

おまけ：SQLの発展的な 機能

サブクエリ、トランザクション、インデックス、そして
更なる世界へ

● このセッションの内容

今回は基本編でカバーできなかった以下の重要概念を学びます：

1. **サブクエリ** - クエリの中のクエリ
2. **トランザクション** - データの整合性を保つ仕組み
3. **インデックス** - クエリを高速化する技術
4. **ビュー** - クエリの再利用
5. **ウィンドウ関数** - 高度な分析機能
6. **CTE** - 複雑なクエリを整理する
7. **その他の重要概念** - データ型、文字列関数など

1. サブクエリ

クエリの中にクエリを書く技術

● サブクエリとは？

サブクエリ（副問い合わせ） = 別のSQL文の中に埋め込まれたSELECT文

- ✓ メインクエリの実行前に、サブクエリが先に実行される
- ✓ 複雑な条件や計算を段階的に実行できる

-- 例：平均価格より高い商品を探す

```
SELECT * FROM products
WHERE price > (SELECT AVG(price) FROM products);
```

↑これがサブクエリ

● サブクエリの種類

1. スカラサブクエリ

単一の値（1行1列）を返すサブクエリ

```
-- 最高価格の商品を探す
SELECT product_name, price
FROM 'data/products.csv'
WHERE price = (SELECT MAX(price) FROM 'data/products.csv');
```

2. 行サブクエリ

1行（複数列可）を返すサブクエリ

● WHERE句でのサブクエリ - IN 演算子

特定の値のリストに含まれるかチェック：

```
-- 購入履歴のある顧客の情報を取得
SELECT customer_id, customer_name, email
FROM 'data/customers.csv'
WHERE customer_id IN (
    SELECT DISTINCT customer_id
    FROM 'data/sales.csv'
);
```

● WHERE句でのサブクエリ - EXISTS演算子

条件を満たすデータが存在するかチェック：

```
-- 一度でも商品を購入した顧客
SELECT c.customer_id, c.customer_name
FROM 'data/customers.csv' c
WHERE EXISTS (
    SELECT 1
    FROM 'data/sales.csv' s
    WHERE s.customer_id = c.customer_id
);
```

● FROM句でのサブクエリ

サブクエリの結果を仮想テーブルとして使用：

```
-- カテゴリごとの平均価格と比較
```

```
SELECT
```

```
    p.product_name,  
    p.category,  
    p.price,  
    cat_avg.avg_price AS カテゴリ平均価格,  
    p.price - cat_avg.avg_price AS 差額
```

```
FROM 'data/products.csv' p
```

```
INNER JOIN (
```

```
    SELECT category, AVG(price) AS avg_price  
    FROM 'data/products.csv'  
    GROUP BY category
```

```
) cat_avg ON p.category = cat_avg.category
```


● 相関サブクエリ

外側のクエリの値を参照するサブクエリ：

```
-- 各顧客の最新購入日を取得
SELECT
  c.customer_name,
  (SELECT MAX(s.order_date)
   FROM 'data/sales.csv' s
   WHERE s.customer_id = c.customer_id) AS 最新購入日
FROM 'data/customers.csv' c;
```

✓ 相関サブクエリは外側のクエリの各行に対して実行される

● サブクエリの実践例

平均購入金額を超える顧客

-- 平均以上の購入金額の顧客を特定

SELECT

c.customer_name,

SUM(s.quantity * p.price) **AS** 総購入金額

FROM 'data/customers.csv' c

INNER JOIN 'data/sales.csv' s **ON** c.customer_id = s.customer_id

INNER JOIN 'data/products.csv' p **ON** s.product_id = p.product_id

GROUP BY c.customer_id, c.customer_name

HAVING SUM(s.quantity * p.price) > (

SELECT AVG(total_amount)

FROM (

SELECT SUM(s2.quantity * p2.price) **AS** total_amount

FROM 'data/sales.csv' s2

● サブクエリのベストプラクティス

- ✓ 可読性を重視 - 複雑になりすぎたらCTEを検討
- ✓ パフォーマンスに注意 - 相関サブクエリは遅い場合がある
- ✓ 適切な場所で使用 - JOINで解決できる場合はJOINを優先

サブクエリが適している場合

- 集計結果との比較（平均より大きい、など）
- 存在チェック（EXISTS）
- 動的な条件設定

2. トランザクション

データの整合性を保つ仕組み

● トランザクションとは？

トランザクション = 複数のSQL操作をひとまとまりとして扱う仕組み

- ✓ 全ての操作が成功するか、全て失敗するか (All or Nothing)
- ✓ データの整合性・一貫性を保証

例：銀行の送金処理

1. A口座から1万円を引く
2. B口座に1万円を足す

→ どちらか片方だけ成功してはダメ！

● ACID特性

トランザクションが満たすべき4つの特性：

A - Atomicity (原子性)

全て成功 or 全て失敗

C - Consistency (一貫性)

データの整合性が保たれる

I - Isolation (独立性)

● トランザクションの基本構文

-- トランザクション開始

BEGIN TRANSACTION;

-- 複数の操作

UPDATE accounts **SET** balance = balance - 10000 **WHERE** id = 'A001';

UPDATE accounts **SET** balance = balance + 10000 **WHERE** id = 'B001';

-- 成功したら確定

COMMIT;

-- 失敗したら取り消し

-- ROLLBACK;

● トランザクションの実例

-- 在庫管理の例

BEGIN TRANSACTION;

-- 1. 注文を記録

INSERT INTO orders (customer_id, product_id, quantity, order_date)
VALUES ('C001', 'P002', 5, '2024-01-30');

-- 2. 在庫を減らす

UPDATE inventory
SET stock = stock - 5
WHERE product_id = 'P002';

-- 3. 在庫が負になっていないかチェック

-- もし負ならROLLBACK、そうでなければCOMMIT

● 分離レベル

複数のトランザクションが同時実行される際の隔離度：

1. **READ UNCOMMITTED** - 最も低い（ダーティリード可能）
2. **READ COMMITTED** - コミット済みデータのみ読める
3. **REPEATABLE READ** - 同じデータを何度読んでも同じ
4. **SERIALIZABLE** - 最も高い（完全に隔離）



分離レベルが高いほど安全だが、パフォーマンスは低下

● デッドロック

2つ以上のトランザクションがお互いのロックを待ち続ける状態：

トランザクション1： テーブルAをロック → テーブルBを待つ
トランザクション2： テーブルBをロック → テーブルAを待つ

対策

- ロックの順序を統一
- タイムアウトの設定
- トランザクションを短く保つ

● DuckDBでのトランザクション

-- DuckDBでのトランザクション例

BEGIN;

-- CSVファイルから一時テーブルを作成

CREATE TEMP TABLE temp_sales **AS**
SELECT * FROM 'data/sales.csv';

-- データの更新

UPDATE temp_sales
SET quantity = quantity * 2
WHERE order_date >= '2024-01-20';

-- 結果を確認してからコミット

SELECT COUNT(*) FROM temp_sales **WHERE** quantity > 10;

3. インデックス

クエリを高速化する技術

● インデックスとは？

インデックス = データベースの「目次」や「索引」

- ✓ 特定の列の値を高速に検索できるようにする仕組み
- ✓ 本の巻末索引と同じ考え方

なぜ必要？

- テーブルが大きくなると検索が遅くなる
- 全データを見なくても目的のデータを見つけられる

● インデックスの仕組み

通常の検索（フルスキャン）：

顧客ID: C001 → C002 → C003 → ... → C999（全部見る）

インデックスを使った検索：

索引: C500 → 直接500番目のデータへジャンプ！

主なインデックスの種類

- **B-tree** - 最も一般的、範囲検索に強い
- **Hash** - 完全一致検索に特化
- **Bitmap** - カーディナリティが低い列に有効

● インデックスの作成

-- 基本的な構文

```
CREATE INDEX idx_customer_name  
ON customers(customer_name);
```

-- 複合インデックス (複数列)

```
CREATE INDEX idx_customer_date  
ON sales(customer_id, order_date);
```

-- ユニークインデックス (重複を許さない)

```
CREATE UNIQUE INDEX idx_email  
ON customers(email);
```

● インデックスのメリット・デメリット

メリット

- SELECT文が高速化
- ORDER BY、GROUP BYも高速化
- JOINの性能向上

デメリット

● 実行計画の確認

クエリがどのように実行されるかを確認：

```
-- DuckDBでの実行計画確認
```

```
EXPLAIN SELECT * FROM customers WHERE customer_name = '田中太郎';
```

```
-- より詳細な情報
```

```
EXPLAIN ANALYZE SELECT * FROM customers WHERE customer_name = '田中太郎';
```

実行計画で確認すること：

- インデックスが使われているか
- 処理時間の見積もり

● インデックス設計のベストプラクティス

- ✓ WHERE句でよく使う列にインデックスを作成
- ✓ JOIN条件の列にインデックスを作成
- ✓ カーディナリティが高い列を優先
- ✓ インデックスの数は最小限に

アンチパターン

- 全ての列にインデックスを作る
- 使われないインデックスを放置

● DuckDBでのインデックス例

-- 売上分析用のインデックス作成

```
CREATE INDEX idx_sales_date ON sales(order_date);
```

```
CREATE INDEX idx_sales_customer ON sales(customer_id);
```

-- パフォーマンス比較

-- インデックスなし

```
SELECT COUNT(*) FROM sales WHERE order_date = '2024-01-20';
```

-- インデックスあり（高速）

```
SELECT COUNT(*) FROM sales WHERE order_date = '2024-01-20';
```

-- 複合条件での検索

```
SELECT * FROM sales
```

```
WHERE customer_id = 'C001'
```

```
AND order_date = '2024-01-15';
```

4. ビュー

クエリの再利用と整理

● ビューとは？

ビュー = 保存されたSELECT文（仮想テーブル）

- ✓ 複雑なクエリに名前を付けて再利用
- ✓ 実際のデータは持たない（クエリの実行定義だけ）

-- ビューの作成

```
CREATE VIEW high_value_customers AS  
SELECT  
    c.customer_id,  
    c.customer_name,  
    SUM(s.quantity * p.price) AS total_purchase  
FROM customers c  
JOIN sales s ON c.customer_id = s.customer_id  
JOIN products p ON s.product_id = p.product_id  
GROUP BY c.customer_id, c.customer_name
```

● ビューの利点

1. 複雑さの隠蔽

長いクエリを短い名前呼び出せる

2. セキュリティ

特定の列だけを見せることができる

3. 独立性

テーブル構造が変わってもビューで吸収できる

● ビューの実例

-- 月次売上サマリービュー

```
CREATE VIEW monthly_sales_summary AS  
SELECT  
    DATE_TRUNC('month', order_date) AS month,  
    COUNT(DISTINCT customer_id) AS unique_customers,  
    COUNT(*) AS total_orders,  
    SUM(quantity) AS total_quantity,  
    SUM(quantity * price) AS total_revenue  
FROM sales s  
JOIN products p ON s.product_id = p.product_id  
GROUP BY DATE_TRUNC('month', order_date);
```

-- 使用例

```
SELECT * FROM monthly_sales_summary  
ORDER BY month ASC
```

● マテリアライズドビュー

通常のビューとの違い：

- **通常のビュー** - 毎回クエリを実行
- **マテリアライズドビュー** - 結果を保存（高速）

-- PostgreSQLなどでの例

```
CREATE MATERIALIZED VIEW product_stats AS  
SELECT  
    product_id,  
    COUNT(*) AS sell_count,  
    SUM(quantity) AS total_sold,  
    AVG(quantity) AS avg_quantity  
FROM sales  
GROUP BY product_id;
```


5. ウィンドウ関数

高度な分析機能

● ウィンドウ関数とは？

ウィンドウ関数 = 行のグループに対して計算を行う関数

- ✓ GROUP BYと違い、各行が残る
- ✓ ランキングや累計計算に便利

```
-- 基本構文
関数名() OVER (
    PARTITION BY 列名    -- グループ分け（省略可）
    ORDER BY 列名       -- 並び順
)
```

● ROW_NUMBER() - 連番付与

各行に連番を振る：

```
-- 顧客ごとに購入履歴に連番を付ける
SELECT
    customer_id,
    order_date,
    product_id,
    ROW_NUMBER() OVER (
        PARTITION BY customer_id
        ORDER BY order_date
    ) AS purchase_number
FROM 'data/sales.csv'
ORDER BY customer_id, order_date;
```

● RANK()とDENSE_RANK() - 順位付け

-- 商品の売上ランキング

SELECT

p.product_name,

SUM(s.quantity * p.price) **AS** total_sales,

RANK() **OVER** (ORDER BY SUM(s.quantity * p.price) **DESC**) **AS** rank,

DENSE_RANK() **OVER** (ORDER BY SUM(s.quantity * p.price) **DESC**) **AS** dense_rank

FROM 'data/sales.csv' s

JOIN 'data/products.csv' p **ON** s.product_id = p.product_id

GROUP BY p.product_id, p.product_name;

● LAG()とLEAD() - 前後の行を参照

-- 前回の購入からの経過日数

SELECT

customer_id,

order_date,

LAG(order_date) **OVER** (

PARTITION BY customer_id

ORDER BY order_date

) **AS** prev_order_date,

order_date - LAG(order_date) **OVER** (

PARTITION BY customer_id

ORDER BY order_date

) **AS** days since last order

● ウィンドウ関数での集計

-- 累計売上の計算

SELECT

order_date,
product_id,
quantity * price **AS** daily_sales,

SUM(quantity * price) **OVER** (

ORDER BY order_date

ROWS BETWEEN UNBOUNDED PRECEDING **AND** **CURRENT ROW**

) **AS** cumulative_sales

FROM 'data/sales.csv' s

JOIN 'data/products.csv' p **ON** s.product_id = p.product_id

ORDER BY order_date;

● 実践例：売上分析ダッシュボード

-- カテゴリ別商品ランキングと売上シェア

SELECT

category,
product_name,
total_sales,

RANK() **OVER** (**PARTITION BY** category **ORDER BY** total_sales **DESC**) **AS** category_rank,
total_sales / **SUM**(total_sales) **OVER** (**PARTITION BY** category) * 100 **AS** sales_share_pct

FROM (

SELECT

p.category,
p.product_name,
SUM(s.quantity * p.price) **AS** total_sales

FROM 'data/sales.csv' s

JOIN 'data/products.csv' p **ON** s.product_id = p.product_id

6. CTE (Common Table Expressions)

WITH句で複雑なクエリを整理

● CTEとは？

CTE = 一時的な名前付き結果セット

- ✓ 複雑なクエリを段階的に組み立てる
- ✓ 可読性が大幅に向上

```
WITH cte_name AS (  
    -- ここにSELECT文  
)  
SELECT * FROM cte_name;
```

● CTEの基本例

```
-- 高額購入顧客の分析
WITH high_value_orders AS (
    SELECT
        customer_id,
        SUM(quantity * price) AS total_amount
    FROM sales s
    JOIN products p ON s.product_id = p.product_id
    GROUP BY customer_id
    HAVING SUM(quantity * price) > 50000
)
SELECT
    c.customer_name,
    h.total_amount
FROM high_value_orders h
JOIN customers c ON h.customer_id = c.customer_id
```

● 複数のCTE

```
-- 段階的な分析
WITH
-- 1. 顧客ごとの購入金額
customer_totals AS (
    SELECT customer_id, SUM(quantity * price) AS total
    FROM sales s
    JOIN products p ON s.product_id = p.product_id
    GROUP BY customer_id
),
-- 2. 全体の平均
overall_avg AS (
    SELECT AVG(total) AS avg_total
    FROM customer_totals
)
-- 3. 平均との比較
SELECT
    ct.customer_id,
    ct.total,
    oa.avg_total,
    ct.total - oa.avg_total AS diff_from_avg
FROM customer_totals ct
```

● 再帰CTE

階層構造や連続データの処理：

```
-- 連続する日付の生成
WITH RECURSIVE date_series AS (
  -- 初期値
  SELECT DATE '2024-01-01' AS date

  UNION ALL

  -- 再帰部分
  SELECT date + INTERVAL 1 DAY
  FROM date_series
  WHERE date < DATE '2024-01-31'
)
SELECT * FROM date_series;
```

● CTEとサブクエリの使い分け

CTEを使うべき場合

- 同じ結果を複数回参照する
- クエリの可読性を重視する
- 段階的な処理を明確にしたい

サブクエリで十分な場合

- 単純な条件フィルタ
- 一度だけ使用する

7. その他の重要概念

実務で役立つ機能たち

● データ型の詳細

日付・時刻型

-- DuckDBの日付関数

SELECT

CURRENT_DATE AS 今日,

CURRENT_TIMESTAMP AS 現在時刻,

DATE '2024-01-20' + INTERVAL 7 DAY AS 一週間後,

DATE_TRUNC('month', CURRENT_DATE) AS 月初,

DATE_PART('year', order_date) AS 年,

DATE_PART('month', order_date) AS 月

FROM sales

LIMIT 5;

● 文字列関数

-- よく使う文字列操作

SELECT

-- 連結

customer_name || ' 様' **AS** 敬称付き,

-- 部分文字列

SUBSTRING(email, 1, **POSITION**(' @ ' **IN** email) - 1) **AS** username,

-- 大文字/小文字

UPPER(category) **AS** category_upper,

-- 置換

REPLACE(phone, '-', '') **AS** phone_digits,

-- 長さ

LENGTH(customer_name) **AS** name_length

FROM customers;

● 正規表現

パターンマッチングの強力な機能：

-- メールアドレスの検証

SELECT

email,

CASE

WHEN email ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\$'

THEN '有効'

ELSE '無効'

END AS email_validity

FROM customers;

-- 特定パターンの抽出

SELECT

customer_name,

● CASE文の高度な使い方

-- 複雑な条件分岐

SELECT

customer_name,
total_purchase,

CASE

WHEN total_purchase >= 100000 **THEN** 'プラチナ'

WHEN total_purchase >= 50000 **THEN** 'ゴールド'

WHEN total_purchase >= 10000 **THEN** 'シルバー'

WHEN total_purchase > 0 **THEN** 'ブロンズ'

ELSE '新規'

END AS customer_rank,

CASE

WHEN last_purchase_date >= CURRENT_DATE - INTERVAL 30 DAY **THEN** 'アクティブ'

WHEN last_purchase_date >= CURRENT_DATE - INTERVAL 90 DAY **THEN** '要注意'

ELSE '休眠'

END AS activity_status

● NULL処理関数

```
-- NULL値の扱い
SELECT
  customer_name,
  -- NULLを別の値に置換
  COALESCE(phone, 'なし') AS phone_display,
  -- 条件によってNULLを返す
  NULLIF(quantity, 0) AS quantity_or_null,
  -- 複数の値から最初の非NULL値
  COALESCE(mobile_phone, home_phone, office_phone, '連絡先なし') AS contact
FROM customers;
```

● 集合演算

-- UNION: 重複を除いて結合

```
SELECT customer_id FROM sales_2023
UNION
SELECT customer_id FROM sales_2024;
```

-- UNION ALL: 重複も含めて結合

```
SELECT product_id FROM inventory_tokyo
UNION ALL
SELECT product_id FROM inventory_osaka;
```

-- INTERSECT: 共通部分

```
SELECT customer_id FROM high_value_customers
INTERSECT
SELECT customer_id FROM frequent_buyers;
```

-- EXCEPT: 差集合

```
SELECT customer_id FROM all_customers
EXCEPT
```

● JSON処理

最近のデータベースではJSON形式のデータも扱える：

```
-- JSON列からデータ抽出
```

```
SELECT
```

```
    customer_id,  
    preferences->>'$.color' AS favorite_color,  
    preferences->>'$.size' AS preferred_size,  
    JSON_ARRAY_LENGTH(preferences->'$.interests') AS interest_count
```

```
FROM customer_preferences;
```

```
-- JSONの生成
```

```
SELECT
```

```
    JSON_OBJECT(  
        'customer_id', customer_id,  
        'name', customer_name,  
        'purchases', JSON_ARRAY(  
            SELECT product_id FROM sales WHERE customer_id = c.customer_id
```

● パフォーマンスチューニングのヒント

- ✓ 適切なインデックスの作成
- ✓ 不要なデータの除外（WHERE句を早めに）
- ✓ JOINの順序を意識する
- ✓ サブクエリよりJOINを優先
- ✓ EXPLAIN ANALYZEで実行計画を確認

```
-- 実行計画の確認  
EXPLAIN ANALYZE  
SELECT ... FROM ... WHERE ...;
```

● まとめ

今回学んだ発展的な機能：

1. **サブクエリ** - 複雑な条件を段階的に処理
2. **トランザクション** - データの整合性を保証
3. **インデックス** - クエリ的高速化
4. **ビュー** - クエリの再利用と整理
5. **ウィンドウ関数** - 高度な分析
6. **CTE** - 可読性の向上
7. **その他** - 実務で役立つ様々な機能

● SQLマスターへの道

次のステップ

1. **実データでの練習** - 自社のデータで試してみる
2. **パフォーマンス意識** - 大量データでの最適化
3. **データベース固有機能** - PostgreSQL、MySQL等の独自機能
4. **NoSQL** - MongoDB、Redisなど別パラダイム
5. **データエンジニアリング** - ETL、データパイプライン



SQLは奥が深い。実践を重ねて、少しずつマスターしていきましょう！

● 参考リソース

ドキュメント

- [DuckDB Documentation](#)
- [PostgreSQL Documentation](#)

学習サイト

- [SQL Tutorial](#)
- [Mode SQL Tutorial](#)

書籍

お疲れ様でした！

これでSQL勉強会の全カリキュラムが完了です。

基本から応用まで幅広く学んだ知識を、
ぜひ実務で活用してください。

Happy Querying! 🎉