■SQL文自動作成

シナリオ：社員管理システム

**背景**

あなたは株式会社ABCのデータベース管理者です。会社では、社員情報を管理するためにデータベースを利用しています。今回、人事部門から社員情報に関する様々な集計・分析の依頼がありました。あなたは、これらの依頼に応えるために、SQLを用いて必要な情報を抽出・加工する必要があります。

**テーブル定義**

社員管理データベースには、以下のテーブルが定義されています。

1. **社員テーブル (従業員)**
   * 社員ID (employee\_id)：INTEGER, 主キー
   * 社員名 (employee\_name)：VARCHAR(255), 必須
   * 部署ID (department\_id)：INTEGER, 外部キー（部署テーブルを参照）
   * 役職 (position)：VARCHAR(255)
   * 入社日 (hire\_date)：DATE
   * 給与 (salary)：INTEGER
2. **部署テーブル (部署)**
   * 部署ID (department\_id)：INTEGER, 主キー
   * 部署名 (department\_name)：VARCHAR(255), 必須
   * 所在地 (location)：VARCHAR(255)

**サンプルデータ**

**従業員テーブル**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **社員ID** | **社員名** | **部署ID** | **役職** | **入社日** | **給与** |
| 1 | 佐藤 太郎 | 101 | 部長 | 2010-04-01 | 800 |
| 2 | 鈴木 花子 | 101 | 課長 | 2015-05-10 | 600 |
| 3 | 高橋 健太 | 102 | マネージャー | 2018-08-20 | 700 |
| 4 | 田中 美咲 | 102 | 一般社員 | 2022-01-15 | 450 |
| 5 | 伊藤 純 | 103 | 一般社員 | 2023-04-01 | 400 |
| 6 | 山本 彩 | 103 | リーダー | 2020-11-01 | 550 |
| 7 | 中村 大輔 | 101 | 一般社員 | 2021-07-01 | 480 |
| 8 | 小林 愛 | 102 | 一般社員 | 2022-09-01 | 460 |
| 9 | 加藤 優 | 103 | 一般社員 | 2023-06-10 | 420 |
| 10 | 吉田 翔太 | 101 | 一般社員 | 2024-02-01 | 500 |

**部署テーブル**

|  |  |  |
| --- | --- | --- |
| **部署ID** | **部署名** | **所在地** |
| 101 | 営業部 | 東京本社 |
| 102 | 開発部 | 大阪支社 |
| 103 | 人事部 | 福岡支社 |

**DBMS:** PostgreSQL (特に指定がないため、標準SQLに準拠した記述を基本とします)

SQL文と解説 (1/5)

**目的:** 全社員の社員ID、社員名、役職、所属部署名を一覧表示する。

SELECT

従業員.社員ID,

従業員.社員名,

従業員.役職,

部署.部署名 AS 所属部署 -- 部署テーブルから部署名を取得し、別名「所属部署」として表示

FROM

従業員

INNER JOIN

部署 ON 従業員.部署ID = 部署.部署ID; -- 従業員テーブルと部署テーブルを部署IDで結合

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

**解説:**

* **SELECT句:** 従業員.社員ID, 従業員.社員名, 従業員.役職 に加えて、部署.部署名 AS 所属部署 を指定しています。
  + AS 所属部署 は、**別名（エイリアス）** を付与する構文です。カラム名 部署名 を 所属部署 という日本語名に置き換えることで、結果の見やすさを向上させています。
* **FROM句:** 従業員 テーブルと INNER JOIN 部署 ON 従業員.部署ID = 部署.部署ID を指定しています。
  + INNER JOIN は、**内部結合** を行うキーワードです。従業員 テーブルと 部署 テーブルを、共通カラムである 部署ID で結合しています。
  + ON 従業員.部署ID = 部署.部署ID は、**結合条件** を指定しています。従業員 テーブルの 部署ID と 部署 テーブルの 部署ID が一致する行同士を結合します。
* このSQL文は、社員テーブルと部署テーブルを結合し、各社員の基本情報と所属部署名を関連付けて表示します。

**注意点:**

* **テーブル名のエイリアス:** テーブル名にも別名（エイリアス）を付与することで、SQL文をさらに簡潔に記述できます。例えば、FROM 従業員 AS emp INNER JOIN 部署 AS dept ON emp.部署ID = dept.部署ID のように記述できます。
* **結合の種類:** 今回は INNER JOIN を使用しましたが、結合の種類（LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN など）を適切に使い分けることが重要です。データの要件に合わせて最適な結合方法を選択する必要があります。

**応用:**

* 特定の部署の社員のみを表示したい場合は、WHERE句を追記して部署IDで絞り込むことができます。
* 役職で絞り込んだり、特定の役職を除外したりするなど、WHERE句の条件を変更することで様々な条件で社員情報を抽出できます。
* ORDER BY句を追加して、結果を社員ID順や部署名順に並び替えることができます。

SQL文と解説 (2/5)

**目的:** 役職が「一般社員」の社員の数を部署別に集計し、部署名と社員数を表示する。

SELECT

部署.部署名,

COUNT(従業員.社員ID) AS 一般社員数 -- 部署ごとの一般社員数をカウントし、別名「一般社員数」として表示

FROM

従業員

INNER JOIN

部署 ON 従業員.部署ID = 部署.部署ID

WHERE

従業員.役職 = '一般社員' -- 役職が「一般社員」の社員に絞り込む

GROUP BY

部署.部署名; -- 部署名でグループ化

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

**解説:**

* **SELECT句:** 部署.部署名 と COUNT(従業員.社員ID) AS 一般社員数 を指定しています。
  + COUNT(従業員.社員ID) は、**集約関数** の一つで、従業員.社員ID の**件数**をカウントします。ここでは、部署ごとに集計される一般社員の数を数えています。
  + AS 一般社員数 は、カウント結果に別名 一般社員数 を付与しています。
* **FROM句, INNER JOIN, ON句:** (1)と同様に、従業員 テーブルと 部署 テーブルを INNER JOIN で結合しています。
* **WHERE句:** 従業員.役職 = '一般社員' という条件を指定し、**役職が「一般社員」の社員のみ** を集計対象としています。
* **GROUP BY句:** 部署.部署名 を指定することで、**部署名ごとに集計** を行います。GROUP BY句に指定されたカラム（ここでは 部署名）が同じ値を持つ行がグループ化され、グループごとに集約関数（ここでは COUNT）が適用されます。

**注意点:**

* **集約関数とGROUP BY句:** GROUP BY句を使用する場合、SELECT句には、GROUP BY句に指定したカラムと集約関数のみを指定できます。それ以外のカラムをSELECT句に含めると、SQLエラーが発生します。
* **WHERE句とGROUP BY句の順序:** WHERE句は GROUP BY句よりも前に記述します。WHERE句で集計対象の行を絞り込んだ後、GROUP BY句でグループ化と集計を行います。

**応用:**

* 役職の種類を '課長', '部長' などに変更することで、他の役職の社員数を部署別に集計できます。
* COUNT(\*) を使用すると、WHERE句で絞り込んだ条件に合致する行数を単純にカウントできます。
* AVG(従業員.給与) や SUM(従業員.給与) などの他の集約関数を用いることで、部署別の平均給与や給与合計などを算出できます。

SQL文と解説 (3/5)

**目的:** 部署ごとの平均給与を算出し、平均給与が高い順に部署名と平均給与を表示する。ただし、平均給与が550以上の部署のみを表示する。

SELECT

部署.部署名,

AVG(従業員.給与) AS 平均給与 -- 部署ごとの平均給与を算出し、別名「平均給与」として表示

FROM

従業員

INNER JOIN

部署 ON 従業員.部署ID = 部署.部署ID

GROUP BY

部署.部署名 -- 部署名でグループ化

-- HAVING句は、GROUP BY句によるグループ化後の結果に対して条件を指定する

HAVING AVG(従業員.給与) >= 550 -- 平均給与が550以上の部署に絞り込む

ORDER BY

平均給与 DESC; -- 平均給与が高い順に並び替え

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

**解説:**

* **SELECT句, FROM句, INNER JOIN, ON句, GROUP BY句:** (2)と同様です。
* **HAVING句:** HAVING AVG(従業員.給与) >= 550 を指定しています。
  + **HAVING句** は、GROUP BY句によってグループ化された結果（ここでは部署ごとの平均給与）に対して**条件を指定** する際に使用します。
  + WHERE句は行単位で条件を絞り込むのに対し、HAVING句はグループ化された結果に対して条件を絞り込む点が異なります。
  + ここでは、平均給与が550以上の部署のみを表示するよう指定しています。
* **ORDER BY句:** ORDER BY 平均給与 DESC を指定しています。
  + ORDER BY句は、**結果を並び替える** 際に使用します。
  + DESC は **降順** を意味し、平均給与が高い順に結果を表示します。ASC （昇順）を指定することも可能です（省略時は ASC がデフォルト）。

**注意点:**

* **WHERE句とHAVING句の使い分け:** WHERE句はグループ化**前** の行を絞り込むのに対し、HAVING句はグループ化**後** のグループに対して条件を適用します。集約関数の結果で条件を指定する場合は、必ずHAVING句を使用する必要があります。WHERE句 に集約関数を含めることはできません。
* **ORDER BY句の複数カラム指定:** ORDER BY句には、複数のカラムを指定できます。例えば、ORDER BY 平均給与 DESC, 部署名 ASC のように指定すると、平均給与で降順に並び替え、平均給与が同じ場合は部署名で昇順に並び替えます。

**応用:**

* HAVING句の条件を変更することで、平均給与の範囲を指定したり、平均給与が低い部署を抽出したりできます。
* ORDER BY句の並び替えの基準を平均給与以外に変更したり、昇順・降順を切り替えたりすることで、様々な順序で結果を表示できます。
* LIMIT句を追加して、表示する部署数を制限できます（例：平均給与が高い上位3部署のみを表示する）。

SQL文と解説 (4/5)

**目的:** 特定の部署（例えば「営業部」）に所属する社員の社員名と役職を表示する。

SELECT

従業員.社員名,

従業員.役職

FROM

従業員

WHERE

従業員.部署ID = (

SELECT 部署ID -- 副問い合わせ（サブクエリ）: 部署名が「営業部」の部署IDを取得

FROM 部署

WHERE 部署名 = '営業部'

); -- WHERE句で、副問い合わせで取得した部署IDと一致する社員を絞り込む

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

**解説:**

* **副問い合わせ (サブクエリ):** WHERE句の条件として、SELECT 部署ID FROM 部署 WHERE 部署名 = '営業部' というSQL文が記述されています。これが **副問い合わせ** （または **サブクエリ**）と呼ばれるものです。
  + 副問い合わせは、**別のSELECT文の中に記述されたSELECT文** です。
  + 副問い合わせは、メインのクエリ（外側のクエリ）の WHERE句, FROM句, HAVING句 など様々な場所で使用できます。
  + ここでは、副問い合わせは WHERE句の条件として使用されており、「**部署名が『営業部』の部署IDを取得する**」という処理を行っています。
* **メインクエリ:** 外側の SELECT 文が **メインクエリ** です。
  + メインクエリは、副問い合わせの結果を利用して、最終的に出力する社員名と役職を取得します。
  + WHERE 従業員.部署ID = (...) の (...) 部分に副問い合わせの結果（営業部の部署ID）が入り、**「従業員テーブルの部署IDが、副問い合わせで取得した部署IDと一致する社員」** という条件で社員を絞り込んでいます。

**注意点:**

* **副問い合わせの種類:** 副問い合わせには、**スカラサブクエリ**、**行サブクエリ**、**テーブルサブクエリ** など、いくつかの種類があります。
  + **スカラサブクエリ:** 単一の値（1行1列）を返す副問い合わせ。今回の例はスカラサブクエリです。スカラサブクエリは、WHERE句の比較条件や SELECT句の式の中で使用できます。
  + **行サブクエリ:** 単一行**複数列**、または複数行**単一列**を返す副問い合わせ。IN, EXISTS などと共に使用します。
  + **テーブルサブクエリ:** 複数行**複数列**のテーブルを返す副問い合わせ。FROM句の中でテーブルのように使用します。
* **相関サブクエリと非相関サブクエリ:** 副問い合わせには、**相関サブクエリ** と **非相関サブクエリ** があります。
  + **非相関サブクエリ:** 副問い合わせがメインクエリのカラムを参照していないもの。今回の例は非相関サブクエリです。副問い合わせは独立して実行でき、その結果がメインクエリで使用されます。
  + **相関サブクエリ:** 副問い合わせがメインクエリのカラムを参照しているもの。副問い合わせはメインクエリの各行に対して**繰り返し**実行されます。

**応用:**

* 副問い合わせの WHERE 部署名 = '営業部' の条件を変更することで、他の部署に所属する社員を抽出できます。
* IN 演算子を使って、**複数の部署** に所属する社員を抽出することも可能です（例：WHERE 従業員.部署ID IN (SELECT 部署ID FROM 部署 WHERE 部署名 IN ('営業部', '開発部'))）。
* 副問い合わせを FROM句 で使用して、**導出テーブル** を作成し、より複雑な分析を行うこともできます。

SQL文と解説 (5/5)

**目的:** 全ての部署名と、各部署に所属する社員数を表示する。部署に社員が一人も所属していない場合でも、部署名は表示し、社員数は0と表示する。

SELECT

部署.部署名,

COALESCE(COUNT(従業員.社員ID), 0) AS 社員数 -- COALESCE関数で、NULLの場合に0を表示

FROM

部署

-- LEFT JOINを使用することで、部署テーブルの全ての部署名を表示し、

-- 従業員テーブルに該当するデータがない場合はNULLとして結合する

LEFT JOIN

従業員 ON 部署.部署ID = 従業員.部署ID

GROUP BY

部署.部署名; -- 部署名でグループ化

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

**解説:**

* **LEFT JOIN:** FROM 部署 LEFT JOIN 従業員 ON 部署.部署ID = 従業員.部署ID を指定しています。
  + **LEFT JOIN (左外部結合)** は、**左側のテーブル（ここでは 部署 テーブル）の全ての行** を結果に含めます。
  + 右側のテーブル（ここでは 従業員 テーブル）に結合条件に一致する行がある場合は結合し、一致する行がない場合は、右側のテーブルのカラムは NULL 値となります。
  + 今回のケースでは、LEFT JOIN を使用することで、**「部署テーブルの全ての部署名を表示し、各部署に紐づく社員数を集計する」** ことを実現します。社員が一人もいない部署の場合でも、部署名が結果に含まれます。
* **COALESCE関数:** COALESCE(COUNT(従業員.社員ID), 0) AS 社員数 を指定しています。
  + **COALESCE関数** は、**引数リストの中で最初にNULLでない値を返す** 関数です。
  + COUNT(従業員.社員ID) は、部署ごとの社員数をカウントしますが、**社員が一人もいない部署の場合、COUNT関数の結果は NULL になります**。
  + COALESCE(COUNT(従業員.社員ID), 0) とすることで、COUNT関数の結果が NULL の場合に 0 を返し、**社員がいない部署の社員数を0と表示** できます。

**注意点:**

* **外部結合の種類:** 外部結合には、LEFT JOIN の他に RIGHT JOIN (右外部結合), FULL OUTER JOIN (完全外部結合) があります。
  + LEFT JOIN: 左側のテーブルの全行を保持し、右側のテーブルで一致する行を結合。
  + RIGHT JOIN: 右側のテーブルの全行を保持し、左側のテーブルで一致する行を結合。
  + FULL OUTER JOIN: 左右両方のテーブルの全行を保持し、一致する行を結合。
  + どの外部結合を使用するかは、**「どのテーブルの全行を結果に含めたいか」** によって選択します。
* **NULL値の扱い:** 外部結合を使用した場合、結合条件に一致する行がない場合に NULL 値が発生します。COALESCE関数や IFNULL関数（MySQLなど）などを用いて、NULL 値を適切な値に変換することが重要です。

**応用:**

* WHERE句を追加して、特定の条件を満たす部署のみを対象に集計できます（例：WHERE 部署.所在地 = '東京本社'）。
* ORDER BY句を追加して、部署名順や社員数順に結果を並び替えることができます。
* 社員数 の代わりに、SUM(従業員.給与) や AVG(従業員.給与) などの集約関数を用いることで、部署ごとの給与合計や平均給与などを、社員がいない部署も含めて算出できます。

ご不明な点や、さらに詳しく知りたい構文、応用例などございましたら、お気軽にご質問ください。

はい、承知いたしました。引き続き、社員管理システムを題材としたSQL文とその解説を作成します。

SQL文と解説 (6/10)

**目的:** 部署ごとに社員の給与額が高い順に順位を付け、部署名、社員名、役職、給与、部署内での給与順位を表示する。

SELECT

部署.部署名,

従業員.社員名,

従業員.役職,

従業員.給与,

-- ROW\_NUMBER() OVER (PARTITION BY 部署.部署名 ORDER BY 従業員.給与 DESC)

-- ウィンドウ関数：部署名でパーティション分割し、給与の降順で順位付け

ROW\_NUMBER() OVER (PARTITION BY 部署.部署名 ORDER BY 従業員.給与 DESC) AS 部署内給与順位

FROM

従業員

INNER JOIN

部署 ON 従業員.部署ID = 部署.部署ID

ORDER BY

部署.部署名, -- 部署名でソート

部署内給与順位; -- 部署内給与順位でソート

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

**解説:**

* **ウィンドウ関数 ROW\_NUMBER():** ROW\_NUMBER() OVER (PARTITION BY 部署.部署名 ORDER BY 従業員.給与 DESC) が **ウィンドウ関数** です。
  + **OVER句**: ウィンドウ関数は、OVER句 を伴って使用されます。OVER句 は、ウィンドウ関数が適用される **ウィンドウ（窓）** の範囲と、**順序** を定義します。
  + **PARTITION BY 部署.部署名**: PARTITION BY は、**パーティション分割** を指定します。ここでは、部署.部署名 を指定しているので、**部署ごとに** ウィンドウを分割します。つまり、部署ごとに独立したランキングが計算されます。
  + **ORDER BY 従業員.給与 DESC**: ORDER BY は、ウィンドウ内での **順序** を指定します。ここでは、従業員.給与 DESC を指定しているので、**給与の降順** で順位が付けられます。
  + **ROW\_NUMBER()**: ウィンドウ関数の一種で、**ウィンドウ内で一意の連番** を割り振ります。ここでは、部署ごとの給与ランキングを生成するために使用しています。
  + AS 部署内給与順位: ROW\_NUMBER() 関数の結果に別名 部署内給与順位 を付与しています。
* **ORDER BY 部署.部署名, 部署内給与順位**: 結果を 部署名 と 部署内給与順位 でソートしています。まず部署名で昇順にソートし、同じ部署内では給与順位で昇順にソートします。

**注意点:**

* **ウィンドウ関数の種類:** ROW\_NUMBER() 以外にも、様々なウィンドウ関数があります。
  + **ランキング関数**: RANK(), DENSE\_RANK(), ROW\_NUMBER() など（順位付け）
  + **集約関数**: SUM(), AVG(), COUNT(), MIN(), MAX() など（ウィンドウ内での集計）
  + **値関数**: LAG(), LEAD(), FIRST\_VALUE(), LAST\_VALUE() など（ウィンドウ内の前後の行や最初/最後の行の値を取得）
  + **分布関数**: CUME\_DIST(), PERCENT\_RANK(), NTILE() など（累積分布、パーセント順位、分割）
  + **PARTITION BY の有無**: PARTITION BY 句を省略した場合、**テーブル全体** が一つのウィンドウとなり、テーブル全体でのランキングや集計が行われます。
* **同順位の扱い**: RANK() と DENSE\_RANK() は同順位を扱うランキング関数です。ROW\_NUMBER() は同順位でも異なる連番を割り振ります。
  + RANK(): 同順位の場合、同じ順位を付与し、次の順位は同順位の数だけスキップします (例: 1位, 1位, 3位...)。
  + DENSE\_RANK(): 同順位の場合、同じ順位を付与し、次の順位はスキップしません (例: 1位, 1位, 2位...)。

**応用:**

* ORDER BY 従業員.給与 DESC を ORDER BY 従業員.入社日 ASC に変更することで、部署ごとの入社日順ランキングを作成できます。
* PARTITION BY 句を省略すると、全社員の給与ランキングを作成できます。
* RANK() や DENSE\_RANK() を使用して、同順位を考慮したランキングを作成できます。
* WHERE句を追加して、特定の部署や役職の社員のみを対象としたランキングを作成できます。

SQL文と解説 (7/10)

**目的:** 各部署の平均給与と、全社の平均給与を同時に表示する。部署名、部署別平均給与、全社平均給与を表示する。

SELECT DISTINCT -- DISTINCT を使用して、部署名が重複しないようにする

部署.部署名,

AVG(従業員.給与) OVER (PARTITION BY 部署.部署名) AS 部署別平均給与, -- ウィンドウ関数で部署別平均給与を算出

AVG(従業員.給与) OVER () AS 全社平均給与 -- ウィンドウ関数で全社平均給与を算出

FROM

従業員

INNER JOIN

部署 ON 従業員.部署ID = 部署.部署ID;

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

**解説:**

* **ウィンドウ関数 (集約関数)**: AVG(従業員.給与) OVER (PARTITION BY 部署.部署名) と AVG(従業員.給与) OVER () は、**集約関数** をウィンドウ関数として使用しています。
  + **AVG(従業員.給与) OVER (PARTITION BY 部署.部署名)**: PARTITION BY 部署.部署名 を指定しているので、**部署ごとに** 平均給与を計算します。ウィンドウ関数として使用することで、**各行にその部署の平均給与を付与** できます。
  + **AVG(従業員.給与) OVER ()**: PARTITION BY 句を **省略** しています。この場合、**テーブル全体** が一つのウィンドウとなり、**全社の平均給与** が計算されます。ウィンドウ関数として使用することで、**各行に全社平均給与を付与** できます。
* **DISTINCT**: SELECT DISTINCT 部署.部署名, ... **DISTINCT** キーワードを使用しています。
  + DISTINCT は、**SELECT句で指定されたカラムの組み合わせ** で重複する行を **1行にまとめる** キーワードです。
  + このクエリでは、ウィンドウ関数によって各行に部署別平均給与と全社平均給与が付与されるため、GROUP BY句を使用しなくても部署別の平均給与を表示できます。
  + ただし、INNER JOIN を使用しているため、社員数が多い部署ほど同じ部署名の行が重複して表示されます。DISTINCT を使用することで、**部署名を1行にまとめ、重複を解消** しています。

**注意点:**

* **GROUP BY vs DISTINCT + ウィンドウ関数**: 部署別の集計を行う場合、通常は GROUP BY句を使用しますが、ウィンドウ関数と DISTINCT を組み合わせることで、GROUP BY句を使わずに同様の結果を得ることができます。
  + GROUP BY句: グループごとに1行の結果を生成する（集約処理）。
  + DISTINCT + ウィンドウ関数: 各行にウィンドウ関数結果を付与し、DISTINCT で重複行を削除する（行ごとの処理）。
  + どちらの方法を使うかは、要件やパフォーマンス、可読性などを考慮して選択します。一般的に、より複雑な分析や、集計結果と元の行データを同時に表示したい場合は、ウィンドウ関数が有効です。
* **ウィンドウ関数のパフォーマンス**: ウィンドウ関数は、大量のデータに対して実行する場合、パフォーマンスに影響を与える可能性があります。適切なインデックス設定やクエリの最適化が必要です。

**応用:**

* AVG 関数を SUM, COUNT, MAX, MIN など他の集約関数に変更することで、部署別および全社の合計、件数、最大値、最小値などを同時に表示できます。
* PARTITION BY 句に複数のカラムを指定することで、より詳細なグループごとの集計が可能です（例：PARTITION BY 部署.部署名, 役職 で、部署と役職ごとの平均給与を算出）。
* WHERE句を追加して、特定の条件を満たす社員のみを対象とした平均給与を算出できます。

SQL文と解説 (8/10)

**目的:** 営業部または人事部に所属する社員の社員名と部署名を一覧表示する。

-- UNION 演算子を使用

SELECT 社員名, 部署名

FROM 従業員

INNER JOIN 部署 ON 従業員.部署ID = 部署.部署ID

WHERE 部署.部署名 = '営業部'

UNION -- UNION は結果セットの和集合を返す（重複行を削除）

SELECT 社員名, 部署名

FROM 従業員

INNER JOIN 部署 ON 従業員.部署ID = 部署.部署ID

WHERE 部署.部署名 = '人事部';

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

**解説:**

* **UNION 演算子**: **UNION** は **集合演算子** の一つで、**複数の SELECT 文の結果セットの和集合** を計算します。
  + UNION は、**結果セットの重複行を削除** します。重複行を保持したい場合は、**UNION ALL** を使用します。
  + UNION で結合する複数の SELECT 文は、**SELECT句のカラム数とデータ型が一致している** 必要があります。
  + ここでは、
    1. SELECT 社員名, 部署名 ... WHERE 部署.部署名 = '営業部' (営業部の社員を抽出)
    2. UNION
    3. SELECT 社員名, 部署名 ... WHERE 部署.部署名 = '人事部' (人事部の社員を抽出)  
       という構成で、**「営業部または人事部に所属する社員」** を抽出しています。

**注意点:**

* **集合演算子の種類**: SQLには、UNION の他に以下の集合演算子があります。
  + **UNION ALL**: 和集合を計算しますが、**重複行を削除しません**。UNION よりも高速に処理できる場合があります。
  + **INTERSECT**: **積集合** を計算します。複数の SELECT 文の結果セットに共通する行を抽出します。
  + **EXCEPT (または MINUS)**: **差集合** を計算します。最初の SELECT 文の結果セットに含まれ、2番目以降の SELECT 文の結果セットに含まれない行を抽出します。
  + DBMSによっては、INTERSECT や EXCEPT が INTERSECT ALL, EXCEPT ALL のようにALLオプションをサポートしている場合があります。ALLオプションは重複行の扱いを制御します。
* **WHERE句の条件**: 各 SELECT 文の WHERE句 で抽出条件を個別に指定できます。複雑な条件でデータを抽出する場合に、集合演算子が役立ちます。
* **ORDER BY 句**: UNION, UNION ALL, INTERSECT, EXCEPT の結果セット全体に対して ORDER BY 句を適用する場合は、**最後の SELECT 文の後** に記述します。

**応用:**

* WHERE 部署.部署名 = '営業部' や '人事部' の条件を変更することで、他の部署の社員を抽出したり、複数の部署を組み合わせたりできます。
* UNION の代わりに UNION ALL を使用して、重複行を保持したまま結果を結合できます。
* INTERSECT や EXCEPT を使用して、より複雑な集合演算を行うことができます (例: 「営業部**かつ**役職が一般社員」, 「営業部**だが**東京本社**以外**の勤務地」など)。
* 複数のテーブルを結合した結果セット同士を集合演算することも可能です。

SQL文と解説 (9/10)

**目的:** 人事部に所属する社員の役職を「人事部員」に更新する。

-- UPDATE文：テーブルのデータを更新する

UPDATE 従業員

SET 役職 = '人事部員' -- 更新するカラムと新しい値を指定

WHERE 部署ID = (

SELECT 部署ID -- 副問い合わせ：部署名が「人事部」の部署IDを取得

FROM 部署

WHERE 部署名 = '人事部'

); -- WHERE句で更新対象の行を絞り込む（人事部の社員）

-- 更新後のデータを確認するためのSELECT文（通常はUPDATE文の直後に実行）

SELECT \* FROM 従業員 WHERE 部署ID = (SELECT 部署ID FROM 部署 WHERE 部署名 = '人事部');

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

**解説:**

* **UPDATE 文**: **UPDATE** 文は、**テーブルのデータを更新** するための **データ操作言語 (DML)** です。
  + UPDATE テーブル名: 更新対象のテーブルを指定します。ここでは 従業員 テーブルを指定しています。
  + SET カラム名 = 新しい値: **更新するカラム** と **新しい値** を指定します。複数カラムを更新する場合は、, 区切りで指定します (例: SET 役職 = '人事部員', 給与 = 給与 \* 1.1)。ここでは、役職 カラムを '人事部員' に更新しています。
  + WHERE 条件: **更新対象の行を絞り込む** 条件を指定します。WHERE 句を省略すると、**テーブルの全行が更新される** ため、注意が必要です。ここでは、WHERE 部署ID = (...) で、**人事部に所属する社員** の行のみを更新対象としています。
  + 副問い合わせ: WHERE 部署ID = (...) の (...) 部分は、**副問い合わせ** です。**「部署名が『人事部』の部署IDを取得する」** ために使用しています。
* **確認用 SELECT 文**: UPDATE 文の実行後、**更新が正しく反映されたか** を確認するために、通常は SELECT 文を実行します。ここでは、SELECT \* FROM 従業員 WHERE 部署ID = (SELECT 部署ID FROM 部署 WHERE 部署名 = '人事部') で、人事部の社員のデータを取得し、役職 が '人事部員' に更新されていることを確認します。

**注意点:**

* **WHERE 句の重要性**: UPDATE 文で WHERE 句を省略すると、**テーブルの全行が更新されてしまい、データが破損する** 可能性があります。WHERE 句は必ず指定し、更新対象の行を**慎重に**絞り込む必要があります。
* **トランザクション処理**: UPDATE 文は、データベースの状態を **永続的に変更** します。誤操作によるデータ破損を防ぐため、BEGIN, COMMIT, ROLLBACK などの **トランザクション制御** を利用することが重要です。
  + BEGIN: トランザクションを開始
  + UPDATE ...: データ更新処理
  + COMMIT: トランザクションをコミット (変更を確定)
  + ROLLBACK: トランザクションをロールバック (変更をキャンセル)
  + 例:
  + BEGIN;
  + UPDATE 従業員 SET 役職 = '人事部員' WHERE 部署ID = (SELECT 部署ID FROM 部署 WHERE 部署名 = '人事部');
  + -- ... (更新内容を確認) ...
  + COMMIT; -- 問題なければコミット

-- ROLLBACK; -- 問題があればロールバック

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

* **バックアップ**: UPDATE 文を実行する前に、**テーブルのバックアップ** を取得しておくことを強く推奨します。

**応用:**

* SET 句で複数のカラムを同時に更新できます (例: SET 役職 = '人事部員', 給与 = 450)。
* SET 句で **計算式** や **関数** を使用できます (例: SET 給与 = 給与 \* 1.1, SET 入社日 = CURRENT\_DATE)。
* WHERE 句の条件を複雑にすることで、より詳細な条件で更新対象の行を絞り込むことができます (例: WHERE 部署名 = '営業部' AND 入社日 < '2020-01-01')。
* UPDATE ... FROM ... WHERE ... 構文 (結合更新) を使用すると、複数のテーブルを結合して更新対象の行を特定できます。

SQL文と解説 (10/10)

**目的:** 部署テーブルに新しい部署を追加する (部署ID: 104, 部署名: 企画部, 所在地: 東京本社)。

-- INSERT INTO文：テーブルに新しい行を挿入する

INSERT INTO 部署 (部署ID, 部署名, 所在地) -- 挿入先のテーブルとカラムリストを指定

VALUES (104, '企画部', '東京本社'); -- 挿入する値を指定

-- 挿入後のデータを確認するためのSELECT文（通常はINSERT文の直後に実行）

SELECT \* FROM 部署 WHERE 部署ID = 104;

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

**解説:**

* **INSERT INTO 文**: **INSERT INTO** 文は、**テーブルに新しい行を挿入** するための **データ操作言語 (DML)** です。
  + INSERT INTO テーブル名: **挿入先のテーブル** を指定します。ここでは 部署 テーブルを指定しています。
  + (カラムリスト): **挿入するカラムのリスト** を () で囲んで指定します (例: (部署ID, 部署名, 所在地) )。カラムリストを省略した場合、VALUES 句には **テーブル定義順** に **全てのカラム** の値を指定する必要があります。カラムリストを指定することで、**一部のカラムのみに値を挿入** したり、**カラムの順序を任意に変更** したりできます。
  + VALUES (値リスト): **挿入する値のリスト** を () で囲んで指定します (例: (104, '企画部', '東京本社') )。値リストは、カラムリストで指定したカラムに対応する値を、**同じ順序** で指定します。
* **確認用 SELECT 文**: INSERT INTO 文の実行後、**挿入が正しく行われたか** を確認するために、通常は SELECT 文を実行します。ここでは、SELECT \* FROM 部署 WHERE 部署ID = 104 で、部署IDが104の部署データを取得し、部署名 と 所在地 が正しく挿入されていることを確認します。

**注意点:**

* **制約**: INSERT INTO 文を実行する際、テーブルに設定されている **制約** (例: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK など) に違反すると、エラーが発生し、挿入は失敗します。
  + NOT NULL 制約: NULL 値を挿入しようとするとエラーになります。
  + UNIQUE 制約: 既存の行と重複する値を挿入しようとするとエラーになります。
  + PRIMARY KEY 制約: UNIQUE 制約と NOT NULL 制約の両方を満たす必要があります。
  + FOREIGN KEY 制約: 参照先のテーブルに存在しない値を挿入しようとするとエラーになります。
  + CHECK 制約: 指定した条件式を満たさない値を挿入しようとするとエラーになります。
* **データ型**: VALUES 句で指定する値は、挿入先のカラムの **データ型** に一致している必要があります。データ型が一致しない場合、暗黙的な型変換が行われるか、エラーが発生します。
* **トランザクション処理**: UPDATE 文と同様に、INSERT INTO 文もトランザクション制御 (BEGIN, COMMIT, ROLLBACK) を利用することが重要です。

**応用:**

* VALUES 句に **複数の値リスト** を指定することで、**一度に複数行** を挿入できます (例: VALUES (104, '企画部', '東京本社'), (105, '広報部', '大阪支社'))。
* INSERT INTO ... SELECT ... 構文を使用すると、**別のテーブルからデータを取得して挿入** できます。
* DEFAULT 制約が設定されているカラムは、VALUES 句で値を指定しなくても、**デフォルト値** が自動的に挿入されます。
* ON CONFLICT (PostgreSQL), ON DUPLICATE KEY UPDATE (MySQL) などの構文を使用すると、**重複キー制約違反が発生した場合の処理** (更新または無視) を指定できます。

引き続き、SQLに関する質問やリクエストがありましたら、お気軽にお知らせください。

SQL文と解説 (11/15)

**目的:** 新たに「研修」テーブルを作成する。研修テーブルは、社員が受講した研修の情報を管理する。

-- CREATE TABLE文：データベースに新しいテーブルを作成する

CREATE TABLE 研修 (

研修ID INTEGER PRIMARY KEY, -- 研修ID：主キー、整数型

研修名 VARCHAR(255) NOT NULL, -- 研修名：必須、文字列型

受講日 DATE NOT NULL, -- 受講日：必須、日付型

社員ID INTEGER, -- 社員ID：整数型、外部キー

-- FOREIGN KEY制約：社員IDカラムを従業員テーブルの社員IDカラムと紐付ける

FOREIGN KEY (社員ID) REFERENCES 従業員(社員ID)

);

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

**解説:**

* **CREATE TABLE 文**: **CREATE TABLE** 文は、**データベースに新しいテーブルを作成** するための **データ定義言語 (DDL)** です。
  + CREATE TABLE テーブル名: **作成するテーブルの名前** を指定します。ここでは 研修 テーブルを作成します。
  + (カラム定義リスト): **テーブルのカラム定義** を () で囲んで指定します。各カラム定義は、, 区切りで記述します。
  + **カラム定義**: 各カラム定義は、以下の要素で構成されます。
    - カラム名: カラムの名前 (例: 研修ID, 研修名, 受講日, 社員ID)
    - データ型: カラムに格納するデータの型 (例: INTEGER, VARCHAR(255), DATE)
    - **制約**: カラムに適用する制約 (例: PRIMARY KEY, NOT NULL, FOREIGN KEY)
* **データ型**: 各カラムに適切なデータ型を指定します。
  + INTEGER: 整数型
  + VARCHAR(255): 可変長文字列型 (最大255文字まで)
  + DATE: 日付型
* **制約**: テーブルおよびカラムにデータ制約を設定することで、データの整合性を保ちます。
  + **PRIMARY KEY 制約**: 研修ID INTEGER PRIMARY KEY
    - **主キー制約** は、テーブル内で **一意** かつ **NULL値** を許可しないカラムを指定します。
    - 主キーは、テーブルの行を一意に識別するために使用されます。
    - 通常、各テーブルに1つの主キーを設定します。
  + **NOT NULL 制約**: 研修名 VARCHAR(255) NOT NULL, 受講日 DATE NOT NULL
    - **NOT NULL制約** は、カラムが **NULL値** を許可しないことを指定します。
    - 研修名 と 受講日 は必須項目であるため、NOT NULL 制約を設定しています。
  + **FOREIGN KEY 制約**: FOREIGN KEY (社員ID) REFERENCES 従業員(社員ID)
    - **外部キー制約** は、あるテーブルのカラム (子テーブル) を別のテーブルのカラム (親テーブル) と **紐付ける** 制約です。
    - FOREIGN KEY (社員ID) は、研修 テーブルの 社員ID カラムを外部キーとして定義することを指定します。
    - REFERENCES 従業員(社員ID) は、参照先のテーブルとカラムを指定します。ここでは、従業員 テーブルの 社員ID カラムを参照しています。
    - 外部キー制約を設定することで、**参照整合性** が保たれます。例えば、従業員 テーブルに存在しない 社員ID を 研修 テーブルに登録することはできなくなります。

**注意点:**

* **テーブル名とカラム名**: テーブル名とカラム名は、**データベース内で一意** である必要があります。また、**命名規則** (例: 英数字とアンダースコアを使用、予約語を避けるなど) に従うことが推奨されます。
* **データ型の選択**: カラムに格納するデータの種類と特性に応じて、適切なデータ型を選択することが重要です。データ型を適切に選択することで、**データの整合性**、**ストレージ効率**、**パフォーマンス** を向上させることができます。
* **制約の種類**: PRIMARY KEY, FOREIGN KEY, NOT NULL 以外にも、様々な制約があります (例: UNIQUE, CHECK, DEFAULT など)。テーブルの要件に応じて、適切な制約を設定することが重要です。
* **DBMSによる差異**: データ型や制約の構文、種類、機能は、DBMSによって異なる場合があります。使用するDBMSのドキュメントを確認することが重要です。

**応用:**

* CREATE TABLE AS SELECT 構文を使用すると、**既存のテーブルからデータを選択して新しいテーブルを作成** できます。
* TEMPORARY キーワードを CREATE TABLE 文に付加すると、**一時テーブル** を作成できます。一時テーブルは、セッション中のみ有効なテーブルです。
* CREATE SCHEMA 文を使用すると、**スキーマ** (テーブルやビューなどのデータベースオブジェクトをまとめる名前空間) を作成できます。

SQL文と解説 (12/15)

**目的:** 人事部ユーザーに研修テーブルのSELECT権限とINSERT権限を付与する。

-- GRANT文：データベースオブジェクトに対する権限を付与する

GRANT SELECT, INSERT -- 付与する権限リスト（SELECT, INSERT権限）

ON 研修 -- 権限を付与する対象オブジェクト（研修テーブル）

TO 人事部ユーザー; -- 権限を付与するユーザーまたはロール（人事部ユーザー）

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

**解説:**

* **GRANT 文**: **GRANT** 文は、**データベースオブジェクトに対する権限をユーザーまたはロールに付与** するための **データ制御言語 (DCL)** です。
  + GRANT 権限リスト: **付与する権限の種類** を指定します。複数権限を付与する場合は、, 区切りで指定します。
    - SELECT: テーブルのデータを **参照** する権限 (SELECT文を実行できる権限)
    - INSERT: テーブルにデータを **挿入** する権限 (INSERT文を実行できる権限)
    - UPDATE: テーブルのデータを **更新** する権限 (UPDATE文を実行できる権限)
    - DELETE: テーブルのデータを **削除** する権限 (DELETE文を実行できる権限)
    - ALL PRIVILEGES: **全ての権限** を付与する
  + ON オブジェクト名: **権限を付与する対象のデータベースオブジェクト** を指定します。
    - テーブル名 (例: 研修)
    - ビュー名
    - シーケンス名
    - 関数名
    - スキーマ名
    - データベース名
    - ALL TABLES IN SCHEMA スキーマ名: 特定スキーマ内の全てのテーブル
    - ALL FUNCTIONS IN SCHEMA スキーマ名: 特定スキーマ内の全ての関数
    - ALL SEQUENCES IN SCHEMA スキーマ名: 特定スキーマ内の全てのシーケンス
  + TO ユーザーまたはロール名: **権限を付与する対象のユーザーまたはロール** を指定します。
    - ユーザー名 (例: 人事部ユーザー)
    - ロール名 (例: 人事部)
    - PUBLIC: **全てのユーザー** (パブリックロール)

**注意点:**

* **権限の種類**: GRANT 文で付与できる権限の種類は、DBMSによって異なります。代表的な権限としては、SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, ALTER, EXECUTE, USAGE などがあります。
* **ロール**: **ロール** は、**複数の権限をまとめたもの** です。ロールを作成し、ロールに権限を付与した後、ユーザーにロールを付与することで、効率的に権限管理を行うことができます。
* **REVOKE 文**: **REVOKE** 文は、**GRANT 文で付与した権限を取り消す** ために使用します。REVOKE 文の構文は GRANT 文とほぼ同じです (例: REVOKE SELECT, INSERT ON 研修 FROM 人事部ユーザー;)。
* **セキュリティ**: データベースのセキュリティを確保するためには、**必要最小限の権限** のみをユーザーに付与することが重要です。不要な権限は付与しないように注意する必要があります。

**応用:**

* GRANT ALL PRIVILEGES ON データベース名 TO データベース管理者; のように、データベース管理者ユーザーに全ての権限を付与できます。
* CREATE ROLE ロール名; GRANT SELECT ON 従業員 TO ロール名; GRANT ロール名 TO ユーザー名; のように、ロールを作成し、ロール経由で権限を付与できます。
* WITH GRANT OPTION 句を GRANT 文に付加すると、権限を付与されたユーザーが **さらに他のユーザーに同じ権限を付与** できるようになります (例: GRANT SELECT ON 研修 TO 人事部ユーザー WITH GRANT OPTION;)。

SQL文と解説 (13/15)

**目的:** 社員テーブルから特定の社員情報を削除するトランザクション処理の例を示す。

-- トランザクション開始：BEGIN文

BEGIN;

-- DELETE文：社員IDが10の社員情報を削除

DELETE FROM 従業員 WHERE 社員ID = 10;

-- 削除が成功したか確認するためのSELECT文

SELECT \* FROM 従業員 WHERE 社員ID = 10;

-- 問題なければコミット（変更を確定）：COMMIT文

COMMIT;

-- ロールバックする場合（変更をキャンセル）：ROLLBACK文

-- ROLLBACK;

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

**解説:**

* **トランザクション**: **トランザクション** は、**一連のSQL操作をまとめて1つの処理単位** とする概念です。トランザクション処理を利用することで、**データの整合性** を保証することができます。
* **BEGIN 文**: **BEGIN** 文は、**トランザクションを開始** します。BEGIN 文を実行すると、それ以降のSQL操作は全て **同じトランザクション内** で実行されます。
* **DELETE 文**: **DELETE FROM 従業員 WHERE 社員ID = 10** は、**社員テーブルから社員IDが10の社員情報を削除** する **データ操作言語 (DML)** です。(9/10)の解説を参照してください。
* **確認用 SELECT 文**: DELETE 文の実行後、**削除が正しく行われたか** を確認するために、SELECT \* FROM 従業員 WHERE 社員ID = 10 を実行しています。削除が成功していれば、この SELECT 文は **何も行を返しません**。
* **COMMIT 文**: **COMMIT** 文は、**トランザクションをコミット (確定)** します。COMMIT 文を実行すると、トランザクション内で行われた全てのSQL操作が **データベースに永続的に反映** されます。
* **ROLLBACK 文**: **ROLLBACK** 文は、**トランザクションをロールバック (取消)** します。ROLLBACK 文を実行すると、トランザクション内で行われた全てのSQL操作が **なかったことに** され、データベースは **トランザクション開始前の状態** に戻ります。
  + ROLLBACK は、**エラーが発生した場合** や、**意図的に変更を取り消したい場合** に使用します。
  + 上記の例では、ROLLBACK 文はコメントアウトされています。実際にロールバックする場合は、COMMIT; をコメントアウトし、ROLLBACK; のコメントアウトを解除して実行します。

**注意点:**

* **ACID特性**: トランザクションは、**ACID特性** (Atomicity, Consistency, Isolation, Durability) を満たすことが求められます。
  + **Atomicity (原子性)**: トランザクション内の全てのSQL操作が **成功するか、全て失敗する** かのどちらかであること (中途半端な状態にならない)。
  + **Consistency (一貫性)**: トランザクション開始前とトランザクション終了後で、データベースの **整合性が保たれている** こと (制約違反などが起こらない)。
  + **Isolation (独立性)**: **複数のトランザクションが同時に実行** されても、**互いに干渉しない** こと (トランザクション間の処理が分離されている)。
  + **Durability (永続性)**: コミットされたトランザクションの結果は、**永続的にデータベースに保存** されること (システム障害などが発生してもデータが失われない)。
* **暗黙的コミット**: DBMSによっては、DDL文 (例: CREATE TABLE, ALTER TABLE, DROP TABLE) や DCL文 (例: GRANT, REVOKE) は、**自動的にコミットされる** (暗黙的コミット) 場合があります。トランザクション制御を行う際は、DBMSの仕様を確認する必要があります。
* **トランザクション分離レベル**: トランザクションの **独立性** のレベル (トランザクション分離レベル) は、設定によって変更できます。代表的なトランザクション分離レベルとしては、**Read Uncommitted, Read Committed, Repeatable Read, Serializable** があります。トランザクション分離レベルを高くするほど、データの整合性は高まりますが、同時実行性能は低下する傾向があります。

**応用:**

* 複数の INSERT, UPDATE, DELETE 文を BEGIN ～ COMMIT または ROLLBACK で囲むことで、一連のデータ操作をまとめてトランザクションとして扱うことができます。
* **SAVEPOINT**: トランザクション中に **セーブポイント** を設定し、ROLLBACK TO SAVEPOINT セーブポイント名; を実行することで、トランザクション全体をロールバックするのではなく、**セーブポイントまでロールバック** できます。
* **ネストしたトランザクション**: 一部のDBMSでは、**トランザクションの中にトランザクションをネスト** させることができます (ネストしたトランザクション)。

SQL文と解説 (14/15)

**目的:** 部署テーブルの所在地カラムにNOT NULL制約を追加する。

-- ALTER TABLE文：既存のテーブル定義を変更する

ALTER TABLE 部署

ADD CONSTRAINT 部署\_所在地\_NN -- 制約名（任意）

CHECK (所在地 IS NOT NULL); -- CHECK制約：所在地カラムがNULLでないことを制約

-- 制約が追加されたか確認するためのテーブル定義確認 (DBMSによって構文は異なる場合があります)

-- PostgreSQLの場合：

-- \d 部署

-- MySQLの場合：

-- DESCRIBE 部署;

-- SQL Serverの場合：

-- sp\_help 部署;

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

**解説:**

* **ALTER TABLE 文**: **ALTER TABLE** 文は、**既存のテーブルの定義を変更** するための **データ定義言語 (DDL)** です。
  + ALTER TABLE テーブル名: **変更対象のテーブル** を指定します。ここでは 部署 テーブルを指定しています。
  + ADD CONSTRAINT 制約名 制約定義: **新しい制約を追加** します。
    - ADD COLUMN カラム定義: 新しいカラムを追加する (例: ADD COLUMN 電話番号 VARCHAR(20))
    - ALTER COLUMN カラム名 データ型: カラムのデータ型を変更する (例: ALTER COLUMN 社員名 VARCHAR(100))
    - DROP COLUMN カラム名: カラムを削除する (例: DROP COLUMN 電話番号)
    - ADD CONSTRAINT 制約名 制約定義: 新しい制約を追加する (今回の例)
    - DROP CONSTRAINT 制約名: 制約を削除する (例: DROP CONSTRAINT 部署\_所在地\_NN)
    - RENAME TO 新しいテーブル名: テーブル名を変更する (例: RENAME TO 部門)
    - RENAME COLUMN 古いカラム名 TO 新しいカラム名: カラム名を変更する (例: RENAME COLUMN 部署名 TO 部門名)
* **ADD CONSTRAINT 部署\_所在地\_NN CHECK (所在地 IS NOT NULL)**: **ADD CONSTRAINT** 句で、新しい制約を追加しています。
  + 部署\_所在地\_NN: **制約名** を指定します。制約名は任意ですが、**分かりやすい名前** を付けることが推奨されます。制約名を指定しない場合、DBMSが自動的に制約名を生成します。
  + CHECK (所在地 IS NOT NULL): **CHECK 制約** を定義しています。
    - **CHECK 制約** は、カラムの値が **指定した条件式を満たす** ことを制約します。
    - 所在地 IS NOT NULL: **所在地カラムが NULL でない** ことを条件とする条件式です。
    - この CHECK 制約を追加することで、**部署テーブルの所在地カラムに NULL 値を登録できなくなります** (NOT NULL制約と同様の効果)。
* **テーブル定義確認**: ALTER TABLE 文実行後、**制約が正しく追加されたか** を確認するために、テーブル定義を確認します。テーブル定義を確認する方法は、DBMSによって異なります。
  + PostgreSQL: \d テーブル名 (psqlコマンド)
  + MySQL: DESCRIBE テーブル名; (mysqlコマンド)
  + SQL Server: sp\_help テーブル名; (SQL Server Management Studio)
  + Oracle: DESCRIBE テーブル名; (SQL\*Plus)
  + これらのコマンドを実行すると、テーブルのカラム定義、データ型、制約などが一覧表示されます。

**注意点:**

* **制約の種類**: ALTER TABLE 文で追加できる制約の種類は、CHECK 制約以外にも、PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL などがあります。ただし、NOT NULL 制約を追加する場合は、ALTER COLUMN カラム名 SET NOT NULL のように、ALTER COLUMN 句を使用するDBMSもあります。
* **制約の削除**: ALTER TABLE テーブル名 DROP CONSTRAINT 制約名; で、**制約を削除** できます。
* **データ型変更**: ALTER TABLE 文でカラムのデータ型を変更する場合、**データ型の互換性** に注意が必要です。データ型の変更によって、既存のデータが失われたり、データ型変換エラーが発生したりする可能性があります。
* **排他制御**: ALTER TABLE 文は、テーブル構造を変更するため、**排他制御** が必要となる場合があります。テーブルにアクセスしている他のセッションがあると、ALTER TABLE 文の実行がブロックされたり、デッドロックが発生したりする可能性があります。

**応用:**

* ALTER TABLE 文を応用することで、テーブルに **新しいカラムを追加** したり、**不要なカラムを削除** したり、**カラム名を変更** したり、**テーブル名を変更** したりできます。
* DEFAULT 制約を追加して、カラムに **デフォルト値** を設定できます (例: ALTER TABLE 従業員 ALTER COLUMN 入社日 SET DEFAULT CURRENT\_DATE;)。
* UNIQUE 制約を追加して、カラムに **一意制約** を設定できます (例: ALTER TABLE 部署 ADD CONSTRAINT 部署\_部署名\_UQ UNIQUE (部署名);)。

SQL文と解説 (15/15)

**目的:** 従業員テーブルの社員名を検索するためのインデックスを作成する。

-- CREATE INDEX文：テーブルにインデックスを作成する

CREATE INDEX 社員名インデックス -- インデックス名（任意）

ON 従業員 (社員名); -- インデックスを作成するテーブルとカラムを指定

-- インデックスが作成されたか確認するためのインデックス一覧表示 (DBMSによって構文は異なる場合があります)

-- PostgreSQLの場合：

-- \d 従業員

-- MySQLの場合：

-- SHOW INDEX FROM 従業員;

-- SQL Serverの場合：

-- sp\_helpindex 従業員;

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

**解説:**

* **インデックス**: **インデックス** は、**テーブルの特定のカラムに対する検索を高速化** するためのデータベースオブジェクトです。インデックスは、書籍の索引のようなもので、特定のキーワードから目的の情報を素早く見つけ出すのに役立ちます。
* **CREATE INDEX 文**: **CREATE INDEX** 文は、**テーブルに新しいインデックスを作成** するための **データ定義言語 (DDL)** です。
  + CREATE INDEX インデックス名: **作成するインデックスの名前** を指定します。インデックス名は任意ですが、**分かりやすい名前** を付けることが推奨されます。
  + ON テーブル名 (カラムリスト): **インデックスを作成するテーブル** と **カラムリスト** を指定します。複数カラムにインデックスを作成する場合は、, 区切りでカラム名を指定します (複合インデックス)。
  + USING インデックスタイプ: **インデックスタイプ** を指定します (例: BTREE, HASH, GiST, GIN など)。インデックスタイプはDBMSによって異なります。省略した場合、DBMSがデフォルトのインデックスタイプを使用します。
    - **BTREEインデックス**: **B-Tree** (Balanced Tree) 構造のインデックス。**範囲検索** や **ソート** に強く、多くのDBMSでデフォルトのインデックスタイプとして使用されています。
    - **HASHインデックス**: **ハッシュ関数** を使用したインデックス。**等価検索** (例: WHERE カラム = '値') に非常に高速ですが、**範囲検索** や **ソート** には不向きです。MySQLのInnoDBストレージエンジンでサポートされています。
    - **GiST/GINインデックス**: **Generalized Search Tree (GiST) / Generalized Inverted Index (GIN)** インデックス。**全文検索** や **空間検索** など、特殊なデータ型や検索処理を効率化するために使用されます。PostgreSQLでサポートされています。
* **CREATE INDEX 社員名インデックス ON 従業員 (社員名)**: **社員名インデックス** という名前のインデックスを、**従業員 テーブルの 社員名 カラム** に作成します。インデックスタイプは省略しているので、DBMSのデフォルトのインデックスタイプ (通常はBTREE) が使用されます。
* **インデックス一覧表示**: CREATE INDEX 文実行後、**インデックスが正しく作成されたか** を確認するために、インデックス一覧を表示します。インデックス一覧を表示する方法は、DBMSによって異なります。
  + PostgreSQL: \d テーブル名 (psqlコマンド)
  + MySQL: SHOW INDEX FROM テーブル名; (mysqlコマンド)
  + SQL Server: sp\_helpindex テーブル名; (SQL Server Management Studio)
  + Oracle: インデックス一覧を表示するSQLクエリを実行する必要があります。

**注意点:**

* **インデックスのメリットとデメリット**: インデックスは、**検索性能を向上** させる一方で、**ディスク容量を消費** し、**データ更新処理 (INSERT, UPDATE, DELETE) の性能を低下** させる可能性があります。インデックスを**闇雲に作成すると、かえって性能が悪化する** 場合もあります。
* **インデックスの適切な作成**: インデックスは、**WHERE 句で頻繁に検索条件として使用されるカラム** や、**結合条件として使用されるカラム** に作成するのが効果的です。**カーディナリティ** (カラムの値の種類数) が高いカラム (値の種類が多いカラム) ほど、インデックスの効果が高くなります。
* **複合インデックス**: **複数カラム** にまたがるインデックス (複合インデックス) を作成することもできます。複合インデックスは、**複数カラムを組み合わせた条件** で検索する場合に有効です。複合インデックスを作成する際は、**カラムの順序** が重要です。
* **クラスタインデックス**: **クラスタインデックス** は、**テーブルのデータ自体をインデックスキーの順序で物理的に並び替える** インデックスです。1つのテーブルに **1つだけ** 作成できます。**範囲検索** に非常に強く、**テーブル全体をスキャン** するようなクエリにも有効です。MySQLのInnoDBストレージエンジンのPRIMARY KEY制約は、デフォルトでクラスタインデックスとして作成されます。
* **インデックスの再構築**: データの更新が頻繁に行われるテーブルでは、インデックスの **断片化** が進み、検索性能が低下する

はい、承知いたしました。引き続き、社員管理システムを題材としたSQL文とその解説を作成します。

SQL文と解説 (16/20)

**目的:** 部署名が「営業部」の部署に所属する社員の中で、給与が部署の平均給与よりも高い社員の社員名と給与を表示する。

SELECT

従業員.社員名,

従業員.給与

FROM

従業員

INNER JOIN

部署 ON 従業員.部署ID = 部署.部署ID

WHERE

部署.部署名 = '営業部'

AND 従業員.給与 > (

-- 相関副問い合わせ：部署ごとの平均給与を取得

SELECT AVG(給与)

FROM 従業員 AS 部署内従業員 -- 副問い合わせ内で従業員テーブルに別名付与

WHERE 部署内従業員.部署ID = 従業員.部署ID -- 相関条件：外側のクエリの部署IDと紐付ける

);

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

**解説:**

* **相関副問い合わせ**: WHERE 従業員.給与 > (...) の (...) 部分に記述されている SELECT AVG(給与) ... が **相関副問い合わせ** です。
  + **相関副問い合わせ** は、**副問い合わせが外側のクエリ (メインクエリ) のカラムを参照する** 副問い合わせです。
  + **非相関副問い合わせ** (4/5) では、副問い合わせは独立して実行できましたが、**相関副問い合わせは、メインクエリの各行に対して繰り返し実行** されます。
  + 今回の例では、副問い合わせ SELECT AVG(給与) ... WHERE 部署内従業員.部署ID = 従業員.部署ID は、**「各社員が所属する部署の平均給与を計算する」** 処理を行います。
    - FROM 従業員 AS 部署内従業員: 副問い合わせ内で 従業員 テーブルに **別名 部署内従業員** を付与しています。これは、外側のクエリの 従業員 テーブルと区別するためです。
    - WHERE 部署内従業員.部署ID = 従業員.部署ID: **相関条件** です。部署内従業員.部署ID (副問い合わせ側の従業員テーブルの部署ID) と 従業員.部署ID (外側のクエリ側の従業員テーブルの部署ID) を **紐付ける** ことで、**「外側のクエリで処理している社員と同じ部署IDの社員」** を副問い合わせの集計対象としています。つまり、**部署ごとに平均給与** を計算します。
* **メインクエリ**: 外側の SELECT 文が **メインクエリ** です。
  + WHERE 部署.部署名 = '営業部': **部署名が「営業部」** の社員に絞り込みます。
  + AND 従業員.給与 > (...): さらに、**給与が副問い合わせで計算された部署の平均給与よりも高い** 社員に絞り込みます。

**注意点:**

* **別名 (エイリアス) の重要性**: 相関副問い合わせでは、内側のクエリと外側のクエリで同じテーブルを参照することが多いため、**テーブルに別名を付与して区別する** ことが非常に重要です。別名を付けないと、どのテーブルのカラムを参照しているのかが曖昧になり、SQLエラーが発生したり、意図しない結果になったりする可能性があります。
* **相関条件**: 相関副問い合わせでは、**相関条件** を適切に記述することが重要です。相関条件によって、副問い合わせが外側のクエリのどの行と関連付けられるかが決まります。相関条件が誤っていると、意図した結果が得られません。
* **パフォーマンス**: 相関副問い合わせは、メインクエリの各行に対して繰り返し実行されるため、**非相関副問い合わせよりもパフォーマンスが低い** 傾向があります。大量のデータを処理する場合は、クエリの実行計画を確認したり、インデックスを適切に設定したりするなどの **パフォーマンスチューニング** が必要となる場合があります。

**応用:**

* > 演算子を < や >= などに変更することで、平均給与よりも低い社員や平均給与以上の社員を抽出できます。
* AVG(給与) を MAX(給与) や MIN(給与) など他の集約関数に変更することで、部署の最高給与や最低給与と比較できます。
* 相関副問い合わせを HAVING 句で使用して、**グループ化された結果** に対して条件を指定することもできます (例: 部署ごとの平均給与が全社平均給与よりも高い部署を抽出)。

SQL文と解説 (17/20)

**目的:** 研修を受講したことのある社員の社員名と役職を表示する。

-- EXISTS述語を使用

SELECT

従業員.社員名,

従業員.役職

FROM

従業員

WHERE

EXISTS ( -- EXISTS述語：副問い合わせの結果が1行以上存在するかどうかを判定

SELECT 1 -- 副問い合わせのSELECT句は通常は定数（1や\*）

FROM 研修

WHERE 研修.社員ID = 従業員.社員ID -- 相関条件：外側のクエリの社員IDと紐付ける

);

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

**解説:**

* **EXISTS 述語**: **EXISTS 述語** は、**副問い合わせの結果が1行以上存在するかどうか** を判定する条件です。
  + EXISTS (副問い合わせ): 副問い合わせの結果が **1行以上** 返された場合、EXISTS 述語は **真 (TRUE)** を返します。副問い合わせの結果が **0行** の場合、EXISTS 述語は **偽 (FALSE)** を返します。
  + EXISTS 述語は、**副問い合わせの結果の行の存在** だけをチェックするため、**副問い合わせのSELECT句には通常は定数 (1や \*)** を記述します。副問い合わせで実際にどのようなカラムを選択するかは、EXISTS 述語の判定結果には影響しません。
  + 今回の例では、副問い合わせ SELECT 1 FROM 研修 WHERE 研修.社員ID = 従業員.社員ID は、**「各社員が研修テーブルにレコードを持っているか (研修を受講したことがあるか) をチェックする」** 処理を行います。
    - WHERE 研修.社員ID = 従業員.社員ID: **相関条件** です。研修.社員ID (副問い合わせ側の研修テーブルの社員ID) と 従業員.社員ID (外側のクエリ側の従業員テーブルの社員ID) を **紐付ける** ことで、**「外側のクエリで処理している社員と同じ社員IDを持つ研修レコード」** が存在するかどうかをチェックします。
* **メインクエリ**: 外側の SELECT 文が **メインクエリ** です。
  + WHERE EXISTS (...): **EXISTS 述語が真 (TRUE) を返す** 社員 (研修を受講したことのある社員) のみを抽出します。

**注意点:**

* **IN 述語との比較**: EXISTS 述語と似たような機能を持つ述語として **IN 述語** があります。
  + IN (副問い合わせ): 副問い合わせの結果に含まれる値を持つかどうかを判定します。
  + 一般的に、**EXISTS 述語の方が IN 述語よりもパフォーマンスが良い** と言われています。特に、副問い合わせの結果セットが非常に大きくなる場合は、EXISTS 述語を使用する方が効率的です。
  + IN 述語は、副問い合わせの結果セットの値を具体的に参照する必要がある場合 (例: WHERE カラム IN (SELECT ...) ) に使用します。EXISTS 述語は、副問い合わせの結果の存在だけをチェックしたい場合に使用します。
* **NOT EXISTS 述語**: EXISTS 述語の反対の述語として **NOT EXISTS 述語** があります。
  + NOT EXISTS (副問い合わせ): 副問い合わせの結果が **0行** の場合、NOT EXISTS 述語は **真 (TRUE)** を返します。副問い合わせの結果が **1行以上** の場合、NOT EXISTS 述語は **偽 (FALSE)** を返します。
  + NOT EXISTS 述語を使用すると、**「〜〜でない」** という条件を表現できます (例: 研修を受講したことのない社員を抽出)。

**応用:**

* EXISTS 述語を NOT EXISTS 述語に変更することで、**研修を受講したことのない社員** を抽出できます。
* 副問い合わせの WHERE 句に条件を追加することで、**特定の研修を受講したことのある社員** や、**特定の期間内に研修を受講したことのある社員** を抽出できます。
* EXISTS 述語は、**複数のテーブル間の関連性** をチェックする場合に非常に役立ちます。例えば、「受注テーブルに受注データが存在する顧客のみを抽出する」、「在庫テーブルに在庫がある商品のみを抽出する」などの場合に利用できます。

SQL文と解説 (18/20)

**目的:** 社員名、部署名、役職、給与を表示し、役職がNULLの場合は「役職なし」と表示する。

SELECT

従業員.社員名,

部署.部署名,

-- COALESCE関数：複数の引数の中でNULLでない最初の値を返す

COALESCE(従業員.役職, '役職なし') AS 役職, -- 役職カラムがNULLの場合に「役職なし」を表示

従業員.給与

FROM

従業員

INNER JOIN

部署 ON 従業員.部署ID = 部署.部署ID;

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

**解説:**

* **COALESCE 関数**: **COALESCE 関数** は、**複数の引数の中でNULLでない最初の値を返す** 関数です。
  + COALESCE(引数1, 引数2, 引数3, ...): 引数を左から順に評価し、**最初にNULLでない値が見つかった時点で、その値を返します**。全ての引数がNULLの場合は、NULLを返します。
  + 今回の例では、COALESCE(従業員.役職, '役職なし') は、以下の処理を行います。
    1. 従業員.役職 の値を評価します。
    2. 従業員.役職 が **NULLでない場合** は、**従業員.役職 の値をそのまま返します**。
    3. 従業員.役職 が **NULLの場合** は、**次の引数 '役職なし' を返します**。
    4. AS 役職: COALESCE 関数の結果に別名 役職 を付与しています。
* **NULL 値の扱い**: データベースにおいて、**NULL 値は「値が存在しない」** ことを意味します。NULL 値は、数値、文字列、日付など、あらゆるデータ型で表現できます。
  + NULL 値の判定: IS NULL 述語 (NULLである場合) または IS NOT NULL 述語 (NULLでない場合) を使用します (例: WHERE カラム IS NULL, WHERE カラム IS NOT NULL)。
  + NULL 値の演算: NULL 値を含む演算の結果は、通常は NULL になります (例: 1 + NULL = NULL, 'abc' || NULL = NULL)。
  + 集約関数と NULL 値: 集約関数 (COUNT, SUM, AVG, MIN, MAX など) は、通常は NULL 値を無視します (例: COUNT(\*) は NULL 値を含む全ての行数をカウントしますが、COUNT(カラム名) は NULL 値でない行数のみをカウントします)。

**注意点:**

* **IFNULL 関数 (MySQL), NVL 関数 (Oracle)**: DBMSによっては、COALESCE 関数と同様の機能を持つ関数として、IFNULL 関数 (MySQL), NVL 関数 (Oracle) などが提供されています。これらの関数は、引数の数が2つに限定されていることが多いです (例: IFNULL(カラム名, '代替値'), NVL(カラム名, '代替値'))。
* **CASE 式**: CASE 式を使用しても、COALESCE 関数と同様の処理を実現できます。
* CASE
* WHEN 従業員.役職 IS NULL THEN '役職なし'
* ELSE 従業員.役職

END AS 役職

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

CASE 式は、より複雑な条件分岐を記述できるため、COALESCE 関数では対応できないような、より高度なNULL値のハンドリングが必要な場合に有効です。

**応用:**

* COALESCE 関数の第2引数以降に、さらに複数の代替値を指定できます (例: COALESCE(カラム1, カラム2, カラム3, 'デフォルト値') )。
* COALESCE 関数を UPDATE 文の SET 句で使用して、カラムの値が NULL の場合にのみ更新処理を行うことができます (例: UPDATE テーブル SET カラム = COALESCE(カラム, '初期値') WHERE 条件;)。
* COALESCE 関数は、NULL 値を別の値に置き換えて表示する場合や、NULL 値を考慮した計算処理を行う場合など、様々な場面で活用できます。

SQL文と解説 (19/20)

**目的:** 社員テーブルから、社員ID、社員名、入社日、勤続年数を表示する。勤続年数は、満**年数**で計算する。

SELECT

従業員.社員ID,

従業員.社員名,

従業員.入社日,

-- DATE\_PART関数 (PostgreSQL)：日付/時刻値から指定された部分を取得

DATE\_PART('year', AGE(CURRENT\_DATE, 従業員.入社日)) AS 勤続年数 -- AGE関数で入社日からの経過期間を計算し、DATE\_PART関数で年部分を抽出

FROM

従業員;

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

**解説:**

* **AGE 関数 (PostgreSQL)**: **AGE 関数** は、**2つの日付/時刻値の間の経過期間** を計算する関数です (PostgreSQL, Oracle, SQL Server など多くのDBMSで類似の関数が提供されています)。
  + AGE(日付/時刻値1, 日付/時刻値2): **日付/時刻値1 - 日付/時刻値2** の経過期間を interval 型 (期間を表すデータ型) で返します。
  + 今回の例では、AGE(CURRENT\_DATE, 従業員.入社日) は、**「現在の日付 (CURRENT\_DATE) から入社日 (従業員.入社日) までの経過期間」** を計算します。
* **DATE\_PART 関数 (PostgreSQL)**: **DATE\_PART 関数** は、**日付/時刻値から指定された部分 (年、月、日、時、分、秒など)** を抽出する関数です (PostgreSQL, Oracle, SQL Server など多くのDBMSで類似の関数が提供されています)。
  + DATE\_PART('部分', 日付/時刻値): **日付/時刻値** から **'部分'** で指定された部分を抽出します。
  + 'year': 年
  + 'month': 月
  + 'day': 日
  + 'hour': 時
  + 'minute': 分
  + 'second': 秒
  + 'week': 週
  + 'quarter': 四半期
  + など、様々な部分を指定できます (DBMSによって指定できる部分は異なります)。
  + 今回の例では、DATE\_PART('year', AGE(CURRENT\_DATE, 従業員.入社日)) は、**「AGE 関数で計算された経過期間から、年部分を抽出する」** ことで、**勤続年数 (満年数)** を計算しています。
  + AS 勤続年数: DATE\_PART 関数の結果に別名 勤続年数 を付与しています。

**注意点:**

* **日付/時刻関数**: DBMSは、日付/時刻データを操作するための様々な関数を提供しています。代表的な関数としては、以下のようなものがあります。
  + **日付/時刻値の取得**: CURRENT\_DATE (現在日付), CURRENT\_TIME (現在時刻), CURRENT\_TIMESTAMP (現在日時), NOW() (現在日時 - DBMSによっては関数), SYSDATE (Oracle - 現在日時) など
  + **日付/時刻値の加算/減算**: DATE\_ADD, DATE\_SUB, INTERVAL (期間を表すデータ型 - DBMSによってはキーワード) など
  + **日付/時刻値の差分計算**: DATEDIFF (日付間の日数差), TIMEDIFF (時刻間の時間差), AGE (日付/時刻値間の経過期間) など
  + **日付/時刻値のフォーマット変換**: TO\_CHAR (日付/時刻値を文字列に変換), TO\_DATE (文字列を日付値に変換), TO\_TIMESTAMP (文字列を日時値に変換) など
  + **日付/時刻値の部分抽出**: DATE\_PART, EXTRACT (日付/時刻値から年、月、日などを抽出) など
  + **日付/時刻値の作成**: DATE (日付値を作成), TIME (時刻値を作成), TIMESTAMP (日時値を作成) など
* **DBMSによる差異**: 日付/時刻関数の種類、関数名、構文、引数、戻り値のデータ型などは、DBMSによって異なる場合があります。使用するDBMSのドキュメントを確認することが重要です。

**応用:**

* DATE\_PART('month', AGE(CURRENT\_DATE, 従業員.入社日)) で、**勤続月数** を計算できます。
* DATE\_PART('day', AGE(CURRENT\_DATE, 従業員.入社日)) で、**勤続日数** を計算できます。
* WHERE DATE\_PART('year', AGE(CURRENT\_DATE, 従業員.入社日)) >= 10 のように、WHERE 句で勤続年数による絞り込みを行うことができます (例: 勤続10年以上の社員を抽出)。
* ORDER BY DATE\_PART('year', AGE(CURRENT\_DATE, 従業員.入社日)) DESC のように、ORDER BY 句で勤続年数順に並び替えることができます (例: 勤続年数の長い順に社員を表示)。

SQL文と解説 (20/20)

**目的:** 社員テーブルから、社員ID、社員名、部署名、役職、給与を表示するビューを作成する。

-- CREATE VIEW文：仮想的なテーブルであるビューを作成する

CREATE VIEW 社員情報ビュー AS -- ビュー名（任意）とASキーワードを指定

SELECT

従業員.社員ID,

従業員.社員名,

部署.部署名,

従業員.役職,

従業員.給与

FROM

従業員

INNER JOIN

部署 ON 従業員.部署ID = 部署.部署ID;

-- 作成したビューを参照するSELECT文

SELECT \* FROM 社員情報ビュー;

-- ビュー定義を確認する (DBMSによって構文は異なる場合があります)

-- PostgreSQLの場合：

-- \dv 社員情報ビュー

-- MySQLの場合：

-- SHOW CREATE VIEW 社員情報ビュー;

-- SQL Serverの場合：

-- sp\_helptext 社員情報ビュー;

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

**解説:**

* **ビュー (View)**: **ビュー** は、**仮想的なテーブル** です。ビュー自体はデータを物理的に格納するのではなく、**定義されたSELECT文 (ビュー定義)** を保存します。ビューを参照する SELECT 文が実行されると、ビュー定義の SELECT 文が実行され、その結果がビューの結果として返されます。
  + ビューは、**テーブルと同様にSELECT文で参照** できますが、**INSERT, UPDATE, DELETE などのデータ操作は、ビューの種類や定義によっては制限される** 場合があります (特に複雑な結合や集約関数を含むビュー)。
  + ビューのメリット:
    - **データの抽象化**: 複雑な結合や条件をビュー定義に隠蔽することで、ユーザーは単純な SELECT \* FROM ビュー名 のようなクエリで必要なデータにアクセスできるようになります。
    - **セキュリティ**: ビューを通じてアクセスできるデータを制限することで、**機密性の高いデータへの不正アクセス** を防ぐことができます。例えば、特定のカラムのみを含むビューを作成し、ユーザーにはそのビューへのSELECT権限のみを付与することで、ユーザーがテーブル全体にアクセスすることを防ぐことができます。
    - **再利用性**: ビュー定義を再利用することで、**同じようなクエリを何度も記述する手間** を省くことができます。
    - **保守性**: ビュー定義を変更することで、**複数のクエリをまとめて修正** することができます。
* **CREATE VIEW 文**: **CREATE VIEW** 文は、**ビューを作成** するための **データ定義言語 (DDL)** です。
  + CREATE VIEW ビュー名 AS SELECT文: **作成するビューの名前** と **ビュー定義の SELECT 文** を指定します。
  + 社員情報ビュー: 作成するビューの名前です。
  + AS SELECT ... FROM ... INNER JOIN ...: ビュー定義の SELECT 文です。ここでは、従業員 テーブルと 部署 テーブルを結合し、必要なカラムを選択する SELECT 文を定義しています。
* **ビューの参照**: 作成したビューは、**テーブルと同様に SELECT \* FROM ビュー名 のように参照** できます。上記の例では、SELECT \* FROM 社員情報ビュー; で、作成した 社員情報ビュー を参照しています。
* **ビュー定義の確認**: ビュー定義を確認する方法は、DBMSによって異なります。
  + PostgreSQL: \dv ビュー名 (psqlコマンド)
  + MySQL: SHOW CREATE VIEW ビュー名; (mysqlコマンド)
  + SQL Server: sp\_helptext ビュー名; (SQL Server Management Studio)
  + Oracle: ビュー定義を確認するSQLクエリを実行する必要があります。

**注意点:**

* **ビューの種類**: ビューには、**単純ビュー** (単一テーブルに基づき、集約関数や GROUP BY 句を含まないビュー) や **複雑ビュー** (複数テーブルの結合や集約関数、GROUP BY 句などを含むビュー) など、様々な種類があります。ビューの種類によって、データ操作の可否や性能特性が異なります。
* **マテリアライズドビュー**: **マテリアライズドビュー** は、**ビュー定義の結果を物理的にテーブルとして格納** するビューです。通常のビュー (仮想ビュー) と異なり、データがディスクに格納されるため、参照性能が向上しますが、データ更新時にはビューのデータも更新する必要があるため、更新処理のオーバーヘッドが増加します。マテリアライズドビューは、参照頻度が高く、更新頻度が低いデータに対して有効です。
* **ビューの更新**: ビューの種類や定義によっては、ビューに対して INSERT, UPDATE, DELETE などのデータ操作を行うことができます (**更新可能なビュー**)。ただし、複雑なビュー (例: 結合ビュー、集約ビュー、GROUP BY 句を含むビュー) は、通常は更新できません。

**応用:**

* 特定の部署の社員のみを表示するビューや、特定の役職以上の社員のみを表示するビューなど、様々な条件でデータを抽出したビューを作成できます。
* 複数のビューを組み合わせて、より複雑なデータアクセスを実現できます。
* ビューを他のビューの定義で使用する (ビューのネスト) ことも可能です。

これで、社員管理システムを題材としたSQL文と解説は20個となります。

引き続き、SQLに関する質問やリクエストがありましたら、お気軽にお知らせください。