS FETCH NEXT 件数 ROWS ONLY`
- **SQL Server**：`OFFSET スキップ数 ROWS FETCH NEXT 件数 ROWS ONLY` または旧バージョンでは `TOP`句

この章では主にMySQL/MariaDBの構文を使用します。

実践的なページネーションの実装

実際のアプリケーションでページネーションを実装する場合、以下のようなコードになります（疑似コード）：

```
ページ番号 = URLから取得またはデフォルト値（例：1）
1ページあたりの件数 = 設定値（例：10）
総レコード数 = SELECTで取得（COUNT(*)を使用）
総ページ数 = CEILING(総レコード数 ÷ 1ページあたりの件数)
OFFSET = (ページ番号 - 1) × 1ページあたりの件数

SQLクエリ = "SELECT * FROM テーブル ORDER BY カラム LIMIT " + 1ページあたりの件数 + " OFFSET
" + OFFSET
```

例6：総レコード数と総ページ数の取得

総レコード数を取得するには：

```sql
SELECT COUNT(*) AS total_records FROM students;
```

実行結果：

total_records
100

この場合、1ページあたり10件表示なら、総ページ数は10ページ（CEILING(100 ÷ 10)）になります。

練習問題

問題8-1

grades（成績）テーブルから、得点（score）の高い順に上位5件の成績レコードを取得するSQLを書いてください。

問題8-2

course_schedule（授業カレンダー）テーブルから、日付（schedule_date）の新しい順に3件のスケジュールを取得するSQLを書いてください。

問題8-3

students（学生）テーブルを学生ID（student_id）の昇順で並べ、11番目から15番目までの学生（5件）を取得するSQLを書いてください。

問題8-4

teachers（教師）テーブルから、教師名（teacher_name）の五十音順で6番目から10番目までの教師情報を取得するSQLを書いてください。

問題8-5

1ページあたり20件表示で、grades（成績）テーブルの3ページ目のデータを得点（score）の高い順に取得するSQLを書いてください。

問題8-6

course_schedule（授業カレンダー）テーブルから、状態（status）が「scheduled」のスケジュールを日付（schedule_date）の昇順で並べ、先頭から10件スキップして次の5件を取得するSQLを書いてください。

解答

解答8-1

```
SELECT * FROM grades ORDER BY score DESC LIMIT 5;
```

解答8-2

```
SELECT * FROM course_schedule ORDER BY schedule_date DESC LIMIT 3;
```

解答8-3

```
SELECT * FROM students ORDER BY student_id LIMIT 5 OFFSET 10;
```

または

```
SELECT * FROM students ORDER BY student_id LIMIT 10, 5;
```

解答8-4

```
SELECT * FROM teachers ORDER BY teacher_name LIMIT 5 OFFSET 5;
```

または

```
SELECT * FROM teachers ORDER BY teacher_name LIMIT 5, 5;
```

解答8-5

```
SELECT * FROM grades ORDER BY score DESC LIMIT 20 OFFSET 40;
```

または

```
SELECT * FROM grades ORDER BY score DESC LIMIT 40, 20;
```

解答8-6

```
SELECT * FROM course_schedule
WHERE status = 'scheduled'
ORDER BY schedule_date
LIMIT 5 OFFSET 10;
```

または

```
SELECT * FROM course_schedule
WHERE status = 'scheduled'
ORDER BY schedule_date
LIMIT 10, 5;
```

まとめ

この章では、クエリ結果の件数を制限するためのLIMIT句と、ページネーションの実装方法について学びました：

1. **LIMIT句の基本**：指定した件数だけのレコードを取得する方法
2. **ORDER BYとの組み合わせ**：順序付けされた結果から一部だけを取得する方法
3. **OFFSET**：結果の先頭から指定した数だけレコードをスキップする方法
4. **ページネーション**：大量のデータを複数のページに分けて表示する実装方法
5. **注意点**：LIMIT句を使用する際の留意事項
6. **データベース製品による違い**：異なるデータベースでの構文の違い

LIMIT句は特にWebアプリケーションの開発で重要な機能です。大量のデータを効率よく表示するためのページネーション機能を実装するために欠かせません。また、トップN（上位N件）やボトムN（下位N件）のデータを取得する際にも使われます。

次の章では、データの集計分析を行うための「集計関数：COUNT、SUM、AVG、MAX、MIN」について学びます。

SQL学習テキスト 完全版

このテキストは、SQLの基礎から応用まで体系的に学習できるように構成されています。 学校データベースを使った実践的な例題と練習問題を通じて、実務で使えるSQLスキルを身につけることができます。

目次

---

9. 集計関数：COUNT、SUM、AVG、MAX、MIN

はじめに

これまでの章では、データベースからレコードを取得して表示する方法を学んできました。しかし、データベースを使う目的の一つは、大量のデータから有用な情報（集計、統計など）を抽出することです。例えば：

- 「学生の総数は何人か？」
- 「成績の平均点は？」
- 「最高点と最低点は？」
- 「全講座の合計受講者数は？」

このような「データの集計」を行うためのSQLの機能が「集計関数」です。この章では、最もよく使われる5つの集計関数（COUNT、SUM、AVG、MAX、MIN）について学びます。

集計関数の基本

集計関数は、複数の行のデータをまとめて一つの値を返す関数です。

> **用語解説**：
>
> - **集計関数**：複数の行（レコード）から計算された単一の値を返す関数です。データの集計や統計に使用されます。

主な集計関数

関数	説明	例
COUNT	レコード数を数える	学生の総数

関数	説明	例
SUM	値の合計を計算する	全成績の点数合計
AVG	値の平均を計算する	平均点
MAX	最大値を取得する	最高点
MIN	最小値を取得する	最低点

COUNT関数：レコード数をカウントする

COUNT関数は、条件に一致するレコードの数を数えるのに使用します。

基本構文

```
SELECT COUNT(カラム名) FROM テーブル名 [WHERE 条件];
```

または、すべての行を数える場合：

```
SELECT COUNT(*) FROM テーブル名 [WHERE 条件];
```

例1：テーブル内の全レコード数

例えば、学生（students）テーブルの全学生数を数えるには：

```
SELECT COUNT(*) AS 学生総数 FROM students;
```

実行結果：

学生総数
100

例2：条件付きのカウント

WHERE句と組み合わせることで、条件に一致するレコードだけをカウントできます。例えば、「出席（present）」状態の出席レコードの数を数えるには：

```
SELECT COUNT(*) AS 出席数 FROM attendance WHERE status = 'present';
```

実行結果：

出席数

出席数
42

例3：NULL以外の値をカウント

COUNT(カラム名)を使うと、そのカラムがNULLでない行だけがカウントされます。これは、COUNT(*)との大きな違いです。例えば、コメント（comment）が入力されている出席レコードの数を数えるには：

```
SELECT COUNT(comment) AS コメントあり FROM attendance;
```

実行結果：

コメントあり
23

例4：DISTINCTを使った重複のないカウント

DISTINCTを使うと、重複を除外してカウントできます。例えば、何種類の評価タイプ（grade_type）があるかを数えるには：

```
SELECT COUNT(DISTINCT grade_type) AS 評価タイプ数 FROM grades;
```

実行結果：

評価タイプ数
3

SUM関数：合計を計算する

SUM関数は、数値カラムの合計を計算するのに使用します。

基本構文

```
SELECT SUM(数値カラム) FROM テーブル名 [WHERE 条件];
```

例1：単純な合計

例えば、全成績レコードの得点（score）の合計を計算するには：

```
SELECT SUM(score) AS 総得点 FROM grades;
```

実行結果：

総得点
3542.5

例2：条件付きの合計

WHERE句と組み合わせることで、条件に一致するレコードだけの合計を計算できます。例えば、「中間テスト」の得点合計を計算するには：

```
SELECT SUM(score) AS 中間テスト合計点
FROM grades
WHERE grade_type = '中間テスト';
```

実行結果：

中間テスト合計点
1728.0

例3：計算式を使った合計

計算式と組み合わせることもできます。例えば、全成績の達成率（score/max_score）の合計を計算するには：

```
SELECT SUM(score/max_score) AS 達成率合計 FROM grades;
```

実行結果：

達成率合計
39.86

AVG関数：平均を計算する

AVG関数は、数値カラムの平均値を計算するのに使用します。

基本構文

```
SELECT AVG(数値カラム) FROM テーブル名 [WHERE 条件];
```

例1：単純な平均

例えば、全成績レコードの得点（score）の平均を計算するには：

```
SELECT AVG(score) AS 平均点 FROM grades;
```

実行結果：

平均点
82.38

例2：条件付きの平均

WHERE句と組み合わせることで、条件に一致するレコードだけの平均を計算できます。例えば、「実技試験」の平均点を計算するには：

```
SELECT AVG(score) AS 実技試験平均点
FROM grades
WHERE grade_type = '実技試験';
```

実行結果：

実技試験平均点
85.6

例3：達成率（%）の計算

計算式と組み合わせることで、例えば平均達成率（%）を計算できます：

```
SELECT AVG(score/max_score * 100) AS 平均達成率 FROM grades;
```

実行結果：

平均達成率
87.24

MAX関数：最大値を取得する

MAX関数は、数値、文字列、日付などのカラムから最大値を取得するのに使用します。

基本構文

```
SELECT MAX(カラム名) FROM テーブル名 [WHERE 条件];
```

例1：数値の最大値

例えば、全成績レコードの中での最高点を取得するには：

```
SELECT MAX(score) AS 最高点 FROM grades;
```

実行結果：

最高点
95.0

例2：文字列の最大値（辞書順で最後）

MAX関数は文字列にも使用でき、この場合は辞書順で「最後」の値を返します。例えば、学生名の辞書順で最後の値（最も「わ行」に近い名前）を取得するには：

```
SELECT MAX(student_name) AS 学生名最終 FROM students;
```

実行結果：

学生名最終
吉川伽羅

例3：日付の最大値（最新の日付）

日付にMAX関数を使うと、最新（最も未来）の日付が取得できます。例えば、最も新しい提出日を取得するには：

```
SELECT MAX(submission_date) AS 最新提出日 FROM grades;
```

実行結果：

最新提出日
2025-05-20

MIN関数：最小値を取得する

MIN関数は、数値、文字列、日付などのカラムから最小値を取得するのに使用します。

基本構文

```
SELECT MIN(カラム名) FROM テーブル名 [WHERE 条件];
```

例1：数値の最小値

例えば、全成績レコードの中での最低点を取得するには：

```
SELECT MIN(score) AS 最低点 FROM grades;
```

実行結果：

最低点
68.0

例2：文字列の最小値（辞書順で最初）

MIN関数は文字列にも使用でき、この場合は辞書順で「最初」の値を返します。例えば、学生名の辞書順で最初の値（最も「あ行」に近い名前）を取得するには：

```
SELECT MIN(student_name) AS 学生名最初 FROM students;
```

実行結果：

学生名最初
相沢吉夫

例3：日付の最小値（最古の日付）

日付にMIN関数を使うと、最も古い日付が取得できます。例えば、最も古い提出日を取得するには：

```
SELECT MIN(submission_date) AS 最古提出日 FROM grades;
```

実行結果：

最古提出日
2025-05-06

複数の集計関数の組み合わせ

複数の集計関数を一つのクエリで使用することもできます。

例：成績の統計情報

例えば、成績（grades）テーブルの統計情報を一度に取得するには：

```sql
SELECT
    COUNT(*) AS レコード数,
    AVG(score) AS 平均点,
    SUM(score) AS 合計点,
    MAX(score) AS 最高点,
    MIN(score) AS 最低点
FROM grades;
```

実行結果：

レコード数	平均点	合計点	最高点	最低点
43	82.38	3542.5	95.0	68.0

集計関数とGROUP BY（次章の内容）

集計関数をより強力に使うためには、「GROUP BY」句と組み合わせます。これにより、データをグループ化して各グループごとに集計できます。GROUP BYについては次章で詳しく学びます。

例えば、講座（course_id）ごとの平均点を計算するには：

```sql
SELECT course_id, AVG(score) AS 平均点
FROM grades
GROUP BY course_id;
```

実行結果：

course_id	平均点
1	86.2
2	83.8
3	80.5
…	…

この例の詳細な説明は次章で行います。

練習問題

問題9-1

students（学生）テーブルから、学生の総数を取得するSQLを書いてください。

問題9-2

grades（成績）テーブルから、全成績の平均点（score）を取得するSQLを書いてください。

問題9-3

attendance（出席）テーブルから、出席状況（status）が「absent」のレコード数を取得するSQLを書いてください。

問題9-4

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」の成績の最高点と最低点を取得するSQLを書いてください。

問題9-5

course_schedule（授業カレンダー）テーブルから、最新（schedule_dateが最大）の授業スケジュールの日付を取得するSQLを書いてください。

問題9-6

grades（成績）テーブルから、成績の総数、平均点、合計点、最高点、最低点を一度に取得するSQLを書いてください。

解答

解答9-1

```
SELECT COUNT(*) AS 学生総数 FROM students;
```

解答9-2

```
SELECT AVG(score) AS 全成績平均点 FROM grades;
```

解答9-3

```
SELECT COUNT(*) AS 欠席数 FROM attendance WHERE status = 'absent';
```

解答9-4

```
SELECT
    MAX(score) AS 中間テスト最高点,
    MIN(score) AS 中間テスト最低点
FROM grades
WHERE grade_type = '中間テスト';
```

解答9-5

```
SELECT MAX(schedule_date) AS 最新授業日 FROM course_schedule;
```

解答9-6

```
SELECT
    COUNT(*) AS 成績総数,
    AVG(score) AS 平均点,
    SUM(score) AS 合計点,
    MAX(score) AS 最高点,
    MIN(score) AS 最低点
FROM grades;
```

まとめ

この章では、データの集計と分析に使用される主要な集計関数について学びました：

1. **COUNT**：レコード数をカウントする関数
2. **SUM**：数値の合計を計算する関数
3. **AVG**：数値の平均を計算する関数
4. **MAX**：最大値（数値、文字列、日付など）を取得する関数
5. **MIN**：最小値（数値、文字列、日付など）を取得する関数

これらの集計関数を使うことで、大量のデータから意味のある統計情報を簡単に抽出できます。ビジネスにおける意思決定や、データ分析において非常に重要な機能です。

次の章では、データをグループ化して集計を行うための「GROUP BY：データのグループ化」について学びます。

---

10. GROUP BY：データのグループ化

はじめに

前章では、集計関数（COUNT、SUM、AVG、MAX、MIN）を使って全体的な集計や統計を計算する方法を学びました。しかし実際のデータ分析では、全体の集計だけでなく、特定のカテゴリや条件ごとに集計したい場合が多くあります。例えば：

- 「講座ごとの平均点は？」
- 「教師ごとの担当講座数は？」
- 「日付ごとの授業数は？」
- 「出席状況（出席/遅刻/欠席）の割合は？」

このような「グループごとの集計」を行うためのSQLコマンドが「GROUP BY」です。この章では、データをグループ化して集計する方法について学びます。

GROUP BYの基本

GROUP BY句は、指定したカラムの値が同じレコードをグループ化し、それぞれのグループに対して集計関数を適用するために使います。

> **用語解説**：
>
> - **GROUP BY**：「～でグループ化する」という意味のSQLコマンドで、同じ値を持つレコードをまとめてグループにします。
> - **グループ化**：データを特定の条件で分類し、それぞれの分類ごとに集計を行うこと。

基本構文

```
SELECT カラム名, 集計関数
FROM テーブル名
[WHERE 条件]
GROUP BY グループ化するカラム名;
```

重要なのは、SELECT句に含めるカラムは、GROUP BY句で指定したカラムか、集計関数（COUNT、SUM、AVG、MAX、MINなど）のいずれかでなければならないということです。

例1：単純なグループ化

例えば、成績（grades）テーブルから、評価タイプ（grade_type）ごとの成績レコード数を集計するには：

```
SELECT grade_type, COUNT(*) AS レコード数
FROM grades
GROUP BY grade_type;
```

実行結果：

grade_type	レコード数
中間テスト	12
レポート1	22
実技試験	9

例2：グループごとの平均

グループごとの平均を計算することも一般的です。例えば、各評価タイプごとの平均点を計算するには：

```
SELECT grade_type, AVG(score) AS 平均点
FROM grades
GROUP BY grade_type;
```

実行結果：

grade_type	平均点
中間テスト	86.25
レポート1	44.77
実技試験	85.61

複数のカラムによるグループ化

複数のカラムを指定してグループ化することもできます。この場合、指定したすべてのカラムの値の組み合わせがグループの単位になります。

例3：複数カラムでのグループ化

例えば、講座（course_id）と評価タイプ（grade_type）の組み合わせごとに成績の平均を計算するには：

```
SELECT course_id, grade_type, AVG(score) AS 平均点
FROM grades
GROUP BY course_id, grade_type;
```

実行結果：

course_id	grade_type	平均点
1	中間テスト	87.33
1	レポート1	45.39
2	実技試験	86.83
…	…	…

グループ化の順序

GROUP BY句でカラムを指定する順序は、結果には影響しません。しかし、可読性のためにSELECT句と同じ順序で指定することが一般的です。

WHERE句とGROUP BYの組み合わせ

WHERE句は、グループ化する前に行単位でレコードを絞り込むのに使います。

> **用語解説**：
>
> - **絞り込み順序**：WHERE句はGROUP BY句の前に評価され、条件に合うレコードだけがグループ化の対象になります。

例4：WHEREとGROUP BYの組み合わせ

例えば、得点（score）が80以上の成績だけを対象に、評価タイプごとの平均を計算するには：

```
SELECT grade_type, AVG(score) AS 平均点
FROM grades
WHERE score >= 80
GROUP BY grade_type;
```

実行結果：

grade_type	平均点
中間テスト	88.92
実技試験	87.25

この例では、score >= 80の条件に一致するレコードだけがグループ化され、集計されています。レポート1はすべての点数が80未満なので、結果には表示されていません。

GROUP BYとORDER BYの組み合わせ

GROUP BY句とORDER BY句を組み合わせることで、グループ化した結果を任意の順序で並べることができます。

例5：GROUP BYとORDER BYの組み合わせ

例えば、講座（course_id）ごとの平均点を計算し、平均点の高い順に並べるには：

```
SELECT course_id, AVG(score) AS 平均点
FROM grades
GROUP BY course_id
ORDER BY 平均点 DESC;
```

実行結果：

course_id	平均点
1	86.21
2	83.79
5	82.33
...	...

集計関数の複数使用

一つのクエリで複数の集計関数を使用することもできます。

例6：複数の集計関数の使用

例えば、評価タイプごとの成績数、平均点、最高点、最低点を一度に取得するには：

```
SELECT grade_type,
       COUNT(*) AS 成績数,
       AVG(score) AS 平均点,
       MAX(score) AS 最高点,
       MIN(score) AS 最低点
FROM grades
GROUP BY grade_type;
```

実行結果：

grade_type	成績数	平均点	最高点	最低点
中間テスト	12	86.25	95.0	78.5
レポート1	22	44.77	49.0	37.0
実技試験	9	85.61	88.0	79.5

計算式を含むグループ化

GROUP BY句で集計した結果に対して、さらに計算を加えることができます。

例7：計算式を含むグループ化

例えば、評価タイプごとに平均達成率（score/max_score）を計算するには：

```
SELECT grade_type,
       AVG(score/max_score * 100) AS 平均達成率
FROM grades
GROUP BY grade_type;
```

実行結果：

grade_type	平均達成率
中間テスト	86.25
レポート1	89.54
実技試験	85.61

GROUP BYとNULLの扱い

GROUP BY句でグループ化する際、NULL値も一つのグループとして扱われます。

例8：NULLを含むグループ化

例えば、出席（attendance）テーブルから、コメント（comment）の有無でグループ化し、それぞれの件数を数えるには：

```
SELECT
    CASE
        WHEN comment IS NULL THEN 'コメントなし'
        ELSE 'コメントあり'
    END AS コメント状態,
    COUNT(*) AS 件数
FROM attendance
GROUP BY
    CASE
        WHEN comment IS NULL THEN 'コメントなし'
        ELSE 'コメントあり'
    END;
```

実行結果：

コメント状態	件数
コメントなし	20
コメントあり	23

HAVINGを使ったグループの絞り込み（次章の内容）

グループ化した後にさらに条件でグループを絞り込むには、「HAVING」句を使います。これについては次章で詳しく学びます。

例えば、5人以上の学生が受講している講座だけを取得するには：

```
SELECT course_id, COUNT(student_id) AS 学生数
FROM student_courses
GROUP BY course_id
HAVING COUNT(student_id) >= 5;
```

この例の詳細な説明は次章で行います。

練習問題

問題10-1

courses（講座）テーブルから、教師ID（teacher_id）ごとに担当している講座の数を取得するSQLを書いてください。

問題10-2

attendance（出席）テーブルから、出席状況（status）ごとのレコード数を取得するSQLを書いてください。

問題10-3

grades（成績）テーブルから、学生ID（student_id）ごとの平均点を計算し、平均点の高い順に並べるSQLを書いてください。

問題10-4

course_schedule（授業カレンダー）テーブルから、教室ID（classroom_id）ごとに予定されている授業の数を取得するSQLを書いてください。

問題10-5

grades（成績）テーブルから、講座ID（course_id）と評価タイプ（grade_type）の組み合わせごとの平均点を計算し、講座ID順、さらに評価タイプ順で並べるSQLを書いてください。

問題10-6

student_courses（受講）テーブルから、講座ID（course_id）ごとの受講者数を取得し、受講者数の多い順に並べるSQLを書いてください。

解答

解答10-1

```
SELECT teacher_id, COUNT(course_id) AS 担当講座数
FROM courses
GROUP BY teacher_id;
```

解答10-2

```
SELECT status, COUNT(*) AS レコード数
FROM attendance
GROUP BY status;
```

解答10-3

```
SELECT student_id, AVG(score) AS 平均点
FROM grades
GROUP BY student_id
ORDER BY 平均点 DESC;
```

解答10-4

```
SELECT classroom_id, COUNT(*) AS 授業数
FROM course_schedule
GROUP BY classroom_id;
```

解答10-5

```
SELECT course_id, grade_type, AVG(score) AS 平均点
FROM grades
GROUP BY course_id, grade_type
ORDER BY course_id, grade_type;
```

解答10-6

```
SELECT course_id, COUNT(student_id) AS 受講者数
FROM student_courses
GROUP BY course_id
ORDER BY 受講者数 DESC;
```

まとめ

この章では、データをグループ化して集計するためのGROUP BY句について学びました：

1. **GROUP BYの基本**：同じ値を持つレコードをグループ化する方法
2. **集計関数との組み合わせ**：グループごとの集計を行う方法
3. **複数カラムによるグループ化**：複数の条件でのグループ化
4. **WHERE句との組み合わせ**：グループ化前のレコード絞り込み
5. **ORDER BYとの組み合わせ**：グループ化結果の並び替え
6. **複数の集計関数の使用**：一度に複数の統計値を取得する方法
7. **計算式を含むグループ化**：より複雑な集計の実現
8. **NULLの扱い**：グループ化におけるNULL値の取り扱い

GROUP BY句を使うことで、データのさまざまな側面から分析が可能になり、意思決定のための有用な情報を抽出できるようになります。

次の章では、グループ化した結果をさらに条件で絞り込むための「HAVING：グループ化後の絞り込み」について学びます。

---

11. HAVING：グループ化後の絞り込み

はじめに

前章では、GROUP BY句を使ってデータをグループ化し、各グループごとに集計を行う方法を学びました。しかし、すべてのグループの集計結果を表示するのではなく、特定の条件を満たすグループだけを表示したい場合があります。例えば：

- 「平均点が80点以上の講座だけを表示したい」
- 「5人以上の学生が登録している講座だけを取得したい」

- 「出席率が90%を超える学生だけをリストアップしたい」

このような「グループ化した結果をさらに絞り込む」ためのSQLコマンドが「HAVING」句です。この章では、グループ化した結果に条件を適用する方法について学びます。

HAVINGの基本

HAVING句は、GROUP BY句でグループ化した後の結果に対して条件を適用し、条件を満たすグループだけを取得するために使います。

> **用語解説**：
>
> - **HAVING**：「～を持っている」という意味のSQLコマンドで、グループ化した結果に対して条件を適用します。

基本構文

```
SELECT カラム名, 集計関数
FROM テーブル名
[WHERE 行レベルの条件]
GROUP BY グループ化するカラム名
HAVING グループレベルの条件;
```

例1：基本的なHAVINGの使用

例えば、受講者数が5人以上の講座だけを取得するには：

```
SELECT course_id, COUNT(student_id) AS 受講者数
FROM student_courses
GROUP BY course_id
HAVING COUNT(student_id) >= 5;
```

実行結果：

course_id	受講者数
1	12
2	8
4	7
5	7
…	…

この例では、まず講座ID（course_id）ごとに学生ID（student_id）の数を数え、その後でHAVING句を使って受講者数が5人以上のグループだけを抽出しています。

WHEREとHAVINGの違い

WHERE句とHAVING句は似ていますが、適用されるタイミングと対象が異なります：

- **WHERE**：グループ化される前の個々の行（レコード）に対して条件を適用します。
- **HAVING**：グループ化された後のグループに対して条件を適用します。

> **用語解説**：
>
> - **行レベルのフィルタリング**：WHEREによる個々のレコードの絞り込み
> - **グループレベルのフィルタリング**：HAVINGによるグループの絞り込み

例2：WHEREとHAVINGの違い

以下の例で違いを確認しましょう：

```
-- WHERE（行レベル）：まず80点以上の成績だけを選び、それからグループ化
SELECT course_id, AVG(score) AS 平均点
FROM grades
WHERE score >= 80
GROUP BY course_id;

-- HAVING（グループレベル）：まずグループ化して平均点を計算し、それから平均点が80以上のグループを選ぶ
SELECT course_id, AVG(score) AS 平均点
FROM grades
GROUP BY course_id
HAVING AVG(score) >= 80;
```

1つ目のクエリは、「80点以上の成績だけ」を対象に講座ごとの平均点を計算します。 2つ目のクエリは、「講座の平均点が80点以上」の講座だけを取得します。

WHERE句の結果：

course_id	平均点
1	88.92
2	87.25
…	…

HAVING句の結果：

course_id	平均点
1	86.21
2	83.79
5	82.33
…	…

HAVINGで使用できる条件

HAVING句では、集計関数（COUNT、SUM、AVG、MAX、MINなど）を使った条件や、GROUP BY句で指定したカラムに対する条件を指定できます。

例3：集計関数を使った条件

例えば、平均点が85点以上の講座を取得するには：

```
SELECT course_id, AVG(score) AS 平均点
FROM grades
GROUP BY course_id
HAVING AVG(score) >= 85;
```

実行結果：

course_id	平均点
1	86.21
…	…

例4：複数条件の指定

HAVING句も、WHERE句と同様に複数の条件を組み合わせることができます：

```
SELECT grade_type, COUNT(*) AS 件数, AVG(score) AS 平均点
FROM grades
GROUP BY grade_type
HAVING COUNT(*) > 10 AND AVG(score) > 80;
```

実行結果：

grade_type	件数	平均点
中間テスト	12	86.25
…	…	…

WHEREとHAVINGの組み合わせ

実際のクエリでは、WHERE句とHAVING句を組み合わせることが多いです。WHERE句でグループ化前の行を絞り込み、HAVING句でグループ化後の結果をさらに絞り込みます。

例5：WHEREとHAVINGの組み合わせ

例えば、「中間テストのみを対象として、平均点が85点以上の講座」を取得するには：

```
SELECT course_id, AVG(score) AS 平均点
FROM grades
WHERE grade_type = '中間テスト'
GROUP BY course_id
HAVING AVG(score) >= 85;
```

実行結果：

course_id	平均点
1	87.33
…	…

HAVINGとORDER BYの組み合わせ

HAVING句で絞り込んだ結果を並べ替えるには、ORDER BY句を追加します。

例6：HAVINGとORDER BYの組み合わせ

例えば、受講者数が5人以上の講座を、受講者数の多い順に並べて取得するには：

```
SELECT course_id, COUNT(student_id) AS 受講者数
FROM student_courses
GROUP BY course_id
HAVING COUNT(student_id) >= 5
ORDER BY 受講者数 DESC;
```

実行結果：

course_id	受講者数
1	12
20	11
9	10
…	…

実践的なHAVINGの使用例

例7：「平均達成率が90%を超える評価タイプ」の抽出

各評価タイプごとの平均達成率（score/max_score）を計算し、90%を超えるものだけを取得します：

```
SELECT grade_type,
       AVG(score/max_score * 100) AS 平均達成率
FROM grades
```

```
GROUP BY grade_type
HAVING AVG(score/max_score * 100) > 90;
```

実行結果：

grade_type	平均達成率
レポート1	93.54
…	…

例8：「特定の講座で成績データが存在する学生」の抽出

例えば、講座ID「1」の成績が2件以上ある学生を取得します：

```
SELECT student_id, COUNT(*) AS 成績件数
FROM grades
WHERE course_id = '1'
GROUP BY student_id
HAVING COUNT(*) >= 2;
```

実行結果：

student_id	成績件数
301	2
302	2
…	…

HAVINGでの注意点

1. **集計関数の使用**: HAVINGで使用する条件には、通常、集計関数（COUNT、SUM、AVG、MAX、MINなど）が含まれます。単純なカラム比較だけならWHERE句を使用すべきです。
2. **パフォーマンスの考慮**: WHERE句はグループ化前に適用されるため、処理対象のレコード数を減らすことができます。可能な限り、グループ化前の条件はWHERE句で指定し、グループ化後の条件だけをHAVING句で指定するようにすると、効率的なクエリになります。
3. **グループレベルの条件**: HAVING句は、GROUP BY句で指定したカラムや集計関数の結果に対する条件でなければならないことを覚えておきましょう。

練習問題

問題11-1

student_courses（受講）テーブルから、受講者数が8人以上の講座ID（course_id）を取得するSQLを書いてください。

問題11-2

grades（成績）テーブルから、平均点（score）が85点以上の評価タイプ（grade_type）を取得するSQLを書いてください。

問題11-3

courses（講座）テーブルから、各教師（teacher_id）が担当する講座数を計算し、担当講座が3つ以上の教師だけを取得するSQLを書いてください。

問題11-4

grades（成績）テーブルから、講座ID（course_id）ごとの最高点（score）を計算し、最高点が90点以上の講座を、最高点の高い順に並べて取得するSQLを書いてください。

問題11-5

attendance（出席）テーブルから、欠席（status = 'absent'）の回数が2回以上ある学生ID（student_id）を取得するSQLを書いてください。

問題11-6

grades（成績）テーブルを使って、中間テスト（grade_type = '中間テスト'）のデータのみを対象に、平均点が85点以上で、かつ最低点が75点以上の講座ID（course_id）を取得するSQLを書いてください。

解答

解答11-1

```
SELECT course_id, COUNT(student_id) AS 受講者数
FROM student_courses
GROUP BY course_id
HAVING COUNT(student_id) >= 8;
```

解答11-2

```
SELECT grade_type, AVG(score) AS 平均点
FROM grades
GROUP BY grade_type
HAVING AVG(score) >= 85;
```

解答11-3

```
SELECT teacher_id, COUNT(course_id) AS 担当講座数
FROM courses
```

```
GROUP BY teacher_id
HAVING COUNT(course_id) >= 3;
```

解答11-4

```
SELECT course_id, MAX(score) AS 最高点
FROM grades
GROUP BY course_id
HAVING MAX(score) >= 90
ORDER BY 最高点 DESC;
```

解答11-5

```
SELECT student_id, COUNT(*) AS 欠席回数
FROM attendance
WHERE status = 'absent'
GROUP BY student_id
HAVING COUNT(*) >= 2;
```

解答11-6

```
SELECT course_id, AVG(score) AS 平均点, MIN(score) AS 最低点
FROM grades
WHERE grade_type = '中間テスト'
GROUP BY course_id
HAVING AVG(score) >= 85 AND MIN(score) >= 75;
```

まとめ

この章では、グループ化した結果に条件を適用するためのHAVING句について学びました：

1. **HAVINGの基本**：グループ化した結果に条件を適用する方法
2. **WHEREとHAVINGの違い**：
   - WHERE：グループ化前の行レベルのフィルタリング
   - HAVING：グループ化後のグループレベルのフィルタリング
3. **HAVING句での条件**：集計関数を使った条件設定
4. **複合条件**：ANDやORを使った複数条件の組み合わせ
5. **WHERE、HAVING、ORDER BYの組み合わせ**：複雑なデータ抽出の手法
6. **実践的な使用例**：実際の業務で使われるようなクエリパターン

HAVING句はデータ分析において非常に重要です。グループ化されたデータに対して「どのグループが重要か」「どのグループに注目すべきか」という条件を設定することで、より意味のある情報を抽出することができます。

次の章では、重複データを除外するための「DISTINCT：重複の除外」について学びます。

---

12. DISTINCT：重複の除外

はじめに

データベースから情報を取得する際、同じ値が複数回出現することがよくあります。例えば、「どの講座がどの教室で行われているか」を調べると、同じ教室が複数回リストされるでしょう。しかし、時には重複を除いた一意の値のリストだけが必要な場合があります。例えば：

- 「学校にはどんな教室があるか」（重複なく知りたい）
- 「どの教師が授業を担当しているか」（重複なく知りたい）
- 「どのような評価タイプがあるか」（重複なく知りたい）

このような「重複を除外」するためのSQLキーワードが「DISTINCT」です。この章では、クエリ結果から重複するデータを除外する方法について学びます。

DISTINCTの基本

DISTINCT句は、SELECT文の結果から重複する行を除外するために使います。

> **用語解説**：
>
> - **DISTINCT**：「異なる」「区別される」という意味のSQLキーワードで、クエリ結果から重複する行を除外します。
> - **重複除外**：同じ値を持つレコードを1つだけにして、残りを除外すること。

基本構文

```
SELECT DISTINCT カラム名 FROM テーブル名 [WHERE 条件];
```

DISTINCTは、SELECT文の直後に配置され、すべての指定されたカラムの組み合わせに対して働きます。

例1：単一カラムでの重複除外

例えば、成績テーブル（grades）から、どのような評価タイプ（grade_type）があるかを重複なしで取得するには：

```
SELECT DISTINCT grade_type FROM grades;
```

実行結果：

grade_type
中間テスト

grade_type
レポート1
実技試験

通常のSELECT文では、テーブル内の各行の評価タイプが表示されますが、DISTINCTを使うと、一意の評価タイプだけが表示されます。

例2：出席状況の種類の取得

出席（attendance）テーブルから、どのような出席状況（status）があるかを重複なしで取得するには：

```
SELECT DISTINCT status FROM attendance;
```

実行結果：

status
present
late
absent

複数カラムでのDISTINCT

DISTINCTは複数のカラムにも適用できます。この場合、指定したすべてのカラムの値の組み合わせが一意であるレコードだけが返されます。

基本構文

```
SELECT DISTINCT カラム名1, カラム名2, ... FROM テーブル名 [WHERE 条件];
```

例3：複数カラムでの重複除外

例えば、授業スケジュール（course_schedule）テーブルから、どの講座（course_id）がどの時限（period_id）に開講されているかを重複なしで取得するには：

```
SELECT DISTINCT course_id, period_id FROM course_schedule;
```

実行結果：

course_id	period_id
1	1

course_id	period_id
2	3
3	4
4	2
...	...

この結果は、course_idとperiod_idの組み合わせが一意のレコードのみを表示します。同じ講座が異なる時限に開講される場合や、異なる講座が同じ時限に開講される場合は、別々のレコードとして表示されます。

COUNT関数との組み合わせ

DISTINCTはCOUNT関数と組み合わせることで、一意の値の数を数えることができます。

例4：一意の値の数を数える

例えば、学生（students）テーブルに何人の学生が登録されているかを数えるには：

```
SELECT COUNT(*) AS 学生総数 FROM students;
```

実行結果：

学生総数
100

一方、受講（student_courses）テーブルに登録されている一意の学生数を数えるには：

```
SELECT COUNT(DISTINCT student_id) AS 受講学生数 FROM student_courses;
```

実行結果：

受講学生数
85

この2つの結果の差は、まだ1つも講座を受講していない学生が15人いることを意味します。

DISTINCTとNULLの扱い

DISTINCTを使う場合、NULL値も1つの値として扱われます。複数のNULL値は1つのNULL値として集約されます。

例5：NULLを含むデータでのDISTINCT

例えば、出席（attendance）テーブルから、コメント（comment）の一意の値を取得するとします：

```
SELECT DISTINCT comment FROM attendance;
```

実行結果：

comment
NULL
15分遅刻
5分遅刻
事前連絡あり
体調不良

...

この結果には、NULL値も1つの行として含まれています。

DISTINCTとORDER BYの組み合わせ

DISTINCTはORDER BY句と組み合わせて使うことができます。これにより、重複を除外した後で結果を並べ替えることができます。

例6：DISTINCTとORDER BYの組み合わせ

例えば、講座（courses）テーブルから、どの教師（teacher_id）が講座を担当しているかを重複なしで取得し、教師IDの順に並べるには：

```
SELECT DISTINCT teacher_id FROM courses ORDER BY teacher_id;
```

実行結果：

teacher_id
101
102
103
104

...

DISTINCTとWHEREの組み合わせ

DISTINCTはWHERE句と組み合わせて使うこともできます。これにより、特定の条件に合うレコードだけを対象に重複を除外できます。

例7：DISTINCTとWHEREの組み合わせ

例えば、2025年5月に授業がある教室（classroom_id）の一覧を重複なしで取得するには：

```
SELECT DISTINCT classroom_id
FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31';
```

実行結果：

classroom_id
101A
102B
201C
202D
...

DISTINCTとGROUP BYの違い

DISTINCTとGROUP BYはどちらも「重複を除外する」という点では似ていますが、目的と使い方に違いがあります。

- **DISTINCT**：単純に重複する行を除外します。
- **GROUP BY**：グループごとに集計を行うために使います。集計関数（COUNT、SUM、AVG、MAX、MINなど）と一緒に使うことが一般的です。

例8：DISTINCTとGROUP BYの比較

例えば、講座（courses）テーブルから、どの教師（teacher_id）が講座を担当しているかを調べる場合：

DISTINCTを使う方法：

```
SELECT DISTINCT teacher_id FROM courses;
```

GROUP BYを使う方法：

```
SELECT teacher_id FROM courses GROUP BY teacher_id;
```

両方とも同じ結果を返しますが、目的が異なります。GROUP BYは集計を行うためのもので、例えば各教師が担当する講座数を数えたい場合は：

```
SELECT teacher_id, COUNT(*) AS 担当講座数
FROM courses
GROUP BY teacher_id;
```

このように、GROUP BYと集計関数を組み合わせて使います。

パフォーマンスへの影響と注意点

1. **処理コスト**：DISTINCTはすべての行を調査して重複を除外するため、大量のデータがある場合は処理コストが高くなる可能性があります。
2. **代替手段の検討**：場合によっては、DISTINCTの代わりにGROUP BYを使ったり、EXISTS/NOT EXISTSを使ったりと、より効率的な方法があることがあります。
3. **部分一致には使えない**：DISTINCTは完全に一致するレコードのみを対象とします。部分的な一致や似ているレコードの除外には使えません。

練習問題

問題12-1

course_schedule（授業カレンダー）テーブルから、授業が行われる日付（schedule_date）のリストを重複なしで取得するSQLを書いてください。

問題12-2

grades（成績）テーブルから、何種類の評価タイプ（grade_type）があるかを数えるSQLを書いてください。

問題12-3

student_courses（受講）テーブルから、講座を受講している学生ID（student_id）を重複なしで取得し、IDの昇順に並べるSQLを書いてください。

問題12-4

course_schedule（授業カレンダー）テーブルから、どの教室（classroom_id）でどの時限（period_id）に授業が行われているかの組み合わせを重複なしで取得するSQLを書いてください。

問題12-5

attendance（出席）テーブルから、コメント（comment）が入力されている（NULLでない）一意のコメント内容を取得するSQLを書いてください。

問題12-6

grades（成績）テーブルから、どの学生（student_id）がどの講座（course_id）を受講したかの組み合わせを重複なしで取得し、学生ID、講座IDの順に並べるSQLを書いてください。

解答

解答12-1

```
SELECT DISTINCT schedule_date FROM course_schedule;
```

解答12-2

```
SELECT COUNT(DISTINCT grade_type) AS 評価タイプ数 FROM grades;
```

解答12-3

```
SELECT DISTINCT student_id FROM student_courses ORDER BY student_id;
```

解答12-4

```
SELECT DISTINCT classroom_id, period_id FROM course_schedule;
```

解答12-5

```
SELECT DISTINCT comment FROM attendance WHERE comment IS NOT NULL;
```

解答12-6

```
SELECT DISTINCT student_id, course_id
FROM grades
ORDER BY student_id, course_id;
```

まとめ

この章では、クエリ結果から重複を除外するためのDISTINCTキーワードについて学びました：

1. **DISTINCTの基本**：クエリ結果から重複する行を除外する方法
2. **単一カラムでの使用**：1つのカラムの重複を除外する方法
3. **複数カラムでの使用**：複数カラムの組み合わせの重複を除外する方法
4. **COUNT関数との組み合わせ**：一意の値の数を数える方法
5. **NULLの扱い**：DISTINCT使用時のNULL値の取り扱い
6. **ORDER BYとの組み合わせ**：重複除外後の並べ替え
7. **WHEREとの組み合わせ**：条件付きでの重複除外
8. **GROUP BYとの違い**：似た機能を持つGROUP BYとの使い分け

9. **パフォーマンスへの影響**：DISTINCTを使用する際の注意点

DISTINCTは、データベースから一意の値のリストを取得するための便利な機能です。特に、テーブル内の特定のカラムにどのような値が存在するかを調べたい場合や、重複のないマスターリストを作成したい場合に役立ちます。

次の章では、複数のテーブルを結合して情報を取得するための「JOIN基本：テーブル結合の概念」について学びます。

SQL学習テキスト 完全版

このテキストは、SQLの基礎から応用まで体系的に学習できるように構成されています。 学校データベースを使った実践的な例題と練習問題を通じて、実務で使えるSQLスキルを身につけることができます。

目次

---

1. SELECT基本：単一テーブルから特定カラムを取得する

はじめに

データベースからデータを取り出す作業は、料理人が大きな冷蔵庫から必要な材料だけを取り出すようなものです。SQLでは、この「取り出す」作業を「SELECT文」で行います。

SELECT文はSQLの中で最も基本的で、最もよく使われる命令です。この章では、単一のテーブル（データの表）から必要な情報だけを取り出す方法を学びます。

基本構文

SELECT文の最も基本的な形は次のとおりです：

```
SELECT カラム名 FROM テーブル名;
```

この文は「テーブル名というテーブルからカラム名という列のデータを取り出してください」という意味です。

> **用語解説：**
>
> - **SELECT**：「選択する」という意味のSQLコマンドで、データを取り出すときに使います。
> - **カラム**：テーブルの縦の列のことで、同じ種類のデータが並んでいます（例：名前のカラム、年齢のカラムなど）。
> - **FROM**：「～から」という意味で、どのテーブルからデータを取るかを指定します。
> - **テーブル**：データベース内の表のことで、行と列で構成されています。

実践例：単一カラムの取得

学校データベースの中の「teachers」テーブル（教師テーブル）から、教師の名前だけを取得してみましょう。

```
SELECT teacher_name FROM teachers;
```

実行結果：

teacher_name
寺内鞍
田尻朋美
内村海凪
藤本理恵
黒木大介
星野涼子
深山誠一
吉岡由佳
山田太郎
佐藤花子

…

これは「teachers」テーブルの「teacher_name」という列（先生の名前）だけを取り出しています。

複数のカラムを取得する

料理に複数の材料が必要なように、データを取り出すときも複数の列が必要なことがよくあります。複数のカラムを取得するには、カラム名をカンマ（,）で区切って指定します。

```
SELECT カラム名1, カラム名2, カラム名3 FROM テーブル名;
```

例えば、教師の番号（ID）と名前を一緒に取得してみましょう：

```
SELECT teacher_id, teacher_name FROM teachers;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍
102	田尻朋美
103	内村海凪
104	藤本理恵
105	黒木大介
106	星野涼子
…	…

すべてのカラムを取得する

テーブルのすべての列を取得したい場合は、アスタリスク（*）を使います。これは「すべての列」を意味するワイルドカードです。

```
SELECT * FROM テーブル名;
```

例：

```
SELECT * FROM teachers;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍
102	田尻朋美
103	内村海凪
…	…

> **注意**：`SELECT *`は便利ですが、実際の業務では必要なカラムだけを指定する方が良いとされています。これは、データ量が多いときに処理速度が遅くなるのを防ぐためです。

カラムに別名をつける（AS句）

取得したカラムに分かりやすい名前（別名）をつけることができます。これは「AS」句を使います。

```
SELECT カラム名 AS 別名 FROM テーブル名;
```

> **用語解説**：
>
> - **AS**：「～として」という意味で、カラムに別名をつけるときに使います。この別名は結果を表示するときだけ使われます。

例えば、教師IDを「番号」、教師名を「名前」として表示してみましょう：

```
SELECT teacher_id AS 番号, teacher_name AS 名前 FROM teachers;
```

実行結果：

番号	名前
101	寺内鞍
102	田尻朋美
103	内村海凪
…	…

ASは省略することも可能です：

```
SELECT teacher_id 番号, teacher_name 名前 FROM teachers;
```

計算式を使う

SELECT文では、カラムの値を使った計算もできます。例えば、成績テーブルから点数と満点を取得して、達成率（パーセント）を計算してみましょう。

```
SELECT student_id, course_id, grade_type,
       score, max_score,
       (score / max_score) * 100 AS 達成率
FROM grades;
```

実行結果：

student_id	course_id	grade_type	score	max_score	達成率

student_id	course_id	grade_type	score	max_score	達成率
301	1	中間テスト	85.5	100.0	85.5
302	1	中間テスト	92.0	100.0	92.0
…	…	…	…	…	…

重複を除外する（DISTINCT）

同じ値が複数ある場合に、重複を除いて一意の値だけを表示するには「DISTINCT」キーワードを使います。

> **用語解説**：
>
> - **DISTINCT**：「異なる」「区別された」という意味で、重複する値を除外して一意の値だけを取得します。

例えば、どの講座にどの教師が担当しているかを重複なしで確認してみましょう：

```
SELECT DISTINCT teacher_id FROM courses;
```

実行結果：

teacher_id
101
102
103
104
…

これにより、courses（講座）テーブルで使われている教師IDが重複なく表示されます。

文字列の結合

文字列を結合するには、MySQLでは「CONCAT」関数を使います。例えば、教師のIDと名前を組み合わせて表示してみましょう：

```
SELECT CONCAT('教師ID:', teacher_id, ' 名前:', teacher_name) AS 教師情報
FROM teachers;
```

実行結果：

教師情報
教師ID:101 名前:寺内鞍

教師情報
教師ID:102 名前:田尻朋美

...

> **用語解説：**
>
> - **CONCAT**：複数の文字列を一つにつなげる関数です。

SELECT文と終了記号

SQLの文は通常、セミコロン（;）で終わります。これは「この命令はここで終わりです」という合図です。

複数のSQL文を一度に実行する場合は、それぞれの文の最後にセミコロンをつけます。

練習問題

問題1-1

students（学生）テーブルから、すべての学生の名前（student_name）を取得するSQLを書いてください。

問題1-2

classrooms（教室）テーブルから、教室ID（classroom_id）と教室名（classroom_name）を取得するSQLを書いてください。

問題1-3

courses（講座）テーブルから、すべての列（カラム）を取得するSQLを書いてください。

問題1-4

class_periods（授業時間）テーブルから、時限ID（period_id）、開始時間（start_time）、終了時間（end_time）を取得し、開始時間には「開始」、終了時間には「終了」という別名をつけるSQLを書いてください。

問題1-5

grades（成績）テーブルから、学生ID（student_id）、講座ID（course_id）、評価タイプ（grade_type）、得点（score）、満点（max_score）、そして得点を満点で割って100を掛けた値を「パーセント」という別名で取得するSQLを書いてください。

問題1-6

course_schedule（授業カレンダー）テーブルから、schedule_date（予定日）カラムだけを重複なしで取得するSQLを書いてください。

解答

解答1-1

```
SELECT student_name FROM students;
```

解答1-2

```
SELECT classroom_id, classroom_name FROM classrooms;
```

解答1-3

```
SELECT * FROM courses;
```

解答1-4

```
SELECT period_id, start_time AS 開始, end_time AS 終了 FROM class_periods;
```

解答1-5

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score / max_score) * 100 AS パーセント
FROM grades;
```

解答1-6

```
SELECT DISTINCT schedule_date FROM course_schedule;
```

まとめ

この章では、SQLのSELECT文の基本を学びました：

1. 単一カラムの取得: `SELECT カラム名 FROM テーブル名;`
2. 複数カラムの取得: `SELECT カラム名1, カラム名2 FROM テーブル名;`
3. すべてのカラムの取得: `SELECT * FROM テーブル名;`
4. カラムに別名をつける: `SELECT カラム名 AS 別名 FROM テーブル名;`
5. 計算式を使う: `SELECT カラム名, (計算式) AS 別名 FROM テーブル名;`
6. 重複を除外する: `SELECT DISTINCT カラム名 FROM テーブル名;`
7. 文字列の結合: `SELECT CONCAT(文字列1, カラム名, 文字列2) FROM テーブル名;`

これらの基本操作を使いこなせるようになれば、データベースから必要な情報を効率よく取り出せるようになります。次の章では、WHERE句を使って条件に合ったデータだけを取り出す方法を学びます。

2. WHERE句：条件に合ったレコードを絞り込む

はじめに

前章では、テーブルからデータを取得する基本的な方法を学びました。しかし実際の業務では、すべてのデータではなく、特定の条件に合ったデータだけを取得したいことがほとんどです。

例えば、「全生徒の情報」ではなく「特定の学科の生徒だけ」や「成績が80点以上の学生だけ」といった形で、データを絞り込みたい場合があります。

このような場合に使用するのが「WHERE句」です。WHERE句は、SELECTコマンドの後に追加して使い、条件に合致するレコード（行）だけを取得します。

基本構文

WHERE句の基本的な形は次のとおりです：

```
SELECT カラム名 FROM テーブル名 WHERE 条件式;
```

> **用語解説**：
>
> - **WHERE**：「〜の場所で」「〜の条件で」という意味のSQLコマンドで、条件に合うデータだけを抽出するために使います。
> - **条件式**：データが満たすべき条件を指定するための式です。例えば「age > 20」（年齢が20より大きい）などです。
> - **レコード**：テーブルの横の行のことで、1つのデータの集まりを表します。

基本的な比較演算子

WHERE句では、様々な比較演算子を使って条件を指定できます：

演算子	意味	例
=	等しい	age = 25
<>	等しくない（≠と同じ）	gender <> 'male'
>	より大きい	score > 80
<	より小さい	price < 1000
>=	以上	height >= 170
<=	以下	weight <= 70

実践例：基本的な条件での絞り込み

例1：等しい（=）

例えば、教師ID（teacher_id）が101の教師のみを取得するには：

```
SELECT * FROM teachers WHERE teacher_id = 101;
```

実行結果：

teacher_id	teacher_name
101	寺内鞍

例2：より大きい（>）

成績（grades）テーブルから、90点を超える成績だけを取得するには：

```
SELECT * FROM grades WHERE score > 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

例3：等しくない（<>）

講座IDが3ではない講座に関する成績を取得するには：

```
SELECT * FROM grades WHERE course_id <> '3';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
301	2	実技試験	88.0	100.0	2025-05-18
…	…	…	…	…	…

文字列の比較

テキスト（文字列）を条件にする場合は、シングルクォーテーション（'）またはダブルクォーテーション（"）で囲みます。MySQLではどちらも使えますが、多くの場合シングルクォーテーションが推奨されます。

```
SELECT * FROM テーブル名 WHERE テキストカラム = 'テキスト値';
```

例えば、教師名（teacher_name）が「田尻朋美」の教師を検索するには：

```
SELECT * FROM teachers WHERE teacher_name = '田尻朋美';
```

実行結果：

teacher_id	teacher_name
102	田尻朋美

日付の比較

日付の比較も同様にシングルクォーテーションで囲みます。日付の形式はデータベースの設定によって異なりますが、一般的にはISO形式（YYYY-MM-DD）が使われます。

```
SELECT * FROM テーブル名 WHERE 日付カラム = '日付';
```

例えば、2025年5月20日に提出された成績を検索するには：

```
SELECT * FROM grades WHERE submission_date = '2025-05-20';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
303	1	中間テスト	78.5	100.0	2025-05-20
…	…	…	…	…	…

また、日付同士の大小関係も比較できます：

```
SELECT * FROM grades WHERE submission_date < '2025-05-15';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date

student_id	course_id	grade_type	score	max_score	submission_date
301	1	レポート1	45.0	50.0	2025-05-10
302	1	レポート1	48.0	50.0	2025-05-10
304	5	ER図作成課題	27.5	30.0	2025-05-14
…	…	…	…	…	…

複数の条件を指定する（次章の内容）

WHERE句では、複数の条件を組み合わせることもできます。この詳細は次章「論理演算子：AND、OR、NOTを使った複合条件」で説明します。

例えば、講座IDが1で、かつ、得点が90以上の成績を取得するには：

```
SELECT * FROM grades WHERE course_id = '1' AND score >= 90;
```

この例の詳細な説明は次章で行います。

練習問題

問題2-1

students（学生）テーブルから、学生ID（student_id）が「310」の学生情報を取得するSQLを書いてください。

問題2-2

classrooms（教室）テーブルから、収容人数（capacity）が30人より多い教室の情報をすべて取得するSQLを書いてください。

問題2-3

courses（講座）テーブルから、教師ID（teacher_id）が「105」の講座情報を取得するSQLを書いてください。

問題2-4

course_schedule（授業カレンダー）テーブルから、「2025-05-15」の授業スケジュールをすべて取得するSQLを書いてください。

問題2-5

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」で、得点（score）が80点未満の成績を取得するSQLを書いてください。

問題2-6

teachers（教師）テーブルから、教師ID（teacher_id）が「101」ではない教師の名前を取得するSQLを書いてください。

解答

解答2-1

```
SELECT * FROM students WHERE student_id = 310;
```

解答2-2

```
SELECT * FROM classrooms WHERE capacity > 30;
```

解答2-3

```
SELECT * FROM courses WHERE teacher_id = 105;
```

解答2-4

```
SELECT * FROM course_schedule WHERE schedule_date = '2025-05-15';
```

解答2-5

```
SELECT * FROM grades WHERE grade_type = '中間テスト' AND score < 80;
```

解答2-6

```
SELECT teacher_name FROM teachers WHERE teacher_id <> 101;
```

まとめ

この章では、WHERE句を使って条件に合ったレコードを取得する方法を学びました：

1. 基本的な比較演算子（=, <>, >, <, >=, <=）の使い方
2. 数値による条件絞り込み
3. 文字列（テキスト）による条件絞り込み
4. 日付による条件絞り込み

WHERE句は、大量のデータから必要な情報だけを取り出すための非常に重要な機能です。実際のデータベース操作では、この条件絞り込みを頻繁に使います。

次の章では、複数の条件を組み合わせるための「論理演算子（AND、OR、NOT）」について学びます。

---

3. 論理演算子：AND、OR、NOTを使った複合条件

はじめに

前章では、WHERE句を使って単一の条件でデータを絞り込む方法を学びました。しかし実際の業務では、より複雑な条件でデータを絞り込む必要があることがよくあります。

例えば：

- 「成績が80点以上**かつ**出席率が90%以上の学生」
- 「数学**または**英語の成績が優秀な学生」
- 「課題を**まだ提出していない**学生」

このような複合条件を指定するために使うのが論理演算子です。主な論理演算子は次の3つです：

- **AND**：両方の条件を満たす（かつ）
- **OR**：いずれかの条件を満たす（または）
- **NOT**：条件を満たさない（〜ではない）

AND演算子

AND演算子は、指定した**すべての条件を満たす**レコードだけを取得したいときに使います。

> **用語解説**：
>
> - **AND**：「かつ」「そして」という意味の論理演算子です。複数の条件をすべて満たすデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 条件1 AND 条件2;
```

例：ANDを使った複合条件

例えば、中間テストで90点以上かつ満点が100点の成績レコードを取得するには：

```
SELECT * FROM grades
WHERE score >= 90 AND max_score = 100;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

この例では、「score >= 90」と「max_score = 100」の両方の条件を満たすレコードだけが取得されます。

3つ以上の条件の組み合わせ

ANDを使って3つ以上の条件を組み合わせることもできます：

```
SELECT * FROM grades
WHERE score >= 90 AND max_score = 100 AND grade_type = '中間テスト';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…

OR演算子

OR演算子は、指定した条件の**いずれか一つでも満たす**レコードを取得したいときに使います。

> **用語解説**：
>
> - **OR**：「または」「もしくは」という意味の論理演算子です。複数の条件のうち少なくとも1つを満たすデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 条件1 OR 条件2;
```

例：ORを使った複合条件

例えば、教師IDが101または102の講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id = 101 OR teacher_id = 102;
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
2	UNIX入門	102
3	Cプログラミング演習	101
29	コードリファクタリングとクリーンコード	101
40	ソフトウェアアーキテクチャパターン	102
…	…	…

この例では、「teacher_id = 101」または「teacher_id = 102」のいずれかの条件を満たすレコードが取得されます。

NOT演算子

NOT演算子は、指定した条件を**満たさない**レコードを取得したいときに使います。

> **用語解説**：
>
> - **NOT**：「～ではない」という意味の論理演算子です。条件を否定して、その条件を満たさないデータを取得します。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE NOT 条件;
```

例：NOTを使った否定条件

例えば、完了（completed）状態ではない授業スケジュールを取得するには：

```
SELECT * FROM course_schedule
WHERE NOT status = 'completed';
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
45	11	2025-05-20	5	401G	106	scheduled

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
46	12	2025-05-21	1	301E	107	scheduled
50	2	2025-05-23	3	101A	102	cancelled
…	…	…	…	…	…	…

この例では、statusが「completed」ではないレコード（scheduled状態やcancelled状態）が取得されます。

NOT演算子は、「～ではない」という否定の条件を作るために使われます。例えば次の2つの書き方は同じ意味になります：

```
SELECT * FROM teachers WHERE NOT teacher_id = 101;
SELECT * FROM teachers WHERE teacher_id <> 101;
```

複合論理条件（AND、OR、NOTの組み合わせ）

AND、OR、NOTを組み合わせて、より複雑な条件を指定することもできます。

例：ANDとORの組み合わせ

例えば、「成績が90点以上で中間テストである」または「成績が45点以上でレポートである」レコードを取得するには：

```
SELECT * FROM grades
WHERE (score >= 90 AND grade_type = '中間テスト')
   OR (score >= 45 AND grade_type = 'レポート1');
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
302	1	レポート1	48.0	50.0	2025-05-10
308	1	レポート1	47.0	50.0	2025-05-09
…	…	…	…	…	…

優先順位と括弧の使用

論理演算子を組み合わせる場合、演算子の優先順位に注意が必要です。基本的に**ANDはORよりも優先順位が高い**です。つまり、ANDが先に処理されます。

例えば：

```
WHERE 条件1 OR 条件2 AND 条件3
```

これは次のように解釈されます：

```
WHERE 条件1 OR (条件2 AND 条件3)
```

意図した条件と異なる場合は、**括弧（）を使って明示的にグループ化**することが重要です：

```
WHERE (条件1 OR 条件2) AND 条件3
```

例：括弧を使った条件のグループ化

教師IDが101または102で、かつ、講座名に「プログラミング」という単語が含まれる講座を取得するには：

```sql
SELECT * FROM courses
WHERE (teacher_id = 101 OR teacher_id = 102)
  AND course_name LIKE '%プログラミング%';
```

実行結果：

course_id	course_name	teacher_id
3	Cプログラミング演習	101
…	…	…

この例では、括弧を使って「teacher_id = 101 OR teacher_id = 102」の部分をグループ化し、その条件と「course_name LIKE '%プログラミング%'」の条件をANDで結合しています。

練習問題

問題3-1

grades（成績）テーブルから、課題タイプ（grade_type）が「中間テスト」かつ点数（score）が85点以上のレコードを取得するSQLを書いてください。

問題3-2

classrooms（教室）テーブルから、収容人数（capacity）が40人以下または建物（building）が「1号館」の教室を取得するSQLを書いてください。

問題3-3

teachers（教師）テーブルから、教師ID（teacher_id）が101、102、103ではない教師の情報を取得するSQLを書いてください。

問題3-4

course_schedule（授業カレンダー）テーブルから、2025年5月20日の授業で、時限（period_id）が1か2で、かつ状態（status）が「scheduled」の授業を取得するSQLを書いてください。

問題3-5

students（学生）テーブルから、学生名（student_name）に「田」または「山」を含む学生を取得するSQLを書いてください。

問題3-6

grades（成績）テーブルから、提出日（submission_date）が2025年5月15日以降で、かつ（「中間テスト」で90点以上または「レポート1」で45点以上）の成績を取得するSQLを書いてください。

解答

解答3-1

```
SELECT * FROM grades
WHERE grade_type = '中間テスト' AND score >= 85;
```

解答3-2

```
SELECT * FROM classrooms
WHERE capacity <= 40 OR building = '1号館';
```

解答3-3

```
SELECT * FROM teachers
WHERE NOT (teacher_id = 101 OR teacher_id = 102 OR teacher_id = 103);
```

または

```
SELECT * FROM teachers
WHERE teacher_id <> 101 AND teacher_id <> 102 AND teacher_id <> 103;
```

解答3-4

```sql
SELECT * FROM course_schedule
WHERE schedule_date = '2025-05-20'
  AND (period_id = 1 OR period_id = 2)
  AND status = 'scheduled';
```

解答3-5

```sql
SELECT * FROM students
WHERE student_name LIKE '%田%' OR student_name LIKE '%山%';
```

解答3-6

```sql
SELECT * FROM grades
WHERE submission_date >= '2025-05-15'
  AND ((grade_type = '中間テスト' AND score >= 90)
       OR (grade_type = 'レポート1' AND score >= 45));
```

まとめ

この章では、論理演算子（AND、OR、NOT）を使って複合条件を作る方法を学びました：

1. **AND**：すべての条件を満たすレコードを取得（条件1 AND 条件2）
2. **OR**：いずれかの条件を満たすレコードを取得（条件1 OR 条件2）
3. **NOT**：指定した条件を満たさないレコードを取得（NOT 条件）
4. **複合条件**：AND、OR、NOTを組み合わせたより複雑な条件
5. **括弧（）**：条件をグループ化して優先順位を明示的に指定

これらの論理演算子を使いこなすことで、より複雑で細かな条件でデータを絞り込むことができるようになります。実際のデータベース操作では、複数の条件を組み合わせることが頻繁にあるため、この章で学んだ内容は非常に重要です。

次の章では、テキストデータに対する検索を行う「パターンマッチング」について学びます。

---

4. パターンマッチング：LIKE演算子と%、_ワイルドカード

はじめに

前章までは、データの完全一致や数値の比較といった条件での絞り込みを学びました。しかし実際の業務では、もっと柔軟な検索が必要なケースがあります。例えば：

- 「山」で始まる名前の学生を検索したい
- 「プログラミング」という単語を含む講座名を探したい

- 電話番号の一部だけ覚えているデータを探したい

このような「部分一致」や「パターン一致」の検索を行うためのSQLの機能が「パターンマッチング」です。この章では、パターンマッチングを行うための「LIKE演算子」と「ワイルドカード文字」について学びます。

LIKE演算子の基本

LIKE演算子は、文字列のパターンマッチングを行うための演算子です。WHERE句と組み合わせて使用します。

> **用語解説**：
>
> - **LIKE**：「～のような」という意味の演算子で、パターンに一致する文字列を検索します。
> - **パターンマッチング**：完全一致ではなく、一定のパターンに合致するデータを検索する方法です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE 文字列カラム LIKE 'パターン';
```

パターンには、通常の文字に加えて、特別な意味を持つ「ワイルドカード文字」を使用できます。

ワイルドカード文字

SQLでは主に2つのワイルドカード文字があります：

1. **%（パーセント）**：0文字以上の任意の文字列に一致します。
2. _**（アンダースコア）**：任意の1文字に一致します。

> **用語解説**：
>
> - **ワイルドカード**：任意の文字や文字列に一致する特殊な文字記号です。トランプのジョーカーのように、様々な値に代用できます。

LIKE演算子の使い方：実践例

例1：%（パーセント）を使ったパターンマッチング

「～で始まる」パターン：前方一致

例えば、「山」で始まる学生名を検索するには：

```
SELECT * FROM students WHERE student_name LIKE '山%';
```

実行結果：

student_id	student_name
312	山本裕子
325	山田翔太
…	…

ここでの「山%」は「山で始まり、その後に0文字以上の任意の文字が続く」という意味です。

「〜で終わる」パターン：後方一致

例えば、「子」で終わる教師名を検索するには：

```
SELECT * FROM teachers WHERE teacher_name LIKE '%子';
```

実行結果：

teacher_id	teacher_name
102	田尻朋美
106	星野涼子
108	吉岡由佳
110	佐藤花子
…	…

「〜を含む」パターン：部分一致

例えば、「プログラミング」という単語を含む講座名を検索するには：

```
SELECT * FROM courses WHERE course_name LIKE '%プログラミング%';
```

実行結果：

course_id	course_name	teacher_id
3	Cプログラミング演習	101
14	IoTデバイスプログラミング実践	110
…	…	…

例2：_（アンダースコア）を使ったパターンマッチング

アンダースコアは任意の1文字に一致します。例えば、教室IDが「10_A」パターン（最初の2文字が「10」、3文字目が任意の1文字、最後が「A」）の教室を検索するには：

```
SELECT * FROM classrooms WHERE classroom_id LIKE '10_A';
```

実行結果：

classroom_id	classroom_name	capacity	building	facilities
101A	1号館コンピュータ実習室A	30	1号館	パソコン30台、プロジェクター
…	…	…	…	…

例3：%と_の組み合わせ

ワイルドカード文字は組み合わせて使うこともできます。例えば、「2文字目が田」の学生を検索するには：

```
SELECT * FROM students WHERE student_name LIKE '_田%';
```

実行結果：

student_id	student_name
321	井上竜也
384	櫻井翼
…	…

NOT LIKEを使った否定形のパターンマッチング

特定のパターンに一致しないレコードを検索したい場合は、「NOT LIKE」を使います。

> **用語解説**：
>
> - **NOT LIKE**：「～のパターンに一致しない」という意味で、指定したパターンに一致しないデータを検索します。

例えば、講座名に「入門」を含まない講座を検索するには：

```
SELECT * FROM courses WHERE course_name NOT LIKE '%入門%';
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
3	Cプログラミング演習	101
…	…	…

エスケープ文字の使用

もし検索したいパターンに「%」や「_」自体が含まれている場合は、それらを特別な文字としてではなく、通常の文字として扱うために「エスケープ文字」を使います。

> **用語解説**：
>
> - **エスケープ文字**：特別な意味を持つ文字を通常の文字として扱うための印です。

MySQLでは、バックスラッシュ（\）をエスケープ文字として使用できます。例えば、「50%」という値そのものを検索するには：

```
SELECT * FROM テーブル名 WHERE カラム LIKE '50\%';
```

または、ESCAPE句を使って明示的にエスケープ文字を指定することもできます：

```
SELECT * FROM テーブル名 WHERE カラム LIKE '50!%' ESCAPE '!';
```

この例では「!」をエスケープ文字として指定しています。

大文字と小文字の区別

MySQLのデフォルト設定では、LIKE演算子は大文字と小文字を区別しません（大文字小文字を同じものとして扱います）。

例えば、次の2つのクエリは同じ結果を返します：

```
SELECT * FROM courses WHERE course_name LIKE '%web%';
SELECT * FROM courses WHERE course_name LIKE '%Web%';
```

もし大文字と小文字を区別した検索が必要な場合は、「BINARY」キーワードを使用します：

```
SELECT * FROM courses WHERE course_name LIKE BINARY '%Web%';
```

この場合、「Web」は「web」とは一致しません。

複合条件との組み合わせ

LIKE演算子は、これまで学んだAND、OR、NOTなどの論理演算子と組み合わせて使うこともできます。

例えば、「田」で始まる名前で、かつ教師IDが102から105の間の教師を検索するには：

```
SELECT * FROM teachers
WHERE teacher_name LIKE '田%'
```

```
    AND teacher_id BETWEEN 102 AND 105;
```

実行結果：

teacher_id	teacher_name
102	田尻朋美
...	...

練習問題

問題4-1

students（学生）テーブルから、学生名（student_name）が「佐藤」で始まる学生の情報をすべて取得するSQLを書いてください。

問題4-2

courses（講座）テーブルから、講座名（course_name）に「データ」という単語を含む講座の情報を取得するSQLを書いてください。

問題4-3

classrooms（教室）テーブルから、教室名（classroom_name）が「コンピュータ実習室」で終わる教室の情報を取得するSQLを書いてください。

問題4-4

teachers（教師）テーブルから、教師名（teacher_name）の2文字目が「木」である教師の情報を取得するSQLを書いてください。

問題4-5

courses（講座）テーブルから、講座名（course_name）に「入門」または「基礎」を含む講座を取得するSQLを書いてください。

問題4-6

students（学生）テーブルから、学生名（student_name）が「山」で始まり、かつ「子」で終わらない学生を取得するSQLを書いてください。

解答

解答4-1

```
SELECT * FROM students WHERE student_name LIKE '佐藤%';
```

解答4-2

```
SELECT * FROM courses WHERE course_name LIKE '%データ%';
```

解答4-3

```
SELECT * FROM classrooms WHERE classroom_name LIKE '%コンピュータ実習室';
```

解答4-4

```
SELECT * FROM teachers WHERE teacher_name LIKE '_木%';
```

解答4-5

```
SELECT * FROM courses
WHERE course_name LIKE '%入門%' OR course_name LIKE '%基礎%';
```

解答4-6

```
SELECT * FROM students
WHERE student_name LIKE '山%' AND student_name NOT LIKE '%子';
```

まとめ

この章では、パターンマッチングを行うためのLIKE演算子と、その中で使用するワイルドカード文字（%と_）について学びました：

1. **LIKE演算子**：文字列パターンに一致するデータを検索するための演算子
2. **%（パーセント）**：0文字以上の任意の文字列に一致するワイルドカード
3. **_（アンダースコア）**：任意の1文字に一致するワイルドカード
4. **前方一致**：「パターン%」で「パターンで始まる」文字列に一致
5. **後方一致**：「%パターン」で「パターンで終わる」文字列に一致
6. **部分一致**：「%パターン%」で「パターンを含む」文字列に一致
7. **NOT LIKE**：指定したパターンに一致しないデータを検索
8. **エスケープ文字**：特殊文字（%や_）を通常の文字として扱うための方法
9. **複合条件との組み合わせ**：AND、ORなどと組み合わせたより複雑な条件

パターンマッチングは、特にテキストデータを扱う際に非常に便利な機能です。部分的な情報しか持っていない場合や、特定のパターンを持つデータを探す場合に活用できます。

次の章では、範囲指定のための「BETWEEN演算子」と「IN演算子」について学びます。

5. 範囲指定：BETWEEN、IN演算子

はじめに

これまでの章では、等号（=）や不等号（>、<）を使って条件を指定する方法や、LIKE演算子を使ったパターンマッチングを学びました。この章では、値の範囲を指定する「BETWEEN演算子」と、複数の値を一度に指定できる「IN演算子」について学びます。

これらの演算子を使うと、次のような検索がより簡単になります：

- 「80点から90点の間の成績」
- 「2025年4月から6月の間のスケジュール」
- 「特定の教師IDリストに該当する講座」

BETWEEN演算子：範囲を指定する

BETWEEN演算子は、ある値が指定した範囲内にあるかどうかを調べるために使います。

> **用語解説**：
>
> - **BETWEEN**：「～の間に」という意味の演算子で、ある値が指定した最小値と最大値の間（両端の値を含む）にあるかどうかを判定します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 BETWEEN 最小値 AND 最大値;
```

この構文は次の条件と同じ意味です：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 >= 最小値 AND カラム名 <= 最大値;
```

例1：数値範囲の指定

例えば、成績（grades）テーブルから、80点から90点の間の成績を取得するには：

```
SELECT * FROM grades
WHERE score BETWEEN 80 AND 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	中間テスト	85.5	100.0	2025-05-20

student_id	course_id	grade_type	score	max_score	submission_date
308	6	小テスト1	89.0	100.0	2025-05-15
...	...	...	...	...	...

例2：日付範囲の指定

日付にもBETWEEN演算子が使えます。例えば、2025年5月10日から2025年5月20日までに提出された成績を取得するには：

```
SELECT * FROM grades
WHERE submission_date BETWEEN '2025-05-10' AND '2025-05-20';
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	1	レポート1	45.0	50.0	2025-05-10
302	1	レポート1	48.0	50.0	2025-05-10
301	1	中間テスト	85.5	100.0	2025-05-20
...	...	...	...	...	...

NOT BETWEEN：範囲外を指定する

NOT BETWEENを使うと、指定した範囲の外にある値を持つレコードを取得できます。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 NOT BETWEEN 最小値 AND 最大値;
```

これは次の条件と同じです：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 < 最小値 OR カラム名 > 最大値;
```

例：範囲外の値を取得

例えば、80点未満または90点より高い成績を取得するには：

```
SELECT * FROM grades
WHERE score NOT BETWEEN 80 AND 90;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
302	1	中間テスト	92.0	100.0	2025-05-20
303	1	中間テスト	78.5	100.0	2025-05-20
311	1	中間テスト	95.0	100.0	2025-05-20
...	...	...	...	...	...

IN演算子：複数の値を指定する

IN演算子は、ある値が指定した複数の値のリストのいずれかに一致するかどうかを調べるために使います。

> **用語解説**：
>
> - **IN**：「～の中に含まれる」という意味の演算子で、ある値が指定したリストの中に含まれているかどうかを判定します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 IN (値1, 値2, 値3, ...);
```

この構文は次のOR条件の組み合わせと同じ意味です：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 = 値1 OR カラム名 = 値2 OR カラム名 = 値3 OR ...;
```

例1：数値リストの指定

例えば、教師ID（teacher_id）が101、103、105のいずれかである講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id IN (101, 103, 105);
```

実行結果：

course_id	course_name	teacher_id
1	ITのための基礎知識	101
3	Cプログラミング演習	101

course_id	course_name	teacher_id
5	データベース設計と実装	105
10	プロジェクト管理手法	103
...	...	...

例2：文字列リストの指定

文字列のリストにも適用できます。例えば、特定の教室ID（classroom_id）のみの教室情報を取得するには：

```
SELECT * FROM classrooms
WHERE classroom_id IN ('101A', '202D', '301E');
```

実行結果：

classroom_id	classroom_name	capacity	building	facilities
101A	1号館コンピュータ実習室A	30	1号館	パソコン30台、プロジェクター
202D	2号館コンピュータ実習室D	25	2号館	パソコン25台、プロジェクター、3Dプリンター
301E	3号館講義室E	80	3号館	プロジェクター、マイク設備、録画設備

NOT IN：リストに含まれない値を指定する

NOT IN演算子を使うと、指定したリストに含まれない値を持つレコードを取得できます。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 NOT IN (値1, 値2, 値3, ...);
```

これは次の条件と同じです：

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 <> 値1 AND カラム名 <> 値2 AND カラム名 <> 値3 AND ...;
```

例：リストに含まれない値を取得

例えば、教師IDが101、102、103以外の教師が担当する講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id NOT IN (101, 102, 103);
```

実行結果：

course_id	course_name	teacher_id
4	Webアプリケーション開発	104
5	データベース設計と実装	105
6	ネットワークセキュリティ	107
…	…	…

IN演算子でのサブクエリの利用（基本）

IN演算子の括弧内には、直接値を書く代わりに、サブクエリ（別のSELECT文）を指定することもできます。これにより、動的に値のリストを生成できます。

> **用語解説**：
>
> - **サブクエリ**：SQL文の中に含まれる別のSQL文のことで、外側のSQL文（メインクエリ）に値や条件を提供します。

基本構文

```
SELECT カラム名 FROM テーブル名
WHERE カラム名 IN (SELECT カラム名 FROM 別テーブル WHERE 条件);
```

例：サブクエリを使ったIN条件

例えば、教師名（teacher_name）に「田」を含む教師が担当している講座を取得するには：

```
SELECT * FROM courses
WHERE teacher_id IN (
    SELECT teacher_id FROM teachers
    WHERE teacher_name LIKE '%田%'
);
```

実行結果：

course_id	course_name	teacher_id
2	UNIX入門	102
10	プロジェクト管理手法	103

course_id	course_name	teacher_id
...	...	...

この例では、まず「teachers」テーブルから名前に「田」を含む教師のIDを取得し、それらのIDを持つ講座を「courses」テーブルから取得しています。

BETWEEN演算子とIN演算子の組み合わせ

BETWEEN演算子とIN演算子は、論理演算子（AND、OR）と組み合わせて、さらに複雑な条件を作ることができます。

例：BETWEENとINの組み合わせ

例えば、「教師IDが101、103、105のいずれかで、かつ、2025年5月15日から2025年6月15日の間に実施される授業」を取得するには：

```
SELECT * FROM course_schedule
WHERE teacher_id IN (101, 103, 105)
  AND schedule_date BETWEEN '2025-05-15' AND '2025-06-15';
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
38	1	2025-05-20	1	102B	101	scheduled
42	10	2025-05-23	3	201C	103	scheduled
45	5	2025-05-28	2	402H	105	scheduled
...	...	...	...	...	...	...

練習問題

問題5-1

grades（成績）テーブルから、得点（score）が85点から95点の間にある成績を取得するSQLを書いてください。

問題5-2

course_schedule（授業カレンダー）テーブルから、2025年5月1日から2025年5月31日までの授業スケジュールを取得するSQLを書いてください。

問題5-3

courses（講座）テーブルから、教師ID（teacher_id）が104、106、108のいずれかである講座の情報を取得するSQLを書いてください。

問題5-4

classrooms（教室）テーブルから、教室ID（classroom_id）が「101A」、「201C」、「301E」、「401G」以外の教室情報を取得するSQLを書いてください。

問題5-5

grades（成績）テーブルから、評価タイプ（grade_type）が「中間テスト」または「実技試験」で、かつ得点（score）が80点から90点の間にない成績を取得するSQLを書いてください。

問題5-6

course_schedule（授業カレンダー）テーブルから、教室ID（classroom_id）が「101A」、「202D」のいずれかで、かつ2025年5月15日から2025年5月30日の間に実施される授業スケジュールを取得するSQLを書いてください。

解答

解答5-1

```
SELECT * FROM grades
WHERE score BETWEEN 85 AND 95;
```

解答5-2

```
SELECT * FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31';
```

解答5-3

```
SELECT * FROM courses
WHERE teacher_id IN (104, 106, 108);
```

解答5-4

```
SELECT * FROM classrooms
WHERE classroom_id NOT IN ('101A', '201C', '301E', '401G');
```

解答5-5

```
SELECT * FROM grades
WHERE grade_type IN ('中間テスト', '実技試験')
```

```
  AND score NOT BETWEEN 80 AND 90;
```

解答5-6

```
SELECT * FROM course_schedule
WHERE classroom_id IN ('101A', '202D')
  AND schedule_date BETWEEN '2025-05-15' AND '2025-05-30';
```

まとめ

この章では、範囲指定のための「BETWEEN演算子」と複数値指定のための「IN演算子」について学びました：

1. **BETWEEN演算子**：値が指定した範囲内（両端を含む）にあるかどうかをチェック
2. **NOT BETWEEN**：値が指定した範囲外にあるかどうかをチェック
3. **IN演算子**：値が指定したリストのいずれかに一致するかどうかをチェック
4. **NOT IN**：値が指定したリストのいずれにも一致しないかどうかをチェック
5. **サブクエリとIN**：動的に生成された値のリストを使用する方法
6. **複合条件**：BETWEEN、IN、論理演算子を組み合わせたより複雑な条件

これらの演算子を使うことで、複数の条件を指定する場合に、SQLをより簡潔に書くことができます。特に、多くの値を指定する場合や範囲条件を指定する場合に便利です。

次の章では、「NULL値の処理：IS NULL、IS NOT NULL」について学びます。

---

6. NULL値の処理：IS NULL、IS NOT NULL

はじめに

データベースの世界では、データがない状態を表すために「NULL」という特別な値が使われます。NULLは「空」や「0」や「空白文字」とは異なる、「値が存在しない」または「不明」であることを表す特殊な概念です。

例えば、学校データベースでは次のようなシナリオがあります：

- まだ成績が付けられていない（NULL）
- コメントが入力されていない（NULL）
- 授業がキャンセルされたため教室が割り当てられていない（NULL）

この章では、NULL値を正しく処理するための「IS NULL」と「IS NOT NULL」演算子について学びます。

NULLとは何か？

NULL値には、いくつかの特徴があります：

1. **値がない**：NULLは値がないことを表します。0でも空文字列（''）でもなく、値そのものが存在しないことを示します。
2. **不明**：データが不明であることを表す場合もあります。
3. **未設定**：まだ値が設定されていないことを表す場合もあります。
4. **比較できない**：NULLは通常の比較演算子（=, <, >など）で比較できません。

> **用語解説**：
>
> - **NULL**：データベースにおいて「値がない」または「不明」を表す特殊な値です。0や空文字とは異なります。

NULL値と通常の比較演算子

通常の比較演算子（=, <>, >, <, >=, <=）では、NULL値を正しく検出できません。例えば：

```
-- この条件はNULL値に対して常にFALSEを返す
SELECT * FROM テーブル名 WHERE カラム名 = NULL;

-- この条件もNULL値に対して常にFALSEを返す
SELECT * FROM テーブル名 WHERE カラム名 <> NULL;
```

これは、NULL値との等価比較は「不明」と評価されるためです。つまり、NULL = NULLでさえFALSEではなく「不明」になります。

IS NULL演算子

NULL値を持つレコードを検索するには、「IS NULL」演算子を使います。

> **用語解説**：
>
> - **IS NULL**：カラムの値がNULLかどうかを調べる演算子です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE カラム名 IS NULL;
```

例：IS NULLの使用

例えば、コメントが入力されていない（NULL）出席レコードを検索するには：

```
SELECT * FROM attendance WHERE comment IS NULL;
```

実行結果：

schedule_id	student_id	status	comment

schedule_id	student_id	status	comment
1	301	present	NULL
1	306	present	NULL
1	307	present	NULL
...	...	...	NULL

IS NOT NULL演算子

逆に、NULL値を持たないレコード（つまり、何らかの値を持つレコード）を検索するには、「IS NOT NULL」演算子を使います。

> **用語解説**：
>
> - **IS NOT NULL**：カラムの値がNULLでないかどうかを調べる演算子です。

基本構文

```
SELECT カラム名 FROM テーブル名 WHERE カラム名 IS NOT NULL;
```

例：IS NOT NULLの使用

例えば、コメントが入力されている（NOT NULL）出席レコードを検索するには：

```
SELECT * FROM attendance WHERE comment IS NOT NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	302	late	15分遅刻
1	303	absent	事前連絡あり
1	308	late	5分遅刻
...	...	...	...

NULL値の論理的な扱い

NULL値は論理演算（AND、OR、NOT）でも特殊な扱いを受けます。

- **NULL AND TRUE** → NULL（不明）
- **NULL AND FALSE** → FALSE
- **NULL OR TRUE** → TRUE
- **NULL OR FALSE** → NULL（不明）

- **NOT NULL** → NULL（不明）

この特殊な振る舞いが、バグや誤った結果の原因になることがあります。

NULL値と結合条件

テーブル結合（JOINなど、後の章で学習）の際も、NULL値は特殊な扱いを受けます。NULL値同士は「等しい」とは判定されないため、通常の結合条件ではNULL値を持つレコードは結合されません。

IS NULLとIS NOT NULLを使った複合条件

IS NULLとIS NOT NULLも、他の条件と組み合わせて使用できます。

例：複合条件でのIS NULLの使用

例えば、「出席状態が "absent"（欠席）で、コメントがNULLでない（理由が入力されている）レコード」を検索するには：

```
SELECT * FROM attendance
WHERE status = 'absent' AND comment IS NOT NULL;
```

実行結果：

schedule_id	student_id	status	comment
1	303	absent	事前連絡あり
1	317	absent	体調不良
…	…	…	…

NVL/IFNULL/COALESCE関数：NULL値の置換

NULL値を別の値に置き換えるための関数が用意されています。データベースによって関数名が異なることがありますが、機能は似ています：

- MySQL/MariaDB: **IFNULL(expr, replace_value)**
- Oracle: **NVL(expr, replace_value)**
- SQL Server: **ISNULL(expr, replace_value)**
- 標準SQL: **COALESCE(expr1, expr2, ..., exprN)** - 最初のNULLでない式を返します

例：IFNULL関数の使用（MySQL）

例えば、コメントがNULLの場合は「特記事項なし」と表示するには：

```
SELECT schedule_id, student_id, status,
       IFNULL(comment, '特記事項なし') AS comment
FROM attendance;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	特記事項なし
1	302	late	15分遅刻
1	303	absent	事前連絡あり
...	...	...	...

NULLを使う際の注意点

1. **除外の罠**：`WHERE カラム名 <> 値` だけでは、NULL値を持つレコードは含まれません。すべてのレコードを対象にするには：

   ```
   WHERE カラム名 <> 値 OR カラム名 IS NULL
   ```

2. **集計関数**：COUNT(*)はすべての行を数えますが、COUNT(カラム名)はそのカラムがNULLでない行だけを数えます。

3. **インデックス**：多くのデータベースでは、NULL値にもインデックスを適用できますが、データベースによって動作が異なる場合があります。

4. **一意性制約**：一般的に、UNIQUE制約ではNULL値は重複としてカウントされません（複数のNULL値が許可されます）。

練習問題

問題6-1

attendance（出席）テーブルから、コメント（comment）がNULLの出席レコードをすべて取得するSQLを書いてください。

問題6-2

course_schedule（授業カレンダー）テーブルから、状態（status）が「cancelled」で、かつ教室ID（classroom_id）がNULLでないレコードを取得するSQLを書いてください。

問題6-3

grades（成績）テーブルから、提出日（submission_date）がNULLの成績レコードを取得するSQLを書いてください。

問題6-4

attendance（出席）テーブルから、出席状態（status）が「present」か「late」で、かつコメント（comment）がNULLのレコードを取得するSQLを書いてください。

問題6-5

以下のSQLで教師（teachers）テーブルから「佐藤」という名前を持つ教師を検索する場合、NULL値を持つレコードも含めるにはどう修正すべきですか？

```
SELECT * FROM teachers WHERE teacher_name <> '佐藤花子';
```

問題6-6

attendance（出席）テーブルのすべてのレコードを取得し、コメント（comment）がNULLの場合は「記録なし」と表示するSQLを書いてください。

解答

解答6-1

```
SELECT * FROM attendance WHERE comment IS NULL;
```

解答6-2

```
SELECT * FROM course_schedule
WHERE status = 'cancelled' AND classroom_id IS NOT NULL;
```

解答6-3

```
SELECT * FROM grades WHERE submission_date IS NULL;
```

解答6-4

```
SELECT * FROM attendance
WHERE (status = 'present' OR status = 'late') AND comment IS NULL;
```

または

```
SELECT * FROM attendance
WHERE status IN ('present', 'late') AND comment IS NULL;
```

解答6-5

```
SELECT * FROM teachers
WHERE teacher_name <> '佐藤花子' OR teacher_name IS NULL;
```

解答6-6

```
SELECT schedule_id, student_id, status,
       IFNULL(comment, '記録なし') AS comment
FROM attendance;
```

まとめ

この章では、データベースにおけるNULL値の概念と、NULL値を扱うための演算子や関数について学びました：

1. **NULL値の概念**：値がない、不明、未設定を表す特殊な値
2. **IS NULL演算子**：NULL値を持つレコードを検索する方法
3. **IS NOT NULL演算子**：NULL値を持たないレコードを検索する方法
4. **NULL値の論理的扱い**：論理演算（AND、OR、NOT）におけるNULLの振る舞い
5. **複合条件**：IS NULL/IS NOT NULLと他の条件の組み合わせ
6. **NULL値の置換**：IFNULL/NVL/COALESCE関数の使用方法
7. **注意点**：NULL値を扱う際の一般的な落とし穴

NULL値の正確な理解と適切な処理は、SQLプログラミングの重要な部分です。不適切なNULL処理は、予期しない結果やバグの原因になります。

次の章では、クエリ結果の並び替えを行うための「ORDER BY：結果の並び替え」について学びます。

---

7. ORDER BY：結果の並び替え

はじめに

これまでの章では、データベースから条件に合ったレコードを取得する方法を学んできました。しかし実際の業務では、取得したデータを見やすく整理する必要があります。例えば：

- 成績を高い順に表示したい
- 学生を名前の五十音順に並べたい
- 日付の新しい順にスケジュールを確認したい

このようなデータの「並び替え」を行うためのSQLコマンドが「ORDER BY」です。この章では、クエリ結果を特定の順序で並べる方法を学びます。

ORDER BYの基本

ORDER BY句は、SELECT文の結果を指定したカラムの値に基づいて並び替えるために使います。

> **用語解説：**
>
> - **ORDER BY**：「～の順に並べる」という意味のSQLコマンドで、クエリ結果の並び順を指定します。

基本構文

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] ORDER BY 並び替えカラム;
```

ORDER BY句は通常、SELECT文の最後に記述します。

例1：単一カラムでの並び替え

例えば、学生（students）テーブルから、学生名（student_name）の五十音順（辞書順）でデータを取得するには：

```
SELECT * FROM students ORDER BY student_name;
```

実行結果：

student_id	student_name
309	相沢吉夫
303	柴崎春花
306	河田咲奈
305	河口菜恵子
…	…

デフォルトの並び順

ORDER BYを使わない場合、結果の順序は保証されません。多くの場合、データがデータベースに保存された順序で返されますが、これは信頼できるものではありません。

昇順と降順の指定

ORDER BY句では、並び順を「昇順」か「降順」のどちらかで指定できます。

> **用語解説：**
>
> - **昇順（ASC）**：小さい値から大きい値へ（A→Z、1→9）の順に並べます。
> - **降順（DESC）**：大きい値から小さい値へ（Z→A、9→1）の順に並べます。

構文

```
SELECT カラム名 FROM テーブル名 ORDER BY 並び替えカラム [ASC|DESC];
```

ASC（昇順）がデフォルトのため、省略可能です。

例2：降順での並び替え

例えば、成績（grades）テーブルから、得点（score）の高い順（降順）に成績を取得するには：

```
SELECT * FROM grades ORDER BY score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20
…	…	…	…	…	…

複数カラムでの並び替え

複数のカラムを使って並び替えることもできます。最初に指定したカラムで並び替え、値が同じレコードがある場合は次のカラムで並び替えます。

構文

```
SELECT カラム名 FROM テーブル名
ORDER BY 並び替えカラム1 [ASC|DESC], 並び替えカラム2 [ASC|DESC], ...;
```

例3：複数カラムでの並び替え

例えば、成績（grades）テーブルから、課題タイプ（grade_type）の五十音順に並べ、同じ課題タイプ内では得点（score）の高い順に成績を取得するには：

```
SELECT * FROM grades
ORDER BY grade_type ASC, score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
301	2	実技試験	88.0	100.0	2025-05-18

student_id	course_id	grade_type	score	max_score	submission_date
321	2	実技試験	85.5	100.0	2025-05-18
…	…	…	…	…	…
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
…	…	…	…	…	…
311	1	レポート1	49.0	50.0	2025-05-08
302	1	レポート1	48.0	50.0	2025-05-10
…	…	…	…	…	…

例4：昇順と降順の混合

各カラムごとに並び順を指定することもできます。例えば、講座ID（course_id）の昇順、評価タイプ（grade_type）の昇順、得点（score）の降順で並べるには：

```
SELECT * FROM grades
ORDER BY course_id ASC, grade_type ASC, score DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	レポート1	49.0	50.0	2025-05-08
…	…	…	…	…	…
311	1	中間テスト	95.0	100.0	2025-05-20
…	…	…	…	…	…
301	2	実技試験	88.0	100.0	2025-05-18
…	…	…	…	…	…

NULLの扱い

ORDER BYでNULL値を並び替える場合、データベース製品によって動作が異なります。多くのデータベースでは、NULL値は最小値または最大値として扱われます。

- MySQL/MariaDBでは、NULL値は昇順（ASC）の場合は最小値として（最初に表示）、降順（DESC）の場合は最大値として（最後に表示）扱われます。

一部のデータベース（PostgreSQLなど）では、NULL値の位置を明示的に指定するための「NULLS FIRST」「NULLS LAST」構文がサポートされています。

例5：NULL値の扱い

例えば、出席（attendance）テーブルからコメント（comment）でソートすると、NULLが最初に来ます：

```
SELECT * FROM attendance ORDER BY comment;
```

実行結果：

schedule_id	student_id	status	comment
1	301	present	NULL
1	306	present	NULL
...	...	...	NULL
1	308	late	5分遅刻
1	323	late	電車遅延
...	...	...	...

カラム番号を使った並び替え

カラム名の代わりに、SELECT文の結果セットにおけるカラムの位置（番号）を使って並び替えることもできます。最初のカラムは1、2番目のカラムは2、という具合です。

構文

```
SELECT カラム名1, カラム名2, ... FROM テーブル名 ORDER BY カラム位置;
```

例6：カラム番号を使った並び替え

例えば、学生（students）テーブルから学生ID（student_id）と名前（student_name）を取得し、名前（2番目のカラム）で並べ替えるには：

```
SELECT student_id, student_name FROM students ORDER BY 2;
```

この場合、「ORDER BY 2」は「ORDER BY student_name」と同じ意味になります。

> **注意**：カラム番号を使う方法は、カラムの順序を変更すると問題が起きるため、実際の業務では使用を避けた方が良いとされています。

式や関数を使った並び替え

ORDER BY句で式や関数を使うことにより、計算結果に基づいて並び替えることもできます。

例7：式を使った並び替え

例えば、成績（grades）テーブルから、得点の達成率（score/max_score）の高い順に並べるには：

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score/max_score)*100 AS 達成率
FROM grades
ORDER BY (score/max_score) DESC;
```

または

```
SELECT student_id, course_id, grade_type, score, max_score,
       (score/max_score)*100 AS 達成率
FROM grades
ORDER BY 達成率 DESC;
```

実行結果：

student_id	course_id	grade_type	score	max_score	達成率
311	1	レポート1	49.0	50.0	98.0
320	1	レポート1	48.5	50.0	97.0
311	1	中間テスト	95.0	100.0	95.0
…	…	…	…	…	…

例8：関数を使った並び替え

文字列関数を使って並び替えることもできます。例えば、月名で並べることを考えましょう：

```
SELECT schedule_date, MONTH(schedule_date) AS month
FROM course_schedule
ORDER BY MONTH(schedule_date);
```

実行結果：

schedule_date	month
2025-04-07	4
2025-04-08	4
…	…
2025-05-01	5

schedule_date	month
2025-05-02	5
…	…
2025-06-01	6
…	…

CASE式を使った条件付き並び替え

さらに高度な並び替えとして、CASE式を使って条件に応じた並び順を定義することもできます。

例9：CASE式を使った並び替え

例えば、出席（attendance）テーブルから、出席状況（status）を「欠席→遅刻→出席」の順に優先して表示するには：

```
SELECT * FROM attendance
ORDER BY CASE
           WHEN status = 'absent' THEN 1
           WHEN status = 'late' THEN 2
           WHEN status = 'present' THEN 3
           ELSE 4
         END;
```

実行結果：

schedule_id	student_id	status	comment
1	303	absent	事前連絡あり
1	317	absent	体調不良
…	…	…	…
1	302	late	15分遅刻
1	308	late	5分遅刻
…	…	…	…
1	301	present	NULL
1	306	present	NULL
…	…	…	…

練習問題

問題7-1

students（学生）テーブルから、すべての学生情報を学生名（student_name）の降順（逆五十音順）で取得するSQLを書いてください。

問題7-2

grades（成績）テーブルから、得点（score）が85点以上の成績を得点の高い順に取得するSQLを書いてください。

問題7-3

course_schedule（授業カレンダー）テーブルから、2025年5月の授業スケジュールを日付（schedule_date）の昇順で取得するSQLを書いてください。

問題7-4

teachers（教師）テーブルから、教師IDと名前を取得し、名前（teacher_name）の五十音順で並べるSQLを書いてください。

問題7-5

grades（成績）テーブルから、講座ID（course_id）ごとに、成績を評価タイプ（grade_type）の五十音順に、同じ評価タイプ内では得点（score）の高い順に並べて取得するSQLを書いてください。

問題7-6

attendance（出席）テーブルから、すべての出席情報を出席状況（status）が「absent」「late」「present」の順番で、同じ状態内ではコメント（comment）の有無（NULLが後）で並べて取得するSQLを書いてください。

解答

解答7-1

```sql
SELECT * FROM students ORDER BY student_name DESC;
```

解答7-2

```sql
SELECT * FROM grades WHERE score >= 85 ORDER BY score DESC;
```

解答7-3

```sql
SELECT * FROM course_schedule
WHERE schedule_date BETWEEN '2025-05-01' AND '2025-05-31'
ORDER BY schedule_date;
```

解答7-4

```
SELECT teacher_id, teacher_name FROM teachers ORDER BY teacher_name;
```

解答7-5

```
SELECT * FROM grades
ORDER BY course_id, grade_type, score DESC;
```

解答7-6

```
SELECT * FROM attendance
ORDER BY
  CASE
    WHEN status = 'absent' THEN 1
    WHEN status = 'late' THEN 2
    WHEN status = 'present' THEN 3
    ELSE 4
  END,
  CASE
    WHEN comment IS NULL THEN 2
    ELSE 1
  END;
```

まとめ

この章では、クエリ結果を特定の順序で並べるための「ORDER BY」句について学びました：

1. **基本的な並び替え**：指定したカラムの値に基づいて結果を並べる方法
2. **昇順と降順**：ASC（昇順）とDESC（降順）の指定方法
3. **複数カラムでの並び替え**：優先順位の高いカラムから順に指定する方法
4. **NULL値の扱い**：NULL値が並び替えでどのように扱われるか
5. **カラム番号**：カラム名の代わりに位置で指定する方法（あまり推奨されない）
6. **式や関数**：計算結果に基づいて並べる方法
7. **CASE式**：条件付きの複雑な並び替え

ORDER BY句は、データを見やすく整理するために非常に重要です。特に大量のデータを扱う場合、適切な並び順はデータの理解を大きく助けます。

次の章では、取得する結果の件数を制限する「LIMIT句：結果件数の制限とページネーション」について学びます。

---

8. LIMIT句：結果件数の制限とページネーション

はじめに

これまでの章では、条件に合うデータを取得し、それを特定の順序で並べる方法を学びました。しかし実際のアプリケーションでは、大量のデータがあるときに、その一部だけを表示したいことがよくあります。例えば：

- 成績上位10件だけを表示したい
- Webページで一度に20件ずつ表示したい（ページネーション）
- 最新の5件のお知らせだけを取得したい

このような「結果の件数を制限する」ためのSQLコマンドが「LIMIT句」です。この章では、クエリ結果の件数を制限する方法と、ページネーションの実装方法を学びます。

LIMIT句の基本

LIMIT句は、SELECT文の結果から指定した件数だけを取得するために使います。

> **用語解説**：
>
> - **LIMIT**：「制限する」という意味のSQLコマンドで、取得する行数を制限します。

基本構文（MySQL/MariaDB）

MySQLやMariaDBでのLIMIT句の基本構文は次のとおりです：

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT 件数;
```

LIMIT句は通常、SELECT文の最後に記述します（ORDER BYの後）。

例1：単純なLIMIT

例えば、学生（students）テーブルから最初の5人だけを取得するには：

```
SELECT * FROM students LIMIT 5;
```

実行結果：

student_id	student_name
301	黒沢春馬
302	新垣愛留
303	柴崎春花
304	森下風凛
305	河口菜恵子

ORDER BYとLIMITの組み合わせ

通常、LIMIT句はORDER BY句と組み合わせて使用します。これにより、「上位N件」「最新N件」などの操作が可能になります。

例2：ORDER BYとLIMITの組み合わせ

例えば、成績（grades）テーブルから得点（score）の高い順に上位3件を取得するには：

```
SELECT * FROM grades ORDER BY score DESC LIMIT 3;
```

実行結果：

student_id	course_id	grade_type	score	max_score	submission_date
311	1	中間テスト	95.0	100.0	2025-05-20
320	1	中間テスト	93.5	100.0	2025-05-20
302	1	中間テスト	92.0	100.0	2025-05-20

例3：最新のレコードを取得

日付でソートして最新のデータを取得することもよくあります。例えば、最新の3つの授業スケジュールを取得するには：

```
SELECT * FROM course_schedule
ORDER BY schedule_date DESC LIMIT 3;
```

実行結果：

schedule_id	course_id	schedule_date	period_id	classroom_id	teacher_id	status
95	28	2026-12-21	3	202D	119	scheduled
94	1	2026-12-21	1	102B	101	scheduled
93	14	2026-12-15	4	202D	110	scheduled

OFFSETとページネーション

Webアプリケーションなどでは、大量のデータを「ページ」に分けて表示することがよくあります（ページネーション）。この機能を実現するためには、「OFFSET」（オフセット）という機能が必要です。

> **用語解説**：
>
> - **OFFSET**：「ずらす」という意味で、結果セットの先頭から指定した数だけ行をスキップします。
> - **ページネーション**：大量のデータを複数のページに分割して表示する技術です。

基本構文（MySQL/MariaDB）

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT 件数 OFFSET スキップ数;
```

または、短縮形として：

```
SELECT カラム名 FROM テーブル名 [WHERE 条件] [ORDER BY 並び順] LIMIT スキップ数, 件数;
```

例4：OFFSETを使ったスキップ

例えば、学生（students）テーブルから6番目から10番目までの学生を取得するには：

```
SELECT * FROM students LIMIT 5 OFFSET 5;
```

または：

```
SELECT * FROM students LIMIT 5, 5;
```

実行結果：

student_id	student_name
306	河田咲奈
307	織田柚夏
308	永田悦子
309	相沢吉夫
310	吉川伽羅

例5：ページネーションの実装

ページネーションを実装する場合、通常は以下の式を使ってOFFSETを計算します：

```
OFFSET = (ページ番号 - 1) × ページあたりの件数
```

例えば、1ページあたり10件表示で、3ページ目のデータを取得するには：

```
SELECT * FROM students ORDER BY student_id LIMIT 10 OFFSET 20;
```

または：

```
SELECT * FROM students ORDER BY student_id LIMIT 20, 10;
```

実行結果：

student_id	student_name
321	井上竜也
322	木村結衣
323	林正義
324	清水香織
325	山田翔太
326	葉山陽太
327	青山凛
328	沢村大和
329	白石優月
330	月岡星奈

LIMIT句を使用する際の注意点

1. ORDER BYの重要性

LIMIT句を使用する場合、通常はORDER BY句も一緒に使うべきです。ORDER BYがなければ、どのレコードが取得されるかは保証されません。

```
-- 良い例：結果が予測可能
SELECT * FROM students ORDER BY student_id LIMIT 5;

-- 悪い例：結果が不確定
SELECT * FROM students LIMIT 5;
```

2. パフォーマンスへの影響

大規模なテーブルで大きなOFFSET値を使用すると、パフォーマンスが低下する可能性があります。これは、データベースがOFFSET分のレコードを読み込んでから破棄する必要があるためです。

3. データベース製品による構文の違い

LIMIT句の構文はデータベース製品によって異なります：

- **MySQL/MariaDB/SQLite**：`LIMIT 件数 OFFSET スキップ数` または `LIMIT スキップ数, 件数`
- **PostgreSQL**：`LIMIT 件数 OFFSET スキップ数`
- **Oracle**：`OFFSET スキップ数 ROWS FETCH NEXT 件数 ROWS ONLY`
- **SQL Server**：`OFFSET スキップ数 ROWS FETCH NEXT 件数 ROWS ONLY` または旧バージョンでは `TOP`句

この章では主にMySQL/MariaDBの構文を使用します。

実践的なページネーションの実装

実際のアプリケーションでページネーションを実装する場合、以下のようなコードになります（疑似コード）：

```
ページ番号 = URLから取得またはデフォルト値（例：1）
1ページあたりの件数 = 設定値（例：10）
総レコード数 = SELECTで取得（COUNT(*)を使用）
総ページ数 = CEILING(総レコード数 ÷ 1ページあたりの件数)
OFFSET = (ページ番号 - 1) × 1ページあたりの件数

SQLクエリ = "SELECT * FROM テーブル ORDER BY カラム LIMIT " + 1ページあたりの件数 + " OFFSET
" + OFFSET
```

例6：総レコード数と総ページ数の取得

総レコード数を取得するには：

```sql
SELECT COUNT(*) AS total_records FROM students;
```

実行結果：

total_records
100

この場合、1ページあたり10件表示なら、総ページ数は10ページ（CEILING(100 ÷ 10)）になります。

練習問題

問題8-1

grades（成績）テーブルから、得点（score）の高い順に上位5件の成績レコードを取得するSQLを書いてください。

問題8-2

course_schedule（授業カレンダー）テーブルから、日付（schedule_date）の新しい順に3件のスケジュールを取得するSQLを書いてください。

問題8-3

students（学生）テーブルを学生ID（student_id）の昇順で並べ、11番目から15番目までの学生（5件）を取得するSQLを書いてください。

問題8-4

teachers（教師）テーブルから、教師名（teacher_name）の五十音順で6番目から10番目までの教師情報を取得するSQLを書いてください。

問題8-5

1ページあたり20件表示で、grades（成績）テーブルの3ページ目のデータを得点（score）の高い順に取得するSQLを書いてください。

問題8-6

course_schedule（授業カレンダー）テーブルから、状態（status）が「scheduled」のスケジュールを日付（schedule_date）の昇順で並べ、先頭から10件スキップして次の5件を取得するSQLを書いてください。

解答

解答8-1

```
SELECT * FROM grades ORDER BY score DESC LIMIT 5;
```

解答8-2

```
SELECT * FROM course_schedule ORDER BY schedule_date DESC LIMIT 3;
```

解答8-3

```
SELECT * FROM students ORDER BY student_id LIMIT 5 OFFSET 10;
```

または

```
SELECT * FROM students ORDER BY student_id LIMIT 10, 5;
```

解答8-4

```
SELECT * FROM teachers ORDER BY teacher_name LIMIT 5 OFFSET 5;
```

または

```
SELECT * FROM teachers ORDER BY teacher_name LIMIT 5, 5;
```

解答8-5

```
SELECT * FROM grades ORDER BY score DESC LIMIT 20 OFFSET 40;
```

または

```
SELECT * FROM grades ORDER BY score DESC LIMIT 40, 20;
```

解答8-6

```
SELECT * FROM course_schedule
WHERE status = 'scheduled'
ORDER BY schedule_date
LIMIT 5 OFFSET 10;
```

または

```
SELECT * FROM course_schedule
WHERE status = 'scheduled'
ORDER BY schedule_date
LIMIT 10, 5;
```

まとめ

この章では、クエリ結果の件数を制限するためのLIMIT句と、ページネーションの実装方法について学びました：

1. **LIMIT句の基本**：指定した件数だけのレコードを取得する方法
2. **ORDER BYとの組み合わせ**：順序付けされた結果から一部だけを取得する方法
3. **OFFSET**：結果の先頭から指定した数だけレコードをスキップする方法
4. **ページネーション**：大量のデータを複数のページに分けて表示する実装方法
5. **注意点**：LIMIT句を使用する際の留意事項
6. **データベース製品による違い**：異なるデータベースでの構文の違い

LIMIT句は特にWebアプリケーションの開発で重要な機能です。大量のデータを効率よく表示するためのページネーション機能を実装するために欠かせません。また、トップN（上位N件）やボトムN（下位N件）のデータを取得する際にも使われます。

次の章では、データの集計分析を行うための「集計関数：COUNT、SUM、AVG、MAX、MIN」について学びます。