## SQLの世界へようこそ！ ウィンドウ関数、ビュー、CTEをマスターしよう

この教材では、SQLの強力な機能である「ウィンドウ関数」「ビュー」「CTE（共通テーブル式）」について学びます。これらの機能を使いこなせば、データ分析やレポート作成の効率が格段に向上します！

### 1. はじめに：それぞれの役割を理解しよう

まずは、それぞれの機能がどのような役割を持っているのか、ざっくりと理解しましょう。

* **ウィンドウ関数:**
  + **例え:** テストの成績表で、クラス全体の平均点だけでなく、「自分の順位」や「平均点との差」も各生徒の行に表示する。
  + **役割:** 各行のデータはそのままに、関連する行のグループ（ウィンドウ）に基づいた計算結果を新しい列として追加する。
  + **得意なこと:** ランキング、移動平均、累積合計など、行ごとの詳細情報を保持したまま計算を行う。
* **ビュー:**
  + **例え:** よく使うWebサイトのブックマーク（ショートカット）。
  + **役割:** よく使うSELECT文（クエリ）に名前を付けてデータベースに保存しておく。
  + **得意なこと:** 複雑なクエリを隠蔽し、簡単な名前でアクセスできるようにする。データの再利用、セキュリティ向上。
* **CTE（共通テーブル式）:**
  + **例え:** 料理のレシピで、下ごしらえやソース作りなどの各ステップを「小さなレシピ」として定義し、最後に組み合わせる。
  + **役割:** SELECT文の前に、一時的な名前付きの結果セット（テーブルのようなもの）を定義する。
  + **得意なこと:** 複雑なクエリを論理的な単位に分割し、可読性・保守性を向上させる。クエリ内での再利用、再帰処理。

### 2. ウィンドウ関数：データを見ながら計算！（詳細解説）

前回の基本編に加えて、より深くウィンドウ関数を理解するための追加の説明と、さまざまなウィンドウ関数の使用例を紹介します。

#### 2.1 ウィンドウ関数の種類と使い分け（詳細）

前回紹介したウィンドウ関数以外にも、様々な関数があります。大きく分けて以下の3つのカテゴリに分類できます。

1. **順位付け関数:** 行に順位を付ける
   * ROW\_NUMBER(): 重複なしの連番を振ります。
   * RANK(): 同じ値には同じ順位を付け、次の順位は飛ばします（1, 2, 2, 4, ...）。
   * DENSE\_RANK(): 同じ値には同じ順位を付け、次の順位は飛ばしません（1, 2, 2, 3, ...）。
   * NTILE(n): 行をn個のグループに分け、各行が属するグループ番号を返します。
2. **集計関数:** ウィンドウ内の値を集計する
   * SUM(): 合計
   * AVG(): 平均
   * MIN(): 最小値
   * MAX(): 最大値
   * COUNT(): 件数
3. **値取得関数:** ウィンドウ内の特定の位置の行の値を取得する
   * LAG(列, n, default): 現在の行からn行前の値を取得します。nを省略すると1、defaultは値がない場合のデフォルト値です。
   * LEAD(列, n, default): 現在の行からn行後の値を取得します。
   * FIRST\_VALUE(列): ウィンドウ内の最初の行の値を取得します。
   * LAST\_VALUE(列): ウィンドウ内の最後の行の値を取得します。
   * NTH\_VALUE(列, n): ウィンドウ内のn番目の行の値を取得します。

**使い分けのポイント:**

* **連番を振りたい:** ROW\_NUMBER()
* **順位を付けたい（同順位を考慮）:** RANK() または DENSE\_RANK()
* **グループ分けしたい:** NTILE()
* **集計したい:** SUM(), AVG(), MIN(), MAX(), COUNT()
* **前後の行の値と比較したい:** LAG(), LEAD()
* **ウィンドウの最初/最後/n番目の値を取得したい:** FIRST\_VALUE(), LAST\_VALUE(), NTH\_VALUE()

#### 2.2 SQL文の例（さまざまなウィンドウ関数）

引き続き、社員テーブルと部署テーブルを使用します。

**[社員テーブル (Shain)]**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **社員ID** | **氏名** | **部署ID** | **給与** | **入社日** |
| 1 | 山田 | 10 | 600 | 2022-04-01 |
| 2 | 田中 | 10 | 700 | 2022-05-01 |
| 3 | 鈴木 | 20 | 500 | 2022-06-01 |
| 4 | 佐藤 | 20 | 550 | 2022-07-01 |
| 5 | 伊藤 | 10 | 800 | 2022-08-01 |
| 6 | 加藤 | 20 | 650 | 2022-09-01 |
| 7 | 渡辺 | 10 | 750 | 2022-10-01 |

**[部署テーブル (Busho)]**

|  |  |
| --- | --- |
| **部署ID** | **部署名** |
| 10 | 営業部 |
| 20 | 開発部 |

**例1：ROW\_NUMBER()：社員ID順に通し番号を振る**

SELECT

ROW\_NUMBER() OVER (ORDER BY 社員ID) AS 連番,

社員ID,

氏名

FROM

社員;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**結果:**

|  |  |  |
| --- | --- | --- |
| **連番** | **社員ID** | **氏名** |
| 1 | 1 | 山田 |
| 2 | 2 | 田中 |
| 3 | 3 | 鈴木 |
| 4 | 4 | 佐藤 |
| 5 | 5 | 伊藤 |
| 6 | 6 | 加藤 |
| 7 | 7 | 渡辺 |

**例2：RANK()：部署ごとに給与が高い順に順位付け（同順位あり）**

SELECT

氏名,

部署名,

給与,

RANK() OVER (PARTITION BY 部署名 ORDER BY 給与 DESC) AS 部署内順位

FROM

社員 s

JOIN

部署 b ON s.部署ID = b.部署ID;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**結果:**

|  |  |  |  |
| --- | --- | --- | --- |
| **氏名** | **部署名** | **給与** | **部署内順位** |
| 伊藤 | 営業部 | 800 | 1 |
| 渡辺 | 営業部 | 750 | 2 |
| 田中 | 営業部 | 700 | 3 |
| 山田 | 営業部 | 600 | 4 |
| 加藤 | 開発部 | 650 | 1 |
| 佐藤 | 開発部 | 550 | 2 |
| 鈴木 | 開発部 | 500 | 3 |

**例3：DENSE\_RANK()：部署ごとに給与が高い順に順位付け（同順位あり、順位を詰める）**

SELECT

氏名,

部署名,

給与,

DENSE\_RANK() OVER (PARTITION BY 部署名 ORDER BY 給与 DESC) AS 部署内順位

FROM

社員 s

JOIN

部署 b ON s.部署ID = b.部署ID;

-- 結果はRANK()と同じ（このデータでは違いが出ない）

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**例：給与を高い方から３３％、３３％、３４％にグルーピング**

SELECT

氏名,

部署名,

給与,

NTILE(3) OVER (ORDER BY 給与 DESC) AS 給与グループ

FROM

社員 s

JOIN

部署 b ON s.部署ID = b.部署ID;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**結果:**  
| 氏名 | 部署名 | 給与 | 給与グループ |  
| ---- | ------ | ---- | ---------- |  
| 伊藤 | 営業部 | 800 | 1 |  
| 渡辺 | 営業部 | 750 | 1 |  
| 田中 | 営業部 | 700 | 2 |  
| 加藤 | 開発部 | 650 | 2 |  
| 山田 | 営業部 | 600 | 3 |  
| 佐藤 | 開発部 | 550 | 3 |  
| 鈴木 | 開発部 | 500 | 3 |  
**例4：LAG()：1つ前の行の給与を取得**

SELECT

氏名,

入社日,

給与,

LAG(給与, 1, 0) OVER (ORDER BY 入社日) AS 1つ前の給与 -- 1つ前がない場合は0

FROM

社員;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**結果:**

|  |  |  |  |
| --- | --- | --- | --- |
| **氏名** | **入社日** | **給与** | **1つ前の給与** |
| 山田 | 2022-04-01 | 600 | 0 |
| 田中 | 2022-05-01 | 700 | 600 |
| 鈴木 | 2022-06-01 | 500 | 700 |
| 佐藤 | 2022-07-01 | 550 | 500 |
| 伊藤 | 2022-08-01 | 800 | 550 |
| 加藤 | 2022-09-01 | 650 | 800 |
| 渡辺 | 2022-10-01 | 750 | 650 |

**例5：LEAD()：1つ後の行の給与を取得**

SELECT

氏名,

入社日,

給与,

LEAD(給与, 1, 0) OVER (ORDER BY 入社日) AS 1つ後の給与 -- 1つ後がない場合は0

FROM

社員;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**結果:**

|  |  |  |  |
| --- | --- | --- | --- |
| **氏名** | **入社日** | **給与** | **1つ後の給与** |
| 山田 | 2022-04-01 | 600 | 700 |
| 田中 | 2022-05-01 | 700 | 500 |
| 鈴木 | 2022-06-01 | 500 | 550 |
| 佐藤 | 2022-07-01 | 550 | 800 |
| 伊藤 | 2022-08-01 | 800 | 650 |
| 加藤 | 2022-09-01 | 650 | 750 |
| 渡辺 | 2022-10-01 | 750 | 0 |

**例6：FIRST\_VALUE()：部署内で最も給与が高い社員の給与を取得**

SELECT

氏名,

部署名,

給与,

FIRST\_VALUE(給与) OVER (PARTITION BY 部署名 ORDER BY 給与 DESC) AS 部署内最高給与

FROM

社員 s

JOIN

部署 b ON s.部署ID = b.部署ID;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**結果:**

|  |  |  |  |
| --- | --- | --- | --- |
| **氏名** | **部署名** | **給与** | **部署内最高給与** |
| 伊藤 | 営業部 | 800 | 800 |
| 渡辺 | 営業部 | 750 | 800 |
| 田中 | 営業部 | 700 | 800 |
| 山田 | 営業部 | 600 | 800 |
| 加藤 | 開発部 | 650 | 650 |
| 佐藤 | 開発部 | 550 | 650 |
| 鈴木 | 開発部 | 500 | 650 |

**例7：LAST\_VALUE()：部署内で最も給与が低い社員の給与を取得（注意点あり）**

SELECT

氏名,

部署名,

給与,

LAST\_VALUE(給与) OVER (PARTITION BY 部署名 ORDER BY 給与 DESC

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS 部署内最低給与

FROM

社員 s

JOIN

部署 b ON s.部署ID = b.部署ID;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**結果:**

|  |  |  |  |
| --- | --- | --- | --- |
| **氏名** | **部署名** | **給与** | **部署内最低給与** |
| 伊藤 | 営業部 | 800 | 600 |
| 渡辺 | 営業部 | 750 | 600 |
| 田中 | 営業部 | 700 | 600 |
| 山田 | 営業部 | 600 | 600 |
| 加藤 | 開発部 | 650 | 500 |
| 佐藤 | 開発部 | 550 | 500 |
| 鈴木 | 開発部 | 500 | 500 |

* **注意:** LAST\_VALUE() を使う場合は、ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING のようなフレーム指定を明示的に行うのが安全です。デフォルトのフレーム指定（RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW）では、期待通りの結果にならないことがあります。

#### 2.3 フレーム指定の詳細

フレーム指定は、ウィンドウ内の計算対象となる行の範囲を細かく制御する機能です。ROWS と RANGE の2つの方法があります。

* **ROWS:** 行数で範囲を指定します。
  + ROWS BETWEEN start AND end の形式で指定します。
  + start と end には以下のいずれかを指定できます。
    - UNBOUNDED PRECEDING: ウィンドウの先頭
    - UNBOUNDED FOLLOWING: ウィンドウの末尾
    - CURRENT ROW: 現在の行
    - n PRECEDING: 現在の行からn行前
    - n FOLLOWING: 現在の行からn行後
* **RANGE:** 値の範囲で指定します。
  + RANGE BETWEEN start AND end の形式で指定します。
  + ORDER BY で指定した列の値に基づいて範囲を決定します。
  + start と end には以下のいずれかを指定できます。
    - UNBOUNDED PRECEDING: ウィンドウの先頭の値
    - UNBOUNDED FOLLOWING: ウィンドウの末尾の値
    - CURRENT ROW: 現在の行の値
    - n PRECEDING: 現在の行の値からnを引いた値
    - n FOLLOWING: 現在の行の値にnを足した値

**ROWS と RANGE の違いの例:**

-- データ

CREATE TABLE sales (

product\_id INT,

sale\_date DATE,

amount INT

);

INSERT INTO sales VALUES

(1, '2023-01-01', 100),

(1, '2023-01-02', 100),

(1, '2023-01-03', 150),

(1, '2023-01-04', 200),

(2, '2023-01-01', 50),

(2, '2023-01-02', 100),

(2, '2023-01-03', 100),

(2, '2023-01-04', 150);

-- ROWS: 直近3日間の売上合計（自分自身を含む）

SELECT

product\_id,

sale\_date,

amount,

SUM(amount) OVER (PARTITION BY product\_id ORDER BY sale\_date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS recent\_3day\_sales\_rows

FROM

sales;

-- RANGE: 直近3日間の売上合計（自分自身を含む）

SELECT

product\_id,

sale\_date,

amount,

SUM(amount) OVER (PARTITION BY product\_id ORDER BY sale\_date RANGE BETWEEN INTERVAL '2' DAY PRECEDING AND CURRENT ROW) AS recent\_3day\_sales\_range

FROM

sales;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**結果 (ROWS):**

|  |  |  |  |
| --- | --- | --- | --- |
| **product\_id** | **sale\_date** | **amount** | **recent\_3day\_sales\_rows** |
| 1 | 2023-01-01 | 100 | 100 |
| 1 | 2023-01-02 | 100 | 200 |
| 1 | 2023-01-03 | 150 | 350 |
| 1 | 2023-01-04 | 200 | 450 |
| 2 | 2023-01-01 | 50 | 50 |
| 2 | 2023-01-02 | 100 | 150 |
| 2 | 2023-01-03 | 100 | 250 |
| 2 | 2023-01-04 | 150 | 350 |

**結果 (RANGE):**

|  |  |  |  |
| --- | --- | --- | --- |
| **product\_id** | **sale\_date** | **amount** | **recent\_3day\_sales\_range** |
| 1 | 2023-01-01 | 100 | 100 |
| 1 | 2023-01-02 | 100 | 200 |
| 1 | 2023-01-03 | 150 | 350 |
| 1 | 2023-01-04 | 200 | 450 |
| 2 | 2023-01-01 | 50 | 50 |
| 2 | 2023-01-02 | 100 | 150 |
| 2 | 2023-01-03 | 100 | 250 |
| 2 | 2023-01-04 | 150 | 350 |

この例では、ROWS と RANGE で結果が同じになりますが、例えば、同じ日に複数の売上がある場合など、ORDER BY で指定した列に重複がある場合は、結果が異なる可能性があります。

#### 2.4 ウィンドウ関数の注意点（再掲）

* ウィンドウ関数はSELECT句とORDER BY句でのみ使用できます。
* OVER()句の指定によっては、パフォーマンスに影響が出る可能性があります。特に、大きなテーブルで複雑なフレーム指定（特にRANGE）を行う場合は注意が必要です。
* データベースシステムによっては、サポートされていないウィンドウ関数や機能があります。

今回は、ウィンドウ関数についてさらに詳しく解説しました。次回は、ビューについて掘り下げていきます。

38.6s

edit

more\_vert

続けてください

edit

more\_vert

はい、承知いたしました。引き続き、ビューについて詳しく解説していきます。

### 3. ビュー：クエリに名前を付けて保存！（詳細解説）

前回はビューの基本的な概念とメリット・注意点について説明しました。今回は、ビューの作成・使用・管理に関するより詳細な情報と、応用的な使い方について解説します。

#### 3.1 ビューの作成（CREATE VIEW）

ビューを作成するには、CREATE VIEW ステートメントを使用します。

CREATE [OR REPLACE] VIEW ビュー名 [ (列名1, 列名2, ...) ]

AS

SELECT ... -- ビューの定義となるSELECT文

[WITH CHECK OPTION];

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

* **CREATE VIEW ビュー名**: 新しいビューを作成します。
* **OR REPLACE** (オプション): 同じ名前のビューが既に存在する場合に、既存のビューを置き換えます。
* **(列名1, 列名2, ...)** (オプション): ビューの列名を指定します。省略した場合は、SELECT文で選択された列の名前が使用されます。
* **AS SELECT ...**: ビューの定義となるSELECT文を記述します。このSELECT文が、ビューにアクセスしたときに実行されるクエリになります。
* **WITH CHECK OPTION** (オプション): ビューを通じて行われる更新（INSERT、UPDATE、DELETE）が、ビューの定義のWHERE句の条件を満たすように強制します。

**例：**

-- 「高給与社員」ビューを作成（給与が800万円以上の社員）

CREATE OR REPLACE VIEW 高給与社員 (社員番号, 氏名, 部署名, 給与)

AS

SELECT

s.社員ID,

s.氏名,

b.部署名,

s.給与

FROM

社員 s

JOIN

部署 b ON s.部署ID = b.部署ID

WHERE

s.給与 >= 8000000

WITH CHECK OPTION;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

この例では、「高給与社員」という名前のビューを作成しています。ビューの列名として、「社員番号」「氏名」「部署名」「給与」を指定しています。WITH CHECK OPTION を指定しているため、このビューを通じて給与が800万円未満になるような更新はできなくなります。

#### 3.2 ビューの使用（SELECT）

ビューは、テーブルと同じように SELECT 文で参照できます。

SELECT 列名1, 列名2, ... FROM ビュー名 WHERE 条件 ...;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**例：**

-- 「高給与社員」ビューから、氏名と給与を取得

SELECT 氏名, 給与 FROM 高給与社員;

-- 「高給与社員」ビューで、給与が1000万円以上の社員を抽出

SELECT \* FROM 高給与社員 WHERE 給与 >= 10000000;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

#### 3.3 ビューの更新（INSERT, UPDATE, DELETE）

単純なビュー（単一のテーブルに対するSELECTで、集計関数やウィンドウ関数などを含まない）は、更新可能な場合があります。ビューに対する更新は、元のテーブルに反映されます。

* **INSERT:** ビューに新しい行を追加します。
* **UPDATE:** ビューの既存の行を更新します。
* **DELETE:** ビューから行を削除します。

**例：**

-- 「営業部社員」ビューを作成

CREATE VIEW 営業部社員 AS

SELECT 社員ID, 氏名, 給与

FROM 社員

WHERE 部署ID = (SELECT 部署ID FROM 部署 WHERE 部署名 = '営業部');

-- 「営業部社員」ビューに新しい社員を追加

INSERT INTO 営業部社員 (社員ID, 氏名, 給与) VALUES (8, '高橋', 6500000);

-- 「営業部社員」ビューの社員の給与を更新

UPDATE 営業部社員 SET 給与 = 7000000 WHERE 社員ID = 8;

-- 「営業部社員」ビューから社員を削除

DELETE FROM 営業部社員 WHERE 社員ID = 8;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**注意点：**

* ビューが更新可能かどうかは、ビューの定義によります。複雑なビュー（複数のテーブルの結合、集計関数、ウィンドウ関数、DISTINCT、GROUP BY、UNION などを含む）は、通常は更新できません。
* WITH CHECK OPTION を指定したビューでは、ビューの定義のWHERE句の条件を満たさない更新は拒否されます。
* ビューを通じてデータを更新する際は、意図しないデータの変更を防ぐため、十分な注意が必要です。

#### 3.4 ビューの変更（ALTER VIEW）

ビューの定義を変更するには、ALTER VIEW ステートメントを使用します（データベースシステムによってはサポートされていない場合があります）。

ALTER VIEW ビュー名 AS

SELECT ... -- 新しいSELECT文

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

#### 3.5 ビューの削除（DROP VIEW）

ビューを削除するには、DROP VIEW ステートメントを使用します。

DROP VIEW ビュー名;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

#### 3.6 ビューの情報を確認する

データベースシステムによっては、ビューの定義や情報を確認するためのコマンドやシステムテーブルが用意されています。

* **MySQL:** SHOW CREATE VIEW ビュー名;
* **PostgreSQL:** \d+ ビュー名
* **SQL Server:** sp\_helptext 'ビュー名'
* **Oracle:** SELECT text FROM user\_views WHERE view\_name = 'ビュー名'; (ビュー名は大文字)

#### 3.7 マテリアライズドビュー (Materialized View)

通常のビューはクエリの定義を保存するだけですが、マテリアライズドビューはクエリの結果を物理的に保存します。

* **メリット:** 頻繁にアクセスされる複雑なクエリの結果を高速に取得できます。
* **デメリット:** データの更新は手動または自動で行う必要があります。

**作成 (PostgreSQLの例):**

CREATE MATERIALIZED VIEW 部署別給与統計 AS

SELECT

b.部署名,

AVG(s.給与) AS 平均給与,

MAX(s.給与) AS 最高給与,

MIN(s.給与) AS 最低給与

FROM

社員 s

JOIN

部署 b ON s.部署ID = b.部署ID

GROUP BY

b.部署名;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**更新 (PostgreSQLの例):**

REFRESH MATERIALIZED VIEW 部署別給与統計;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**注意点：**

* マテリアライズドビューは、すべてのデータベースシステムでサポートされているわけではありません（PostgreSQL, Oracle, SQL Serverなどではサポートされていますが、MySQLでは標準ではサポートされていません）。
* マテリアライズドビューの更新方法（手動/自動、頻度など）は、データベースシステムや要件によって異なります。

#### 3.8 ビューの応用例

* **データの抽象化:** 複雑なテーブル構造を隠蔽し、ユーザーにシンプルなインターフェースを提供します。
* **セキュリティ:** 特定のユーザーに、特定の列や行のみを公開します。
* **レポート作成:** よく使う集計クエリをビューとして定義し、レポート作成を効率化します。
* **アプリケーション開発:** アプリケーションから直接テーブルにアクセスする代わりに、ビューを通じてアクセスすることで、テーブル構造の変更の影響を最小限に抑えます。
* **データの仮想的な分割**　大きなテーブルを、論理的に分割したように見せる。  
  例えば、年ごとに社員情報を管理している場合。
* CREATE VIEW 2022年社員 AS SELECT \* FROM 社員 WHERE 入社日 BETWEEN '2022-01-01' AND '2022-12-31';

CREATE VIEW 2023年社員 AS SELECT \* FROM 社員 WHERE 入社日 BETWEEN '2023-01-01' AND '2023-12-31';

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

今回は、ビューについてさらに詳しく解説しました。次回はCTEについて掘り下げていきます。

22.7s

edit

more\_vert

続けてください

edit

more\_vert

はい、承知いたしました。引き続き、CTE（共通テーブル式）について詳しく解説していきます。

### 4. CTE（共通テーブル式）：クエリを分割して整理！（詳細解説）

前回はCTEの基本的な概念とメリット・注意点について説明しました。今回は、CTEのより詳細な構文、再帰CTE、複数のCTEの組み合わせ、そして他のSQL要素（ウィンドウ関数やビュー）との比較について解説します。

#### 4.1 CTEの構文（詳細）

CTEは、WITH句を使って定義します。

WITH CTE名 [(列名1, 列名2, ...)] AS (

SELECT ... -- CTEの定義となるSELECT文

)

SELECT ... -- メインクエリ（CTEを参照できる）

;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

* **WITH CTE名**: CTEを定義します。CTE名は、メインクエリ内で参照する際の名前になります。
* **(列名1, 列名2, ...)** (オプション): CTEの列名を指定します。省略した場合は、SELECT文で選択された列の名前が使用されます。
* **AS ( SELECT ... )**: CTEの定義となるSELECT文を記述します。このSELECT文の結果が、CTEの内容になります。
* **SELECT ...**: メインクエリです。このクエリの中で、CTEをテーブルのように参照できます。

**ポイント:**

* CTEは、定義したクエリの中でのみ有効です。クエリの実行が終了すると、CTEも消滅します。
* CTEは、データベースに永続的に保存されるオブジェクトではありません。
* CTEは、主にクエリの可読性と再利用性を高めるために使用されます。

#### 4.2 再帰CTE (Recursive CTE)

再帰CTEは、CTE自身を参照するCTEです。階層構造を持つデータ（組織図、部品表など）を処理する際に非常に便利です。

WITH RECURSIVE CTE名 (列名リスト) AS (

-- ベースケース（再帰の初期値）

SELECT ...

UNION ALL

-- 再帰部分（CTE自身を参照）

SELECT ...

FROM CTE名

WHERE ...

)

SELECT ... -- メインクエリ

FROM CTE名;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

* **WITH RECURSIVE**: 再帰CTEを定義することを宣言します。
* **ベースケース:** 再帰の初期値となるSELECT文です。最初に実行され、再帰の起点となります。
* **UNION ALL**: ベースケースと再帰部分の結果を結合します。
* **再帰部分:** CTE自身を参照するSELECT文です。ベースケースの結果に対して再帰的に適用され、結果が追加されていきます。
* **WHERE句 (再帰部分):** 再帰を停止する条件を指定します。この条件が満たされなくなると、再帰は終了します。

**例：社員テーブルから、上司と部下の関係を再帰的にたどり、階層構造を表示する**

-- 社員テーブル (Shain) に、上司の社員ID (上司ID) 列があると仮定

WITH RECURSIVE 社員階層 AS (

-- ベースケース：社長（上司IDがNULL）を起点とする

SELECT

社員ID,

氏名,

上司ID,

CAST(氏名 AS VARCHAR(255)) AS 階層パス, -- 階層パスを文字列で表現

0 AS レベル

FROM

社員

WHERE

上司ID IS NULL

UNION ALL

-- 再帰部分：部下をたどる

SELECT

e.社員ID,

e.氏名,

e.上司ID,

CONCAT(h.階層パス, ' -> ', e.氏名), -- 階層パスを連結

h.レベル + 1

FROM

社員 e

INNER JOIN

社員階層 h ON e.上司ID = h.社員ID -- 上司と部下をJOIN

)

SELECT

\*

FROM

社員階層

ORDER BY

階層パス;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**解説:**

1. **ベースケース:** 上司ID IS NULL の社員（社長）を起点として、社員ID、氏名、上司ID、階層パス（最初は氏名）、レベル（最初は0）を取得します。
2. **再帰部分:**
   * 社員階層 CTE（h）と 社員 テーブル（e）を e.上司ID = h.社員ID で結合します（部下の上司IDと上司の社員IDを一致させる）。
   * 部下の社員ID、氏名、上司IDを取得します。
   * 階層パスは、上司の階層パスに -> と部下の氏名を連結して作成します。
   * レベルは、上司のレベルに1を加算します。
3. **UNION ALL:** ベースケースの結果と再帰部分の結果を結合します。
4. **再帰の停止:** 再帰部分は、社員階層 CTEに新しい行が追加されなくなるまで（つまり、すべての部下をたどり終えるまで）繰り返されます。
5. **メインクエリ:** 社員階層 CTEからすべての列を選択し、階層パスで並べ替えて表示します。

**注意点:**

* 再帰CTEは、無限ループに陥る可能性があります。再帰部分のWHERE句で適切な停止条件を指定することが重要です。
* データベースシステムによっては、再帰の深さに制限がある場合があります（MAXRECURSION オプションなどで設定できる場合もあります）。

#### 4.3 複数のCTEの組み合わせ

複数のCTEを定義し、それらを組み合わせて使うことで、複雑なクエリをより構造的に記述できます。

WITH CTE1 AS (

SELECT ...

), CTE2 AS (

SELECT ...

), CTE3 AS (

SELECT ...

)

SELECT ...

FROM CTE1

JOIN CTE2 ON ...

WHERE ...

;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

**例：**

-- 各部署の平均給与を計算し、その平均給与よりも高い給与の社員を抽出し、

-- さらに、その社員が参加しているプロジェクトの平均予算を計算する

WITH 部署別平均給与 AS (

SELECT

b.部署ID,

AVG(s.給与) AS 平均給与

FROM

社員 s

JOIN

部署 b ON s.部署ID = b.部署ID

GROUP BY

b.部署ID

), 高給与社員 AS (

SELECT

s.社員ID,

s.部署ID

FROM

社員 s

JOIN

部署別平均給与 d ON s.部署ID = d.部署ID

WHERE

s.給与 > d.平均給与

), 参加プロジェクト AS (

SELECT

p.プロジェクトID,

p.予算,

h.社員ID

FROM

プロジェクト p

JOIN

プロジェクト参加 pa ON p.プロジェクトID = pa.プロジェクトID

JOIN

高給与社員 h ON h.社員ID = pa.社員ID

)

SELECT

AVG(予算) AS 平均予算

FROM

参加プロジェクト;

Use code [with caution](https://support.google.com/legal/answer/13505487).SQL

#### 4.4 CTEと他のSQL要素との比較

* **CTE vs. サブクエリ:**
  + **サブクエリ:** SELECT文の中に別のSELECT文を記述する。
  + **CTE:** SELECT文の前に名前付きのクエリを定義する。
  + **CTEの利点:**
    - 可読性が高い：クエリを論理的な単位に分割できる。
    - 再利用性：同じCTEをクエリ内で複数回参照できる。
    - 再帰処理：再帰CTEを使える。
  + **サブクエリの利点:**
    - 簡潔性：単純なクエリの場合は、サブクエリの方が簡潔に記述できる場合がある。
* **CTE vs. ビュー:**
  + **ビュー:** データベースに永続的に保存されるオブジェクト。
  + **CTE:** クエリの実行中にのみ存在する一時的なオブジェクト。
  + **ビューの利点:**
    - 永続性：複数のクエリやアプリケーションで再利用できる。
    - セキュリティ：ユーザーに特定の列や行のみを公開できる。
  + **CTEの利点:**
    - 可読性：複雑なクエリを分割できる。
    - 再帰処理：再帰CTEを使える。
    - 一時的な利用：クエリ内でのみ必要な中間結果を定義するのに適している。

#### 4.5 CTEの応用例

* **複雑な集計クエリの可読性向上:** 複数の集計ステップをCTEで分割し、段階的に結果を構築する。
* **再帰的なデータ処理:** 組織図、部品表、ネットワーク経路など、階層構造を持つデータを処理する。
* **ウィンドウ関数との組み合わせ:** ウィンドウ関数の結果をCTEに格納し、さらに別のウィンドウ関数を適用する。

今回は、CTEについてさらに詳しく解説しました。これで、ウィンドウ関数、ビュー、CTEの3つの機能について、基本的な使い方から応用的な使い方まで、一通りの解説が完了しました。