==========

10-1. パフォーマンス最適化：効率的なクエリの書き方

はじめに

これまでの章でSQLの基本的な書き方を学びました。この章では、実際の業務でデータベースを効率よく使うための「パフォーマンス最適化」について学びます。

データベースが大きくなったり、ユーザーが増えたりすると、SQLクエリの実行速度が遅くなることがあります。このような問題を解決するために、効率的なクエリの書き方を身につけることが重要です。

> **用語解説**：
>
> - **パフォーマンス最適化**：プログラムやシステムの動作速度を向上させるための改善作業です。
> - **クエリ**：データベースに対する問い合わせのことで、SELECT文などのSQL文を指します。
> - **実行速度**：SQLが処理される速さのことで、通常は秒数やミリ秒で測定されます。

ケーススタディ1：インデックスを活用した検索の高速化

問題状況

学校データベースで「特定の教師が担当する講座」を検索するクエリが遅いという問題が発生しました。

```
-- 遅いクエリの例
SELECT * FROM courses WHERE teacher_id = 105;
```

このクエリは、教師テーブルに10,000件のデータがある場合、すべてのレコードを順番に確認する必要があり、時間がかかります。

解決方法：インデックスの活用

インデックスとは、データベースの検索を高速化するための仕組みです。本の索引のように、データの場所を素早く見つけることができます。

> **用語解説**：
>
> - **インデックス**：データベースのテーブルに対して作成される「検索の目印」です。本の索引と同じように、データを素早く見つけるために使われます。

```
-- teacher_idにインデックスを作成
CREATE INDEX idx_teacher_id ON courses(teacher_id);
```

```
-- これで同じクエリが高速化される
SELECT * FROM courses WHERE teacher_id = 105;
```

効果：

- 検索時間が大幅に短縮される
- 大量のデータがあっても検索速度が維持される

注意点：

- インデックスはデータの更新時に時間がかかる場合がある
- ストレージ容量を多く使用する

実践例：複合インデックス

複数のカラムを組み合わせた検索でも、インデックスを活用できます。

```
-- 日付と時限の組み合わせでよく検索する場合
CREATE INDEX idx_date_period ON course_schedule(schedule_date, period_id);

-- このクエリが高速化される
SELECT * FROM course_schedule
WHERE schedule_date = '2025-05-20' AND period_id = 1;
```

ケーススタディ2：適切なJOINの使用

問題状況

学生の成績情報と講座名を一緒に表示したいが、以下のような非効率なクエリを使用していました。

```
-- 非効率なクエリ例（サブクエリの多用）
SELECT student_id, score,
       (SELECT course_name FROM courses WHERE course_id = grades.course_id) AS
course_name
FROM grades
WHERE score >= 80;
```

このクエリは、成績レコードの数だけサブクエリが実行されるため、非常に遅くなります。

> **用語解説**：
>
> - **サブクエリ**：SQL文の中に含まれる別の完全なSQL文のことです。
> - **JOIN**：複数のテーブルを結合してデータを取得する方法です。

解決方法：JOINの使用

```
-- 効率的なクエリ（JOINを使用）
SELECT g.student_id, g.score, c.course_name
FROM grades g
JOIN courses c ON g.course_id = c.course_id
WHERE g.score >= 80;
```

効果：

- サブクエリの繰り返し実行が避けられる
- データベースエンジンが最適化された結合処理を行う
- 大幅な速度向上が期待できる

実践例：適切なJOINの選択

```
-- 内部結合（INNER JOIN）- 両方のテーブルにデータがある場合のみ
SELECT s.student_name, g.score, c.course_name
FROM students s
INNER JOIN student_courses sc ON s.student_id = sc.student_id
INNER JOIN courses c ON sc.course_id = c.course_id
INNER JOIN grades g ON s.student_id = g.student_id AND c.course_id = g.course_id;

-- 左外部結合（LEFT JOIN）- 学生情報は全て表示、成績は有無を問わない
SELECT s.student_name, g.score
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id;
```

ケーススタディ3：WHERE句の最適化

問題状況

以下のようなクエリで、条件の順序が効率的でない場合があります。

```
-- 最適化前：関数を使った条件
SELECT * FROM course_schedule
WHERE YEAR(schedule_date) = 2025 AND teacher_id = 101;
```

この例では、`YEAR(schedule_date)`という関数を使っているため、インデックスが使用されません。

解決方法：条件の書き換え

```
-- 最適化後：範囲指定を使用
SELECT * FROM course_schedule
WHERE schedule_date >= '2025-01-01'
  AND schedule_date < '2026-01-01'
  AND teacher_id = 101;
```

改善のポイント：

- 関数を使わずに範囲指定を使用
- インデックスが活用できるようになる
- より限定的な条件（teacher_id）を先に書く

実践例：効率的な条件の書き方

```
-- 良い例：限定的な条件を先に書く
SELECT * FROM grades
WHERE course_id = '1'         -- 限定的（特定の講座）
  AND score >= 90             -- より広範囲（高得点）
  AND grade_type = '中間テスト';

-- 悪い例：LIKE演算子の前方一致以外
SELECT * FROM students
WHERE student_name LIKE '%田%';  -- 中間一致は遅い

-- 良い例：前方一致を使用
SELECT * FROM students
WHERE student_name LIKE '田%';   -- 前方一致は比較的高速
```

ケーススタディ4：LIMIT句の効果的な使用

問題状況

管理画面で成績上位者を表示したいが、全データを取得してから並べ替えていました。

```
-- 非効率な例：全データを取得
SELECT * FROM grades ORDER BY score DESC;
-- アプリケーション側で上位10件を表示
```

解決方法：LIMIT句の使用

```
-- 効率的な例：必要な分だけ取得
SELECT student_id, course_id, score
FROM grades
ORDER BY score DESC
LIMIT 10;
```

効果：

- ネットワーク転送量の削減
- メモリ使用量の削減
- 応答速度の向上

実践例：ページネーション

```
-- 1ページ目（1-10位）
SELECT student_id, course_id, score
FROM grades
ORDER BY score DESC
LIMIT 10 OFFSET 0;

-- 2ページ目（11-20位）
SELECT student_id, course_id, score
FROM grades
ORDER BY score DESC
LIMIT 10 OFFSET 10;
```

ケーススタディ5：集計処理の最適化

問題状況

各講座の受講者数を計算するのに時間がかかっています。

```
-- 非効率な例：サブクエリで個別計算
SELECT course_id, course_name,
       (SELECT COUNT(*) FROM student_courses sc WHERE sc.course_id = c.course_id)
AS student_count
FROM courses c;
```

解決方法：GROUP BYの使用

```
-- 効率的な例：GROUP BYで一括集計
SELECT c.course_id, c.course_name, COUNT(sc.student_id) AS student_count
FROM courses c
LEFT JOIN student_courses sc ON c.course_id = sc.course_id
GROUP BY c.course_id, c.course_name;
```

効果：

- サブクエリの繰り返し実行を回避
- データベースエンジンの集計機能を活用
- 大幅な性能向上

実践例：条件付き集計

```
-- 各講座の成績分布を効率的に取得
SELECT course_id,
       COUNT(*) AS total_grades,
```

```
        COUNT(CASE WHEN score >= 90 THEN 1 END) AS excellent_count,
        COUNT(CASE WHEN score >= 80 THEN 1 END) AS good_count,
        AVG(score) AS average_score
FROM grades
GROUP BY course_id;
```

ケーススタディ6：EXISTSとINの使い分け

問題状況

「成績データがある学生」を取得したいが、どちらの書き方が効率的か分からない。

```
-- パターン1：IN句を使用
SELECT * FROM students
WHERE student_id IN (SELECT student_id FROM grades);

-- パターン2：EXISTS句を使用
SELECT * FROM students s
WHERE EXISTS (SELECT 1 FROM grades g WHERE g.student_id = s.student_id);
```

解決方法：適切な使い分け

INを使う場合：

- サブクエリの結果が少ない場合
- サブクエリにNULL値が含まれない場合

EXISTSを使う場合：

- サブクエリの結果が多い場合
- サブクエリの結果の存在だけを確認したい場合

```
-- 推奨：EXISTSを使用（通常はこちらが効率的）
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 >= 80
);
```

パフォーマンス測定のヒント

実行計画の確認

```

```

```
-- クエリの実行計画を確認（MySQL）
EXPLAIN SELECT * FROM courses WHERE teacher_id = 105;
```

実行計画では以下の項目を確認します：

- **type**: 結合タイプ（ALL, index, range, ref等）
- **key**: 使用されているインデックス
- **rows**: 処理対象の推定行数

> **用語解説**：
>
> - **実行計画**：データベースがクエリをどのように処理するかの計画書です。処理速度の改善点を見つけるために使います。

実行時間の測定

```
-- 実行時間を測定する設定（MySQL）
SET profiling = 1;

-- クエリを実行
SELECT * FROM courses WHERE teacher_id = 105;

-- 実行時間を確認
SHOW PROFILES;
```

まとめ：効率的なクエリを書くための原則

1. **インデックスを活用する**
   - 検索条件によく使われるカラムにインデックスを作成
   - WHERE句の条件がインデックスを使えるように書く
2. **適切なJOINを使用する**
   - サブクエリよりもJOINを優先する
   - 必要なデータだけを結合する
3. **条件を効率的に書く**
   - 限定的な条件を先に書く
   - 関数の使用を避ける
   - 前方一致のLIKEを使用する
4. **必要なデータだけを取得する**
   - SELECT *を避けて、必要なカラムだけを指定
   - LIMIT句を適切に使用する
5. **集計処理を最適化する**

- GROUP BYを活用する
- サブクエリの繰り返し実行を避ける

これらの原則を意識することで、データベースの性能を大幅に改善することができます。実際の業務では、データ量やアクセスパターンに応じて最適な手法を選択することが重要です。

10-2. テーブル設計テクニック：実践的なデータベース設計

はじめに

データベースの設計は、システム全体の性能や保守性に大きく影響します。この章では、実際のプロジェクトで役立つテーブル設計のテクニックを、具体的なケーススタディを通じて学びます。

良いデータベース設計は、データの整合性を保ち、検索速度を向上させ、将来の変更に対応しやすくします。

> **用語解説**：
>
> - **テーブル設計**：データベースにおけるテーブルの構造（カラム、データ型、制約など）を決める作業です。
> - **データの整合性**：データに矛盾や重複がなく、正しい状態を保つことです。
> - **保守性**：システムの修正や機能追加がしやすいかどうかの度合いです。

ケーススタディ1：正規化とパフォーマンスのバランス

問題状況：過度な正規化による性能問題

学生管理システムで、以下のような完全に正規化されたテーブル設計がありました。

```
-- 過度に正規化されたテーブル例
CREATE TABLE students (
    student_id BIGINT PRIMARY KEY,
    student_name VARCHAR(64),
    prefecture_id INT,
    city_id INT,
    ward_id INT
);

CREATE TABLE prefectures (
    prefecture_id INT PRIMARY KEY,
    prefecture_name VARCHAR(20)
);

CREATE TABLE cities (
    city_id INT PRIMARY KEY,
    city_name VARCHAR(50),
    prefecture_id INT
);
```

```
CREATE TABLE wards (
    ward_id INT PRIMARY KEY,
    ward_name VARCHAR(50),
    city_id INT
);
```

この設計では、学生の住所を表示するために4つのテーブルを結合する必要があり、頻繁にアクセスされる一覧画面が遅くなりました。

> **用語解説**：
>
> - **正規化**：データの重複を排除し、データベースの構造を整理する手法です。
> - **結合（JOIN）**：複数のテーブルからデータを組み合わせて取得する処理です。

解決方法：実用的な非正規化

住所情報のように、あまり変更されないデータは、適度に非正規化することで性能を向上させることができます。

```
-- 実用的なバランスを考慮した設計
CREATE TABLE students (
    student_id BIGINT PRIMARY KEY,
    student_name VARCHAR(64),
    prefecture_name VARCHAR(20),
    city_name VARCHAR(50),
    ward_name VARCHAR(50),
    full_address VARCHAR(200),  -- 検索用の完全住所
    postal_code VARCHAR(8),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

メリット：

- 学生一覧の表示が高速化
- 住所検索が簡単になる
- アプリケーションコードがシンプルになる

注意点：

- データの更新時に複数のカラムを同時に変更する必要がある
- ストレージ容量が多少増加する

実践例：履歴テーブルの設計

住所変更の履歴を管理したい場合は、履歴テーブルを別途作成します。

```
```

```sql
-- 住所変更履歴テーブル
CREATE TABLE student_address_history (
    history_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    student_id BIGINT,
    old_prefecture VARCHAR(20),
    old_city VARCHAR(50),
    old_ward VARCHAR(50),
    new_prefecture VARCHAR(20),
    new_city VARCHAR(50),
    new_ward VARCHAR(50),
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    changed_by BIGINT,  -- 変更者ID
    FOREIGN KEY (student_id) REFERENCES students(student_id)
);
```

ケーススタディ2：階層データの効率的な管理

問題状況：組織階層の管理

学校の組織構造（学部→学科→専攻）を管理する必要がありました。

```sql
-- 単純な階層テーブル（問題のある設計）
CREATE TABLE organizations (
    org_id INT PRIMARY KEY,
    org_name VARCHAR(100),
    parent_id INT,
    level INT  -- 1:学部, 2:学科, 3:専攻
);
```

この設計では、特定の組織から上位組織への辿りや、全ての子組織の取得が複雑になります。

解決方法：パス情報の追加

階層の深さに関係なく効率的にデータを取得できるよう、パス情報を追加します。

```sql
-- 改良された階層テーブル
CREATE TABLE organizations (
    org_id INT PRIMARY KEY,
    org_name VARCHAR(100),
    parent_id INT,
    level INT,
    org_path VARCHAR(500),  -- '/1/3/15' のような形式
    org_path_names VARCHAR(500),  -- '工学部/情報学科/AI専攻' のような形式
    display_order INT,  -- 表示順序
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (parent_id) REFERENCES organizations(org_id)
);
```

```
-- パス検索用のインデックス
CREATE INDEX idx_org_path ON organizations(org_path);
CREATE INDEX idx_level_order ON organizations(level, display_order);
```

検索の例：

```
-- 特定組織の全ての子組織を取得
SELECT * FROM organizations
WHERE org_path LIKE '/1/3/%'
ORDER BY level, display_order;

-- 特定レベルの組織一覧を取得
SELECT * FROM organizations
WHERE level = 2 AND is_active = TRUE
ORDER BY display_order;

-- 組織の上位階層を表示
SELECT org_path_names FROM organizations WHERE org_id = 15;
-- 結果: '工学部/情報学科/AI専攻'
```

実践例：組織変更の管理

組織改編に対応するため、有効期間を管理するテーブルも作成します。

```
-- 組織の有効期間管理
CREATE TABLE organization_periods (
    period_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    org_id INT,
    valid_from DATE,
    valid_to DATE,
    reason VARCHAR(200),  -- 変更理由
    FOREIGN KEY (org_id) REFERENCES organizations(org_id)
);
```

ケーススタディ3：柔軟な属性管理（EAVパターン）

問題状況：多様な学生属性の管理

学生には様々な属性（出身校、趣味、特技、保有資格など）があり、学生によって持つ属性が異なります。

```
-- 硬直的な設計例（避けるべき）
CREATE TABLE students (
    student_id BIGINT PRIMARY KEY,
    student_name VARCHAR(64),
    high_school VARCHAR(100),
    hobby1 VARCHAR(50),
```

```
    hobby2 VARCHAR(50),
    hobby3 VARCHAR(50),
    skill1 VARCHAR(50),
    skill2 VARCHAR(50),
    -- ...さらに多くのカラムが必要
);
```

この設計では、新しい属性の追加のたびにテーブル構造を変更する必要があります。

> **用語解説**：
>
> - **EAVパターン**：Entity（実体）、Attribute（属性）、Value（値）の3つの要素でデータを管理する設計パターンです。

解決方法：EAVパターンの適用

```
-- メインの学生テーブル
CREATE TABLE students (
    student_id BIGINT PRIMARY KEY,
    student_name VARCHAR(64),
    email VARCHAR(200),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 属性定義テーブル
CREATE TABLE attribute_definitions (
    attribute_id INT PRIMARY KEY,
    attribute_name VARCHAR(50),
    attribute_type ENUM('text', 'number', 'date', 'boolean'),
    display_name VARCHAR(100),
    is_required BOOLEAN DEFAULT FALSE,
    display_order INT
);

-- 学生属性値テーブル（EAV）
CREATE TABLE student_attributes (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    student_id BIGINT,
    attribute_id INT,
    text_value VARCHAR(500),
    number_value DECIMAL(10,2),
    date_value DATE,
    boolean_value BOOLEAN,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (attribute_id) REFERENCES attribute_definitions(attribute_id),
    UNIQUE KEY uk_student_attribute (student_id, attribute_id)
);

-- 検索用インデックス
```

```
CREATE INDEX idx_student_attr ON student_attributes(student_id, attribute_id);
CREATE INDEX idx_text_value ON student_attributes(text_value);
```

データの例：

```
-- 属性定義の登録
INSERT INTO attribute_definitions VALUES
(1, 'high_school', 'text', '出身高校', FALSE, 1),
(2, 'hobby', 'text', '趣味', FALSE, 2),
(3, 'toeic_score', 'number', 'TOEIC点数', FALSE, 3),
(4, 'graduation_date', 'date', '卒業予定日', FALSE, 4);

-- 学生属性の登録
INSERT INTO student_attributes (student_id, attribute_id, text_value) VALUES
(301, 1, '東京高等学校'),
(301, 2, 'プログラミング');

INSERT INTO student_attributes (student_id, attribute_id, number_value) VALUES
(301, 3, 850);
```

検索の例：

```
-- 特定学生の全属性を取得
SELECT ad.display_name,
       COALESCE(sa.text_value,
                CAST(sa.number_value AS CHAR),
                CAST(sa.date_value AS CHAR),
                CASE sa.boolean_value WHEN TRUE THEN '有' ELSE '無' END) AS value
FROM student_attributes sa
JOIN attribute_definitions ad ON sa.attribute_id = ad.attribute_id
WHERE sa.student_id = 301
ORDER BY ad.display_order;

-- TOEIC点数が800点以上の学生を検索
SELECT s.student_id, s.student_name
FROM students s
JOIN student_attributes sa ON s.student_id = sa.student_id
WHERE sa.attribute_id = 3 AND sa.number_value >= 800;
```

ケーススタディ4：ログテーブルの効率的な設計

問題状況：システムログの管理

Webアプリケーションのアクセスログやエラーログを効率的に管理する必要がありました。

解決方法：パーティショニングとログレベル管理

```
```

```
-- ログテーブルの設計
CREATE TABLE system_logs (
    log_id BIGINT AUTO_INCREMENT,
    log_date DATE,
    log_time TIME,
    log_level ENUM('DEBUG', 'INFO', 'WARN', 'ERROR', 'FATAL'),
    user_id BIGINT,
    action VARCHAR(100),
    target_table VARCHAR(50),
    target_id BIGINT,
    ip_address VARCHAR(45),  -- IPv6対応
    user_agent TEXT,
    request_data JSON,
    response_time_ms INT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (log_id, log_date)  -- パーティション用
) PARTITION BY RANGE (TO_DAYS(log_date)) (
    PARTITION p_202501 VALUES LESS THAN (TO_DAYS('2025-02-01')),
    PARTITION p_202502 VALUES LESS THAN (TO_DAYS('2025-03-01')),
    PARTITION p_202503 VALUES LESS THAN (TO_DAYS('2025-04-01')),
    -- 月ごとにパーティションを作成
    PARTITION p_future VALUES LESS THAN MAXVALUE
);

-- 効率的な検索用インデックス
CREATE INDEX idx_user_action ON system_logs(user_id, action, log_date);
CREATE INDEX idx_log_level_date ON system_logs(log_level, log_date);
CREATE INDEX idx_target ON system_logs(target_table, target_id, log_date);
```

> **用語解説**：
>
> - **パーティショニング**：大きなテーブルを複数の小さな部分に分割して管理する手法です。検索速度の向上や管理の効率化が図れます。

パーティション管理の自動化：

```
-- 新しい月のパーティションを追加
ALTER TABLE system_logs ADD PARTITION (
    PARTITION p_202504 VALUES LESS THAN (TO_DAYS('2025-05-01'))
);

-- 古いパーティションを削除（3ヶ月前のデータを削除）
ALTER TABLE system_logs DROP PARTITION p_202501;
```

実践例：集計テーブルの作成

ログデータから定期的に集計データを作成し、レポート作成を高速化します。

```sql
-- 日次集計テーブル
CREATE TABLE daily_log_summary (
    summary_date DATE PRIMARY KEY,
    total_requests INT,
    total_users INT,
    error_count INT,
    avg_response_time_ms DECIMAL(8,2),
    top_action VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 集計処理の例
INSERT INTO daily_log_summary (summary_date, total_requests, total_users,
error_count, avg_response_time_ms, top_action)
SELECT
    log_date,
    COUNT(*) as total_requests,
    COUNT(DISTINCT user_id) as total_users,
    SUM(CASE WHEN log_level IN ('ERROR', 'FATAL') THEN 1 ELSE 0 END) as
error_count,
    AVG(response_time_ms) as avg_response_time_ms,
    (SELECT action FROM system_logs sl2
     WHERE sl2.log_date = sl.log_date
     GROUP BY action
     ORDER BY COUNT(*) DESC
     LIMIT 1) as top_action
FROM system_logs sl
WHERE log_date = CURDATE() - INTERVAL 1 DAY
GROUP BY log_date;
```

ケーススタディ5：多言語対応テーブルの設計

問題状況：国際化への対応

学校システムを多言語（日本語、英語、中国語）に対応させる必要がありました。

解決方法：多言語テーブルの設計

```sql
-- 言語マスターテーブル
CREATE TABLE languages (
    language_code VARCHAR(5) PRIMARY KEY,  -- 'ja', 'en', 'zh-CN'
    language_name VARCHAR(50),
    display_order INT,
    is_active BOOLEAN DEFAULT TRUE
);

-- メイン講座テーブル（言語に依存しないデータ）
CREATE TABLE courses (
    course_id VARCHAR(16) PRIMARY KEY,
    teacher_id BIGINT,
```

```
    capacity INT,
    credit_hours INT,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id)
);

-- 講座多言語テーブル
CREATE TABLE course_translations (
    course_id VARCHAR(16),
    language_code VARCHAR(5),
    course_name VARCHAR(128),
    course_description TEXT,
    course_objectives TEXT,
    prerequisite_note VARCHAR(500),
    PRIMARY KEY (course_id, language_code),
    FOREIGN KEY (course_id) REFERENCES courses(course_id),
    FOREIGN KEY (language_code) REFERENCES languages(language_code)
);

-- 教師多言語テーブル
CREATE TABLE teacher_translations (
    teacher_id BIGINT,
    language_code VARCHAR(5),
    teacher_name VARCHAR(64),
    title VARCHAR(50),
    biography TEXT,
    specialization VARCHAR(200),
    PRIMARY KEY (teacher_id, language_code),
    FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id),
    FOREIGN KEY (language_code) REFERENCES languages(language_code)
);
```

データの例：

```
-- 言語設定
INSERT INTO languages VALUES
('ja', '日本語', 1, TRUE),
('en', 'English', 2, TRUE),
('zh-CN', '中文（简体）', 3, TRUE);

-- 講座の多言語データ
INSERT INTO course_translations VALUES
('1', 'ja', 'ITのための基礎知識', 'コンピュータとネットワークの基本概念を学びます', '基本的なIT知識の習得', '特になし'),
('1', 'en', 'IT Fundamentals', 'Learn basic concepts of computers and networks', 'Acquire basic IT knowledge', 'None'),
('1', 'zh-CN', 'IT基础知识', '学习计算机和网络的基本概念', '掌握基本IT知识', '无');
```

多言語データの取得：

```sql
-- 特定言語での講座一覧取得
SELECT c.course_id, ct.course_name, ct.course_description, t.teacher_name
FROM courses c
JOIN course_translations ct ON c.course_id = ct.course_id
LEFT JOIN teacher_translations tt ON c.teacher_id = tt.teacher_id AND
ct.language_code = tt.language_code
WHERE ct.language_code = 'en' AND c.is_active = TRUE;

-- フォールバック機能付きの取得（英語がない場合は日本語を表示）
SELECT c.course_id,
       COALESCE(ct_en.course_name, ct_ja.course_name) AS course_name,
       COALESCE(ct_en.course_description, ct_ja.course_description) AS
course_description
FROM courses c
LEFT JOIN course_translations ct_en ON c.course_id = ct_en.course_id AND
ct_en.language_code = 'en'
LEFT JOIN course_translations ct_ja ON c.course_id = ct_ja.course_id AND
ct_ja.language_code = 'ja'
WHERE c.is_active = TRUE;
```

ケーススタディ6：バージョン管理機能の実装

問題状況：データの変更履歴管理

カリキュラムや講座内容の変更履歴を管理し、いつでも過去の状態に戻せるようにする必要がありました。

解決方法：バージョン管理テーブルの設計

```sql
-- カリキュラムマスターテーブル
CREATE TABLE curricula (
    curriculum_id VARCHAR(20) PRIMARY KEY,
    curriculum_code VARCHAR(50),
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- カリキュラムバージョンテーブル
CREATE TABLE curriculum_versions (
    version_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    curriculum_id VARCHAR(20),
    version_number DECIMAL(3,1),  -- 1.0, 1.1, 2.0など
    curriculum_name VARCHAR(200),
    curriculum_description TEXT,
    total_credit_hours INT,
    effective_date DATE,
    expiry_date DATE,
    status ENUM('draft', 'approved', 'active', 'deprecated'),
    created_by BIGINT,
    approved_by BIGINT,
    approved_at TIMESTAMP NULL,
```

```
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (curriculum_id) REFERENCES curricula(curriculum_id),
    INDEX idx_curriculum_version (curriculum_id, version_number),
    INDEX idx_effective_date (effective_date, status)
);

-- 講座とカリキュラムの関連テーブル（バージョン対応）
CREATE TABLE curriculum_course_relations (
    relation_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    version_id BIGINT,
    course_id VARCHAR(16),
    required_type ENUM('required', 'elective', 'optional'),
    semester INT,
    display_order INT,
    FOREIGN KEY (version_id) REFERENCES curriculum_versions(version_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id),
    UNIQUE KEY uk_version_course (version_id, course_id)
);
```

バージョン管理の実践例：

```
-- 現在有効なカリキュラムを取得
SELECT cv.*, c.curriculum_code
FROM curriculum_versions cv
JOIN curricula c ON cv.curriculum_id = c.curriculum_id
WHERE cv.status = 'active'
  AND cv.effective_date <= CURDATE()
  AND (cv.expiry_date IS NULL OR cv.expiry_date > CURDATE());

-- 新しいバージョンの作成（既存バージョンからコピー）
INSERT INTO curriculum_versions (curriculum_id, version_number, curriculum_name,
curriculum_description, total_credit_hours, effective_date, status, created_by)
SELECT curriculum_id, version_number + 0.1, curriculum_name,
curriculum_description, total_credit_hours, '2025-04-01', 'draft', 1
FROM curriculum_versions
WHERE curriculum_id = 'CURR001' AND status = 'active';

-- 関連講座もコピー
INSERT INTO curriculum_course_relations (version_id, course_id, required_type,
semester, display_order)
SELECT @new_version_id, course_id, required_type, semester, display_order
FROM curriculum_course_relations
WHERE version_id = @old_version_id;
```

ケーススタディ7：大量データ対応のテーブル設計

問題状況：アクセスログの急激な増加

Webアプリケーションの利用者が増え、1日に数百万件のアクセスログが発生するようになりました。

解決方法：時系列データベース設計

```
-- 年月別のパーティション戦略
CREATE TABLE access_logs (
    log_id BIGINT AUTO_INCREMENT,
    access_date DATE,
    access_time TIME(3),  -- ミリ秒まで記録
    user_id BIGINT,
    session_id VARCHAR(128),
    page_url VARCHAR(500),
    referrer_url VARCHAR(500),
    ip_address VARCHAR(45),
    country_code VARCHAR(2),
    device_type ENUM('desktop', 'mobile', 'tablet'),
    browser_name VARCHAR(50),
    response_code INT,
    response_size_bytes INT,
    response_time_ms INT,
    created_at TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP(3),
    PRIMARY KEY (log_id, access_date)
) PARTITION BY RANGE (TO_DAYS(access_date)) (
    PARTITION p_202501 VALUES LESS THAN (TO_DAYS('2025-02-01')),
    PARTITION p_202502 VALUES LESS THAN (TO_DAYS('2025-03-01')),
    PARTITION p_202503 VALUES LESS THAN (TO_DAYS('2025-04-01')),
    -- 自動でパーティションを管理
    PARTITION p_future VALUES LESS THAN MAXVALUE
);

-- 高速検索用のインデックス
CREATE INDEX idx_user_date ON access_logs(user_id, access_date);
CREATE INDEX idx_page_date ON access_logs(page_url(100), access_date);
CREATE INDEX idx_response_code ON access_logs(response_code, access_date);
```

効率的な集計テーブルの設計：

```
-- 時間別集計テーブル
CREATE TABLE hourly_access_stats (
    stat_date DATE,
    stat_hour TINYINT,
    page_url VARCHAR(500),
    total_accesses INT,
    unique_users INT,
    total_response_time_ms BIGINT,
    error_count INT,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (stat_date, stat_hour, page_url(100))
);

-- 日別集計テーブル
CREATE TABLE daily_access_stats (
```

```
    stat_date DATE PRIMARY KEY,
    total_accesses INT,
    unique_users INT,
    total_sessions INT,
    avg_response_time_ms DECIMAL(8,2),
    top_page VARCHAR(500),
    error_rate DECIMAL(5,2),
    mobile_rate DECIMAL(5,2),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

自動集計処理の例：

```
-- 時間別集計の更新
INSERT INTO hourly_access_stats (stat_date, stat_hour, page_url, total_accesses,
unique_users, total_response_time_ms, error_count)
SELECT
    access_date,
    HOUR(access_time) as stat_hour,
    page_url,
    COUNT(*) as total_accesses,
    COUNT(DISTINCT user_id) as unique_users,
    SUM(response_time_ms) as total_response_time_ms,
    SUM(CASE WHEN response_code >= 400 THEN 1 ELSE 0 END) as error_count
FROM access_logs
WHERE access_date = CURDATE() - INTERVAL 1 DAY
GROUP BY access_date, HOUR(access_time), page_url
ON DUPLICATE KEY UPDATE
    total_accesses = VALUES(total_accesses),
    unique_users = VALUES(unique_users),
    total_response_time_ms = VALUES(total_response_time_ms),
    error_count = VALUES(error_count);
```

設計チェックリスト

良いテーブル設計を行うためのチェックポイントをまとめます。

1. データの整合性

```
-- 適切な制約の設定
CREATE TABLE example_table (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(200) UNIQUE NOT NULL,
    age INT CHECK (age >= 0 AND age <= 150),
    status ENUM('active', 'inactive', 'suspended') DEFAULT 'active',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

2. パフォーマンスの考慮

```
-- 検索頻度の高いカラムにインデックス
CREATE INDEX idx_email ON users(email);
CREATE INDEX idx_created_date ON users(created_at);
CREATE INDEX idx_status_date ON users(status, created_at);
```

3. 将来の拡張性

```
-- 拡張可能な設計例
CREATE TABLE user_settings (
    user_id BIGINT,
    setting_key VARCHAR(100),
    setting_value TEXT,
    data_type ENUM('string', 'number', 'boolean', 'json'),
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, setting_key)
);
```

4. セキュリティの考慮

```
-- 個人情報の適切な管理
CREATE TABLE user_profiles (
    user_id BIGINT PRIMARY KEY,
    display_name VARCHAR(50),
    -- 機密情報は別テーブルで管理
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE user_private_info (
    user_id BIGINT PRIMARY KEY,
    real_name_encrypted BLOB,
    phone_encrypted BLOB,
    address_encrypted BLOB,
    encryption_key_id INT,
    FOREIGN KEY (user_id) REFERENCES user_profiles(user_id)
);
```

まとめ

効果的なテーブル設計のための重要なポイント：

1. **適切な正規化レベルの選択**
    - 完全な正規化と実用性のバランスを考慮
    - アクセスパターンに基づいた最適化

2. **パフォーマンスを考慮した設計**
    - 適切なインデックスの設計
    - パーティショニングの活用
    - 集計テーブルの準備
3. **拡張性と保守性**
    - 将来の要件変更に対応できる柔軟な設計
    - 多言語対応やバージョン管理の考慮
4. **データの整合性とセキュリティ**
    - 適切な制約の設定
    - 機密情報の保護
5. **運用を考慮した設計**
    - ログの効率的な管理
    - 履歴データの保持戦略

これらのテクニックを組み合わせることで、長期間にわたって安定して運用できるデータベースシステムを構築することができます。実際のプロジェクトでは、要件や制約に応じて最適な設計手法を選択することが重要です。
