**## SQL括弧のチートシート（ON句／USING句対応）**

**### 1. 基本的な考え方（迷ったときの即決ルール）**

\*   **\*\*構文的に必要か分からない場合\*\***: `付ける`

    \*   理由: 付けても減点されませんが、省略すると構文エラーになるリスクがあります。

\*   **\*\*リスト (**`IN`**,** `VALUES`**,** `USING`**) の場合\*\***: `必ず付ける`

    \*   理由: リストは「まとまり」を示すため必須です。`USING (項目)` もこの一種と捉えます。

\*   **\*\*式 (**`ON`**,** `AND/OR`**,** `算術演算`**) の場合\*\***: `なくても動くが、曖昧なら付ける`

    \*   理由: 処理の意図を明確にし、誤読を防ぐためです。

\*   **\*\*句や識別子 (**`SELECT`**の列名,** `FROM`**の表名) の場合\*\***: `付けない`

    \*   理由: 括弧を付けると構文として冗長、またはエラーになります。

**### 2. 具体的な状況別の判断（「一瞬の判断」５秒ルール）**

\*   `IN`, `VALUES`, `USING`, `PRIMARY KEY（表レベル）` を見たら:

    \*   → `括弧必須`です。反射的に `()` を付けましょう。

\*   `FROM (SELECT …)` のようにSQL文の中に別のSQL文（サブクエリ）がある場合:

    \*   → `必ず括弧`が必要です。`サブクエリ`は括弧で囲うのがルールです。

\*   `AND` と `OR` が混在している場合:

    \*   → 優先して評価したい条件式を`括弧で囲い`、処理の順序を明示します。

\*   `ON` 句で等価結合を書いている場合:

    \*   → `括弧不要`です（例: `ON A.ID = B.ID`）。

\*   `CREATE TABLE` の中で `PRIMARY KEY` を単一の列に設定する場合（列レベル制約）:

    \*   → `括弧不要`です（例: `社員ID INT PRIMARY KEY`）。

**### 3. よくある間違い（試験で狙われやすい誤答パターン）**

\*   `USING 部署ID`:

    \*   **\*\*誤解\*\***: 単一の項目だから括弧は不要だと思い込んでしまう。

    \*   **\*\*正解\*\***: `USING (部署ID)` が正しいです。`USING` はリスト構文として扱います。

\*   `PRIMARY KEY 社員ID, 部署ID`:

    \*   **\*\*誤解\*\***: カンマで区切られているので正しいと錯覚してしまう。

    \*   **\*\*正解\*\***: `PRIMARY KEY (社員ID, 部署ID)` が正しいです。複数の列をまとめる`表レベル制約`では括弧が必須です。

\*   `WHERE A OR B AND C`:

    \*   **\*\*誤解\*\***: 演算子の優先順位 (`AND`が`OR`より先) を忘れて、意図しない結果を招く。

    \*   **\*\*正解\*\***: 意図した順序で評価させるため、`(A OR B) AND C` のように括弧で明示します。

\*   `FROM SELECT … AS T`:

    \*   **\*\*誤解\*\***: `サブクエリ`の括弧を省略してしまい、構文エラーになる。

    \*   **\*\*正解\*\***: `FROM (SELECT …) AS T` が正しいです。

\*   `ON (A.ID = B.ID)`:

    \*   **\*\*誤解\*\***: 安全のために括弧を付けた方が良いと思い込んでしまう。

    \*   **\*\*正解\*\***: `ON`句の単純な条件式に括弧は冗長です。`ON A.ID = B.ID` が標準的な書き方です。

**### 4. 括弧の要否まとめ**

**#### 【必須なケース】**

\*   `サブクエリ`: `(SELECT …)`

\*   `IN句`: `IN (値1, 値2)`

\*   `VALUES句`: `VALUES (値1, 値2)`

\*   `USING句`: `USING (項目)`

\*   `PRIMARY KEY（表レベル）`: `PRIMARY KEY (項目1, 項目2)`

**#### 【推奨されるケース】**

\*   `AND/OR混在時`: `(A OR B) AND C` のように優先順位を明示する場合。

\*   `算術式の明示`: `(給与 \* 1.1) + 1000` のように計算順序を明確にしたい場合。

**#### 【不要なケース】**

\*   `SELECT句`の列名

\*   `FROM句`の表名

\*   `PRIMARY KEY（列レベル）`

\*   `ON句`の単純な条件式

**### 5. 結論（3行まとめ）**

1.  SQLの括弧は「`リスト`」「`サブクエリ`」「`表レベル制約`」では`必須`です。

2.  「`式`」の優先順位を明示する場合は`任意（推奨）`、「`列`・`句`」そのものには`不要`です。

3.  迷ったら`付ける`のが安全です。付けすぎは許容されますが、省略は構文エラーに繋がります。

---

**## AP試験 SQL穴埋め問題の体系的解法**

**## 1. SQL穴埋め問題への基本アプローチ**

**### 1.1. 3つの着眼点（チートシート）**

1.  **\*\*本文のキーワード\*\***: 問題文中の日本語表現が、適切なSQL構文に正しく反映されているか。

2.  **\*\*ER図とテーブル構造\*\***: テーブル間の関係性や構造が、SQLに正確に落とし込まれているか。

3.  **\*\*SQL構文の正当性\*\***: SQLの基本的な文法ルールが守られているか。

**### 1.2. 着眼点1：本文のキーワードから必要な構文を把握する　※必要な計算・構文を把握**

\* 「～ごとに」 → `GROUP BY`

\* 「～がない場合でも」 → `LEFT JOIN`

\* 「～が存在しない場合」 → `NOT EXISTS`

\* 「～が存在する場合のみ」 → `INNER JOIN`

\* 「～が0の場合は0と表示」 → `COALESCE(SUM(...), 0)`

\* 　本文に「昇順」「降順」とあれば → `ORDER BY`

\* その他

  \* `AVG`, `COUNT(\*)`, エイリアス（例：部署別売上合計）などに注目し、全体の意図を把握する

  \* エイリアスがある場合は、該当列を集計している可能性が高いため、`GROUP BY` やウィンドウ関数（`SUM(...) OVER(PARTITION BY ...)`）の使用を疑う

**### 1.3. 着眼点2：ER図とテーブル構造を分析する　※必要なテーブルの扱いを把握**

\*   **\*\***`SELECT`**句の整合性\*\***

    \*   設題が表示しようとしている内容と、`SELECT`句の列が合致しているか。

    \*   集約関数（`AVG`, `SUM`など）がある場合、`GROUP BY`句は存在するか。

    \*   集約関数がない場合、`どこで集約しようとしているのか`を概ね把握する。

\*   **\*\***`FROM`**句と**`JOIN`**条件の妥当性\*\***

    \*   **\*\*FROM内の属性\*\***: `FROM`句内のテーブルの`属性`が、`SELECT`句、`WHERE`句、`JOIN`条件などで`過不足なく`使用されているか。

    \*   **\*\*リレーションシップの確認\*\***: ER図におけるテーブル間のリレーション（例: 「多←1」の矢印）と、主キーと外部キーに基づく正しい`JOIN`条件（例: `ON T1.外部キー = T2.主キー`）が整合しているか。

    \*   **\*\*中間表の役割\*\***: 「多対多」の関係を解消するための中間表（例：注文明細テーブル）があるか。当該表と、`JOIN`により連結されているか。

\*   **\*\***`NULL`**関連処理の確認\*\***

    \*   `LEFT JOIN`、`COALESCE`による`NULL`の置換や、`WHERE 属性 IS NOT NULL`による`NULL`行の除外など、適切に処理がなされているか。

\*   **\*\*SQL文の全体把握\*\***

    \*  SELECTの集約関数、JOIN句、サブクエリ（SELECT,FROM）、エイリアス、WITH（FROM句代わり）、UNION又はUNION ALL、などにより、複数テーブルが連結されている。その全体を把握する。

**### 1.4. 着眼点3：SQL構文の正当性をチェックする　※個々の構文が適切に使用されているか確認**

\*   `SELECT`句に集約関数があれば、すべての非集約列を`GROUP BY`句に記述する。

\*   サブクエリの戻り値の数や型（例：スカラサブクエリは必ず単一の値（1行1列）を返す必要がある）。

---

**### 1.5. 午後試験頻出構文まとめ**

**#### 1.5.1. ウィンドウ関数（順位付け・累計）**

**##### 使用目的：**

集計関数をグループ化せずに、行ごとに計算する。

**##### 使用例：**

```sql

SELECT 社員名, 売上金額,

       RANK() OVER(ORDER BY 売上金額 DESC) AS 売上順位

FROM 売上表;

```

**##### 解説：**

・`RANK()`は順位を付ける関数。売上金額の降順で順位を付ける。

・`OVER(ORDER BY ...)`で並び順を指定。

・同順位がある場合、次の順位は飛ばされる（例：１位が２人なら次は３位）。

---

**##### 使用例（累計）：**

```sql

SELECT 部署, 社員名, 売上日, 売上金額,

       SUM(売上金額) OVER(PARTITION BY 部署 ORDER BY 売上日) AS 累計売上

FROM 売上表;

```

**##### 解説：**

・`SUM(...) OVER(...)`で累計を計算。

・`PARTITION BY`で部署ごとに分割。

・`ORDER BY`で日付順に並べて累計。

**#### 1.5.2. HAVING句（集約後の条件）**

**##### 使用目的：**

`GROUP BY`で集計した後の結果に条件を付ける。

**##### 使用例：**

```sql

SELECT 部署, SUM(金額) AS 合計金額

FROM 売上表

GROUP BY 部署

HAVING SUM(金額) > 100000;

```

**##### 解説：**

・`WHERE`では集計前の行に条件を付ける。

・`HAVING`は集計後の値に条件を付ける。

・`SUM(金額)`などの集計関数と併用する。

---

**#### 1.5.3. CASE式（条件分岐）**

**##### 使用例：**

```sql

SELECT 社員名,

       CASE WHEN 売上金額 >= 100000 THEN '優秀'

            WHEN 売上金額 >= 50000 THEN '普通'

            ELSE '要改善' END AS 評価

FROM 売上表;

```

**##### 解説：**

・`CASE WHEN 条件 THEN 値`で条件分岐。

・`ELSE`はどの条件にも該当しない場合の値。

・`AS 評価`で列名を指定。

---

**#### 1.5.4. サブクエリ（IN・EXISTS）**

**##### 使用例（IN）：**

```sql

SELECT 社員名

FROM 社員表

WHERE 部署ID IN (SELECT 部署ID FROM 部署表 WHERE 地域 = '東北');

```

**##### 解説：**

・`IN`はリストに含まれるかを判定。

・サブクエリは１列のみ返す必要がある。

・属性（型）が一致している必要あり。

---

**##### 使用例（EXISTS）：**

```sql

SELECT 社員名

FROM 社員表 S

WHERE EXISTS (

  SELECT 1 FROM 売上表 U

  WHERE U.社員ID = S.社員ID AND U.売上金額 > 100000

);

```

**##### 解説：**

・`EXISTS`は条件を満たす行が存在するかを判定。

・`SELECT 1`は任意の値でよく、存在確認が目的。

・属性の一致は不要。

---

**#### 1.5.5. WITH句（共通テーブル式）**

**##### 使用例：**

```sql

WITH 部署別集計 AS (

  SELECT 部署, SUM(金額) AS 合計金額

  FROM 売上表

  GROUP BY 部署

)

SELECT \* FROM 部署別集計　　※部署別集計をFROMとして使用している。

WHERE 合計金額 > 100000;

```

**##### 解説：**

・`WITH 名称 AS (...)`で一時的なテーブルを定義。

・後続の`SELECT`でそのテーブルを参照。

・複数の`WITH`句を連結することも可能。

---

**#### 1.5.6. LIMIT・OFFSET（行制限）**

**##### 使用目的：**

表示件数や開始位置を制限する。

**##### 使用例（上位Ｎ件）：**

```sql

SELECT 社員名, 売上金額

FROM 売上表

ORDER BY 売上金額 DESC

LIMIT 10;

```

**##### 使用例（Ｎ件目から取得）：**

```sql

SELECT 社員名

FROM 社員表

LIMIT 10 OFFSET 20;

```

**##### 解説：**

・`LIMIT`は最大件数を指定。

・`OFFSET`は開始位置を指定（０から始まる）。

・ページング処理に利用される。

---

**#### 1.5.7. DISTINCT（重複排除）**

**##### 使用例：**

```sql

SELECT DISTINCT 部署 FROM 社員表;

```

**##### 解説：**

・`DISTINCT`は指定列の重複を除外。

・複数列を指定すると、組み合わせの重複を除外。

---

**#### 1.5.8. DATE関数（期間条件）**

**##### 使用例（日付範囲）：**

```sql

SELECT \* FROM 売上表

WHERE 売上日 BETWEEN '2025-01-01' AND '2025-12-31';

```

**##### 使用例（月単位集計）：**

```sql

SELECT EXTRACT(MONTH FROM 売上日) AS 月, SUM(金額)

FROM 売上表

GROUP BY EXTRACT(MONTH FROM 売上日);

```

**##### 解説：**

・`BETWEEN`は範囲指定。両端を含む。

・`EXTRACT`は日付から年月日などを抽出。

---

**#### 1.5.9. NULL処理**

**##### 使用例（０に置換）：**

```sql

SELECT 社員名, COALESCE(売上金額, 0) AS 売上金額

FROM 売上表;

```

**##### 使用例（NULL除外）：**

```sql

SELECT \* FROM 社員表

WHERE メールアドレス IS NOT NULL;

```

**##### 解説：**

・`COALESCE`はNULLを指定値に置換。

・`IS NOT NULL`はNULLを除外する条件。

---

**### 1.6. 複合構文テンプレート集（午後試験で迷いやすいパターン）**

**#### 部署別売上合計＋順位付け**

```sql

SELECT 部署, SUM(金額) AS 合計,

       RANK() OVER(ORDER BY SUM(金額) DESC) AS 順位

FROM 売上

GROUP BY 部署;

```

**\*\*使用方法\*\***

・`GROUP BY`で部署単位に集計し、`RANK()`で順位付け。

・`OVER(ORDER BY ...)`は並び順を指定。`DESC`で降順。

・注意：`RANK()`は同順位があり得る。連番なら`ROW\_NUMBER()`。

---

**#### 条件付き集計（CASE＋GROUP BY）**

```sql

SELECT 部署,

       SUM(CASE WHEN 区分='A' THEN 金額 ELSE 0 END) AS 区分A合計

FROM 売上

GROUP BY 部署;

```

**\*\*使用方法\*\***

・`CASE`で条件分岐し、該当条件のみ集計。

・`ELSE 0`で条件外は０加算。

・注意：`CASE`は`SELECT`句内で使うのが基本。

---

**#### サブクエリで最大値を取得**

```sql

SELECT 社員名, 金額

FROM 売上

WHERE 金額 = (SELECT MAX(金額) FROM 売上);

```

**\*\*使用方法\*\***

・スカラサブクエリで最大値を取得し、外側で比較。

・注意：サブクエリは１行１列を返す必要あり。

---

**#### WITH句＋JOIN＋HAVING**

```sql

WITH 部署別 AS (

  SELECT 部署ID, SUM(金額) AS 合計 FROM 売上 GROUP BY 部署ID

)

SELECT 部署名, 合計

FROM 部署別 JOIN 部署 ON 部署別.部署ID = 部署.ID

WHERE 合計 > 100000;

```

**\*\*使用方法\*\***

・`WITH`で共通テーブル式を定義し、後続の`SELECT`で利用。

・複雑なSQLを分割して可読性向上。

・注意：`WITH`はスコープがクエリ全体。

---

**#### USING（JOIN条件を簡潔に）**

```sql

SELECT 社員名, 部署名

FROM 社員 JOIN 部署 USING(部署ID);

```

**\*\*使用方法\*\***

・同名列で結合する場合、`ON`の代わりに`USING`で簡潔化。

・注意：`USING`指定列は結果セットで１列に統合。

---

**#### PRECEDING（ウィンドウ関数の範囲指定）**

```sql

SELECT 日付, 金額,

       SUM(金額) OVER(ORDER BY 日付 ROWS 2 PRECEDING) AS 直近3件累計

FROM 売上;

```

**\*\*使用方法\*\***

・`ROWS 2 PRECEDING`は「現在行＋前２行」を範囲に含む。

・移動平均や直近累計で使用。

・注意：`ORDER BY`必須。

---

**#### CURRENT DATE（今日の日付を取得）**

```sql

SELECT \* FROM 売上

WHERE 日付 >= CURRENT *DATE* - INTERVAL '30' DAY;

```

**\*\*使用方法\*\***

・`CURRENT DATE`でシステム日付を取得。

・`INTERVAL`で期間指定（例：過去３０日）。

・注意：日付型との比較で使う。

---

**## 第2部：頻出の誤答パターンと対策（出題者の罠）**

**### 2.1. 条件句の混同**

**#### 2.1.1.** `WHERE`**句 と** `HAVING`**句 の混同**

\*   **\*\*出題者の狙い\*\***: 集約「前」の条件と集約「後」の条件の使い分けを理解しているかを問う。

\*   **\*\*正しい理解\*\***:

    \*   `WHERE`：`GROUP BY`で集約する**\*\*前\*\***の、**\*\*個々の行\*\***に対する条件を指定する。

    \*   `HAVING`：`GROUP BY`で集約した**\*\*後\*\***の、**\*\*グループ\*\***に対する条件を指定する。

\*   **\*\*正答例\*\***: 集約結果に対する条件は`HAVING`句に記述する。

    ```sql

    SELECT 部署ID, AVG(給与) AS 平均給与

    FROM 社員

    GROUP BY 部署ID

    HAVING AVG(給与) >= 300000;

    ```

**#### 2.1.2.** `ON`**句 と** `WHERE`**句 の混同 (特に**`LEFT JOIN`**時)**

\*   **\*\*出題者の狙い\*\***: `LEFT JOIN`において、結合条件と絞り込み条件の違いが結果にどう影響するかの理解を問う。

\*   **\*\*正しい理解\*\***:

    \*   `ON`：テーブルを**\*\*結合するための条件\*\***である。ここに右側テーブルの条件を記述すると、条件に合わない行は`NULL`として結合されるが、左側テーブルの行は結果に残る。

    \*   `WHERE`：結合が**\*\*完了した後の結果\*\***から行を絞り込む条件である。ここに右側テーブルの条件を記述すると、`NULL`の行が除外され、意図せず`INNER JOIN`と同じ結果になることがある。

\*   **\*\*注意すべき例\*\***: `LEFT JOIN ... WHERE B.状態='有効'` と書くと、`B`テーブルに対応するデータがない行（この場合`B.状態`が`NULL`になる）が`WHERE`句によって除外されてしまう。

**### 2.2.** `NULL`**の扱いに伴う罠**

**#### 2.2.1.** `NOT IN` **vs** `NOT EXISTS` **の挙動の違い（サブクエリにNULLを含む場合）**

サブクエリの結果に`NULL`が含まれる可能性がある場合、`NOT IN`は意図しない結果を返すため、`NOT EXISTS`の使用が安全である。

\*   **\*\***`NOT IN` **の場合\*\***

    \*   サブクエリに`NULL`が含まれていると、比較結果がすべて`UNKNOWN`になる。

    \*   `UNKNOWN`を含むと`NOT IN`全体の評価が`FALSE`とみなされるため、**\*\*出力結果はゼロ件（全件除外）\*\***になる。

\*   **\*\***`NOT EXISTS` **の場合\*\***

    \*   `NOT EXISTS`は、相関サブクエリにおいて、親クエリの各行の値を基に、サブクエリが1行でも結果を返すかどうかを評価する。

    \*   サブクエリ内の比較において、`NULL`との比較は不成立（評価結果は `UNKNOWN`）となり、その行は一致したとはみなされない。`FALSE`と同様に扱われる。

    \*   結果として、サブクエリが一致する行を1件も返さなければ`NOT EXISTS`は`TRUE`となり、親クエリの当該行が出力される。

\*   **\*\*挙動のまとめ\*\***

| クエリ形式 | サブクエリ内に `NULL` がある場合 | 出力結果の挙動 |

| :--- | :--- | :--- |

| **\*\***`NOT IN`**\*\*** | 比較不能 → 全体が `FALSE` | **\*\*出力ゼロ件（全件除外）\*\*** |

| **\*\***`NOT EXISTS`**\*\*** | `NULL` は無視 →比較可能な値のみ評価 | **\*\*NULL以外の未一致行は出力される\*\*** |

**#### 2.2.2.** `LEFT JOIN` **後の** `COALESCE` **忘れ**

\*   **\*\*出題者の狙い\*\***: 「～ない場合は0と表示」といった要件に対し、`LEFT JOIN`だけで満足し、`NULL`を`0`に置換し忘れるミスを誘う。

\*   **\*\*正しい理解\*\***: `LEFT JOIN`で対応するデータがなかった場合、集計関数(`SUM`, `COUNT`など)の結果は`NULL`になる。これを`0`として表示するには`COALESCE(SUM(...), 0)`のように明示的な変換する。

**### 2.3. 集約と列指定のルール違反**

**#### 2.3.1.** `GROUP BY` **の記述漏れ**

\*   **\*\*出題者の狙い\*\***: `SELECT`句に**\*\*集約関数と非集約列を混在\*\***させる際のルールを理解しているかを問う。

\*   **\*\*正しい理解\*\***: `SELECT`句に集約関数とそれ以外の列（非集約列）を記述する場合、**\*\*すべての非集約列を**`GROUP BY`**句に指定\*\***しなければならない。

**#### 2.3.2. スカラサブクエリの複数行返却**

\*   **\*\*出題者の狙い\*\***: `SELECT`句内や`WHERE`句の`=`の右辺で使うサブクエリが、必ず単一の値（スカラ値：1行1列）を返さなければならないというルールを突く。

\*   **\*\*正しい理解\*\***: サブクエリが複数行を返す可能性がある場合、`MAX()`や`MIN()`などで結果を1行に絞るか、`IN`や`EXISTS`を使うなど、構文を修正する。