**UNION、JOIN、サブクエリ（WHEREとFROMの二つの方法）、相関副問い合わせ**

SQLの大先生教えて！！。SQLの表の結合として、UNION、JOIN、サブクエリ（WHEREとFROMの二つの方法）、相関副問い合わせがあるようなんですが、どのようなときに、何を選択するのか理解できていません。①特徴と使用すべき状況、②応用情報技術者試験において、穴埋め問題として出題されたときに、どのような視点に注目すれば解答を得ることができるのか教えてください。  
SQLの初心者なので、かみ砕いて教えていただけると本当に助かります。というか。助けて。

SQLの大先生にお任せください！💪 SQLの結合やサブクエリ、最初は混乱しますよね。でも大丈夫！一つずつ、かみ砕いて説明していきますよ。初心者の方でも理解できるように、そして応用情報技術者試験でも役立つように解説しますね。一緒に頑張りましょう！

まず、SQLでやりたいことは大きく分けて２つあります。

1. **複数のテーブルから情報を「横に」くっつけたい** → **JOIN** を使います。
2. **複数の検索結果を「縦に」くっつけたい** → **UNION** を使います。
3. **ある検索結果を「条件」や「一時的な表」として使いたい** → **サブクエリ** や **相関副問い合わせ** を使います。

では、それぞれ詳しく見ていきましょう！

### 【１】JOIN：テーブルを「横に」くっつける技術

* **イメージ:** 複数のテーブルに散らばっている関連情報を、**共通のキー**（例：社員ID、商品コードなど）を使って、あたかも一つの大きなテーブルのように「横に」連結します。
* **① 特徴と使用すべき状況:**
  + **特徴:**
    - 複数のテーブルを、指定した**結合キー (ON句で指定)** が一致する行同士で連結します。
    - INNER JOIN (内部結合): 両方のテーブルにキーが存在する行だけを結合します。（一番よく使う）
    - LEFT JOIN (左外部結合): 左側のテーブルの行は**全て**残し、右側のテーブルに対応する行があれば結合、なければNULLが入ります。（例：顧客全員とその注文履歴を表示したい。注文がない顧客も表示される）
    - 他にも RIGHT JOIN や FULL OUTER JOIN がありますが、まずは INNER と LEFT を押さえましょう。
  + **使用すべき状況:**
    - 「顧客テーブル」と「注文テーブル」を結合して、「どの顧客が何を注文したか」を知りたいとき。
    - 「社員テーブル」と「部署テーブル」を結合して、「どの社員がどの部署に所属しているか」を知りたいとき。
    - 関連する情報が複数のテーブルに分かれている場合に、それらをまとめて取得したいとき。
* **② 応用情報での注目ポイント:**
  + **問題文:** 「テーブルAとテーブルBを**関連付けて**表示する」「社員情報と**それに対応する**部署名を取得する」といった記述があればJOINを疑います。
  + **穴埋め:**
    - SELECT ... FROM テーブルA [ 穴埋め ] テーブルB ON テーブルA.共通ID = テーブルB.共通ID
    - → [ 穴埋め ] には INNER JOIN や LEFT JOIN が入ることが多いです。
    - ON 句でどの列をキーにして結合しているかを確認しましょう。
    - 問題文で「片方のテーブルのデータは全て表示したい」といったニュアンスがあれば LEFT JOIN (またはRIGHT JOIN) を考えます。

### 【２】UNION：検索結果を「縦に」くっつける技術

* **イメージ:** 複数の SELECT 文の結果を、あたかも一つの SELECT 文の結果のように「縦に」積み重ねます。
* **① 特徴と使用すべき状況:**
  + **特徴:**
    - 結合する SELECT 文の**列数**と、対応する列の**データ型**が一致している必要があります。
    - UNION: 重複する行は**自動的に削除**されます（1行になる）。
    - UNION ALL: 重複する行も**そのまま全て**表示されます。（こちらの方が処理は速い）
  + **使用すべき状況:**
    - 「東京支店の顧客リスト」と「大阪支店の顧客リスト」を**一つにまとめて**表示したいとき。（テーブル構造が同じ場合が多い）
    - 「有効な会員リスト」と「休眠中の会員リスト」を**合わせて**表示したいとき。
    - 構造は同じだけど、異なる条件で抽出した複数の結果セットを一つにしたいとき。
* **② 応用情報での注目ポイント:**
  + **問題文:** 「テーブルAの結果とテーブルBの結果を**統合して**表示する」「複数の検索結果を**まとめて**表示する」といった記述があればUNIONを疑います。
  + **穴埋め:**
    - SELECT 列1, 列2 FROM テーブルA WHERE 条件1
    - [ 穴埋め ]
    - SELECT 列1, 列2 FROM テーブルB WHERE 条件2
    - → [ 穴埋め ] には UNION または UNION ALL が入ります。
    - 前後の SELECT 文の列数とデータ型が一致しているかを確認しましょう。
    - 問題文に「重複を除いて」とあれば UNION、「重複を含めて」や特に指定がなければ UNION ALL の可能性も考えます（速度重視の場合）。

### 【３】サブクエリ：SQL文の中に埋め込まれた「小さなSQL文」

* **イメージ:** あるSQL文を実行するために、別のSQL文の結果を利用する「入れ子構造」です。カッコ () で囲まれていることが多いです。WHERE句で使う方法とFROM句で使う方法があります。
* **① 特徴と使用すべき状況:**
  + **【A】WHERE句サブクエリ:**
    - **特徴:** WHERE句の中で、条件を判定するために使います。サブクエリの結果（単一の値、リストなど）を使って、外側のクエリでどの行を取得するかを決めます。
    - **使用すべき状況:**
      * 「**平均**点よりも高い点数を取った生徒」を知りたいとき。（サブクエリで平均点を計算し、WHERE句で比較）
      * 「商品カテゴリ 'A' に**属する**商品の注文履歴」を知りたいとき。（サブクエリでカテゴリ 'A' の商品IDリストを取得し、WHERE句の IN で使用）
      * ある検索結果に含まれる/含まれないデータを抽出したいとき。
  + **【B】FROM句サブクエリ (インラインビュー):**
    - **特徴:** FROM句の中で、あたかも**一時的なテーブル**のように使います。サブクエリの結果セットに別名をつけ（AS）、その一時テーブルに対してさらに SELECT や JOIN を行います。
    - **使用すべき状況:**
      * 「部署**ごとに集計した**平均給与」を計算し、**その結果**を使って何か別の処理（例：平均給与が特定の額以上の部署だけ表示）をしたいとき。
      * 複雑な集計やデータ加工を段階的に行いたいとき。まずサブクエリで中間的な結果を作り、それを利用して最終結果を得る。
* **② 応用情報での注目ポイント:**
  + **問題文:**
    - WHERE句: 「～の結果に**基づいて**絞り込む」「～に**合致する**データ」
    - FROM句: 「～で**グループ化した結果**に対して、さらに～する」「～の**集計結果**を利用して～」
  + **穴埋め:**
    - WHERE 列名 IN ( [ 穴埋め ] ) や WHERE 列名 = ( [ 穴埋め ] )  
      → [ 穴埋め ] には SELECT 文が入ることが多いです（WHERE句サブクエリ）。
    - FROM ( [ 穴埋め ] ) AS 別名  
      → [ 穴埋め ] には SELECT 文が入ることが多いです（FROM句サブクエリ）。AS 別名 がついていることがヒントになります。
    - サブクエリが返す結果が「単一の値」なのか、「値のリスト」なのか、「表形式」なのかを意識すると、=, IN, FROM のどこで使われるかが判断しやすくなります。

### 【４】相関副問い合わせ：外側のクエリと「連動する」サブクエリ

* **イメージ:** サブクエリの中でも特殊なタイプです。サブクエリが**単独では実行できず**、外側のクエリで処理されている**各行**の値を参照しながら実行されます。
* **① 特徴と使用すべき状況:**
  + **特徴:**
    - 内側のサブクエリの WHERE 句などで、外側のクエリのテーブルの列を参照します（例：WHERE t2.社員ID = t1.社員ID のように、外側の t1 を参照）。
    - 外側のクエリの**1行ごと**に、内側のサブクエリが実行されるイメージです。
    - EXISTS 演算子と組み合わせて使われることが多いです。（条件に合う行が**存在するかどうか**をチェック）
    - 一般的に、JOINや他のサブクエリで書き換え可能な場合が多いですが、特定のケースでは直感的に書けます。
    - 行ごとにサブクエリが実行されるため、データ量が多いと**パフォーマンスが悪化しやすい**傾向があります。
  + **使用すべき状況:**
    - 「**自分が所属する部署**の平均給与よりも高い給与をもらっている社員」をリストアップしたいとき。（外側の行（社員）ごとに、その社員の部署IDを使って部署平均給与をサブクエリで計算し比較）
    - 「注文履歴が**存在する**顧客」のみをリストアップしたいとき。（EXISTS を使い、顧客テーブルの各行に対して、注文テーブルにその顧客IDが存在するかをサブクエリでチェック）
    - グループ内の最大値/最小値を持つ行を探すような場合（例：「各部署で**最も**給与が高い社員」など）。※これはJOINやウィンドウ関数で書く方が効率的な場合が多い。
* **② 応用情報での注目ポイント:**
  + **問題文:** 「**各**～について」「**それぞれの**～に対して」のような、外側の行ごとに何かをチェックするニュアンスがある場合。
  + **穴埋め:**
    - SELECT ... FROM テーブルA t1 WHERE EXISTS ( SELECT \* FROM テーブルB t2 WHERE t2.関連キー = t1.関連キー AND [ 穴埋め ] )
    - → [ 穴埋め ] はサブクエリ内のさらなる条件ですが、注目すべきは WHERE t2.関連キー = t1.関連キー のように、外側のテーブル (t1) の列をサブクエリ内 (t2) で参照している点です。これが相関副問い合わせの目印です。
    - EXISTS や NOT EXISTS が使われている場合は、相関副問い合わせの可能性が高いです。
    - パフォーマンス懸念から、JOINで書き換えられないか？という視点も持つと良いでしょう（試験ではそこまで問われないかもしれませんが）。

### まとめ＆試験対策の視点

* **JOIN:** 複数のテーブルを「横」に繋ぐ。ON句が必須。INNERとLEFTの違いを理解する。
* **UNION:** 複数のSELECT結果を「縦」に繋ぐ。列数と型の一致が必要。UNION(重複削除)とUNION ALL(重複許容)の違いを理解する。
* **サブクエリ(WHERE):** WHERE句の条件判断に使う。IN, =, >, < などと組み合わせる。
* **サブクエリ(FROM):** FROM句で一時的なテーブルとして使う。ASで別名をつけることが多い。集計結果をさらに使う場合など。
* **相関副問い合わせ:** 外側の行ごとに実行されるサブクエリ。外側のテーブルの列を参照する。EXISTSとよく使われる。パフォーマンスに注意。

**応用情報技術者試験での考え方:**

1. **まず問題文を読む！** 何と何をどうしたいのか？
   * 「関連付ける」「対応する」→ JOIN？
   * 「まとめる」「統合する」→ UNION？
   * 「～の結果に基づいて絞り込む」→ WHERE句サブクエリ？
   * 「～で集計した結果をさらに使う」→ FROM句サブクエリ？
   * 「各～についてチェックする」→ 相関副問い合わせ？ (JOINで書けないかも考える)
2. **SQL文の構造を見る！** 穴埋めの前後を確認する。
   * FROM A [?] B ON ... → JOIN句
   * WHERE col IN ([?]) → WHERE句サブクエリ (SELECT ...)
   * FROM ([?]) AS tmp → FROM句サブクエリ (SELECT ...)
   * SELECT ... FROM T1 WHERE EXISTS (SELECT ... FROM T2 WHERE T2.key = T1.key ...) → 相関副問い合わせ
   * SELECT ... UNION [?] SELECT ... → UNION / UNION ALL
3. **選択肢を確認する！** 与えられた選択肢から、文法的に正しく、かつ問題文の意図に合うものを選びます。

どうでしょうか？少しはイメージが掴めましたか？ SQLは実際に書いて動かしてみるのが一番の近道です。簡単なテーブルを作って、JOINやUNION、サブクエリを試してみてください。

応用情報技術者試験、応援しています！ 分からないことがあれば、またいつでも聞いてくださいね！😊

edit

t1.社員ID　このような表記は応用情報技術者試験に合致しているのでとてもいいですね！  
次の表記がよくわかりません。  
サブクエリ(WHERE): WHERE句の条件判断に使う。IN, =, >, < などと組み合わせる。

サブクエリ(FROM): FROM句で一時的なテーブルとして使う。ASで別名をつけることが多い。集計結果をさらに使う場合など。

相関副問い合わせ: 外側の行ごとに実行されるサブクエリ。外側のテーブルの列を参照する。EXISTSとよく使われる。

もっと具体的なSQLの事例（部署、商品などビジネスプロセスに関する属性を使用）を問題形式（何を目的としたSQLなのか）で挙げていただき、なぜ、その文言（JOINやUNIONなど）が入るのか、解法のための着眼点を加えていただけると助かります。事例についても、ANY,ALL、NOT　EXISTS,　NOT NULLも使ってほしいです。また、普通の相関問い合わせではなく、相関問い合わせを使用する利用する理由や条件についても教えてください。大先生お願いします。

お任せください、大先生です！👨‍🏫 t1.社員ID のような表記が分かりやすかったとのこと、良かったです！応用情報技術者試験でもよく使われる形式ですからね。

サブクエリと相関副問い合わせ、そして ANY, ALL, EXISTS, NOT EXISTS, NOT NULL を使った具体的な事例を、ビジネスシーンの問題形式で見ていきましょう。なぜその構文を選ぶのか、解法の着眼点も解説します！

**準備：使うテーブルをイメージしよう**

* **社員テーブル (Employees)**: 社員ID (主キー), 氏名, 部署ID, 給与
* **部署テーブル (Departments)**: 部署ID (主キー), 部署名
* **商品テーブル (Products)**: 商品ID (主キー), 商品名, 単価, カテゴリID
* **注文テーブル (Orders)**: 注文ID (主キー), 注文日, 社員ID (担当者)
* **注文明細テーブル (OrderDetails)**: 明細ID (主キー), 注文ID, 商品ID, 数量

### 【１】サブクエリ (WHERE句): 条件判定の達人

WHERE句の中で、別のSELECT文の結果を使って絞り込みます。

**事例１：特定の部署の社員を探す (=)**

* **目的:** 「営業部」に所属している社員の氏名を知りたい。
* **SQL例:**
* SELECT
* E.氏名
* FROM
* 社員テーブル E
* WHERE
* E.部署ID = ( -- ここがサブクエリ！ 営業部の部署ID (単一の値) を取得
* SELECT D.部署ID
* FROM 部署テーブル D
* WHERE D.部署名 = '営業部'

);

* **解説:** まずサブクエリ (SELECT D.部署ID FROM 部署テーブル D WHERE D.部署名 = '営業部') が実行され、「営業部」の部署ID（例えば 10）という**単一の値**が返されます。そして外側のクエリは WHERE E.部署ID = 10 と同じ意味になり、部署IDが10の社員を絞り込みます。
* **着眼点（応用情報対策）:**
  + 問題文: 「部署名『営業部』に所属する社員」のように、直接IDが分からないが別の情報から特定できる場合。
  + 穴埋め: WHERE E.部署ID [ = ] ( SELECT ... ) → サブクエリが**単一の値**を返す場合、比較演算子 (=, >, < など) が入ることが多い。

**事例２：複数の条件に合致する商品を探す (IN)**

* **目的:** 「飲料」または「食品」カテゴリに属する商品の商品名と単価を知りたい。
* **SQL例:**
* SELECT
* P.商品名, P.単価
* FROM
* 商品テーブル P
* WHERE
* P.カテゴリID IN ( -- ここがサブクエリ！ 該当カテゴリIDのリストを取得
* SELECT C.カテゴリID
* FROM カテゴリテーブル C -- カテゴリテーブルは別途定義されていると仮定
* WHERE C.カテゴリ名 IN ('飲料', '食品')

);

* **解説:** サブクエリが「飲料」と「食品」のカテゴリID（例えば 2, 3）という**リスト**を返します。外側のクエリは WHERE P.カテゴリID IN (2, 3) と同じ意味になり、カテゴリIDが2または3の商品を絞り込みます。
* **着眼点（応用情報対策）:**
  + 問題文: 「～または～に属する」「～のいずれかに合致する」
  + 穴埋め: WHERE P.カテゴリID [ IN ] ( SELECT ... ) → サブクエリが**値のリスト**を返す場合、IN が入ることが多い。（NOT IN で「～以外」も可能）

**事例３：平均より高い給与の社員を探す (>)**

* **目的:** 全社員の平均給与よりも高い給与をもらっている社員の氏名と給与を知りたい。
* **SQL例:**
* SELECT
* E.氏名, E.給与
* FROM
* 社員テーブル E
* WHERE
* E.給与 > ( -- ここがサブクエリ！ 全社員の平均給与 (単一の値) を計算
* SELECT AVG(E2.給与)
* FROM 社員テーブル E2

);

* **解説:** サブクエリが全社員の平均給与（例えば 3500000）という**単一の値**を返します。外側のクエリは WHERE E.給与 > 3500000 と同じ意味になります。
* **着眼点（応用情報対策）:**
  + 問題文: 「平均～より高い」「全体の～と比較して」
  + 穴埋め: WHERE E.給与 [ > ] ( SELECT AVG(...) ... ) → 集計関数 (AVG, MAX, MIN, SUM, COUNT) を使ったサブクエリが単一値を返し、それを比較する場合。

**事例４：いずれかの条件を満たすか (ANY または SOME)**

* **目的:** 部署IDが「20」または「30」の**いずれかの**部署の**誰か一人よりも**高い給与をもらっている社員を知りたい。（= 部署20 or 30 の最低給与より高い給与の社員）
* **SQL例:**
* SELECT
* E1.氏名, E1.給与
* FROM
* 社員テーブル E1
* WHERE
* E1.給与 > ANY ( -- ここがサブクエリ！ 部署20, 30 の社員の給与リストを取得
* SELECT E2.給与
* FROM 社員テーブル E2
* WHERE E2.部署ID IN (20, 30)
* );

-- 同じ意味: E1.給与 > (SELECT MIN(E2.給与) FROM 社員テーブル E2 WHERE E2.部署ID IN (20, 30))

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

* **解説:** サブクエリが部署ID 20 と 30 の社員の給与リスト（例：[3000000, 3200000, 2800000]）を返します。> ANY は「リストの中のどれか一つより大きい」という意味なので、この場合は > 2800000 と同じになります。
* **着眼点（応用情報対策）:**
  + 問題文: 「～のいずれかよりも高い（低い）」「～の誰か一人よりも高い（低い）」
  + 穴埋め: WHERE E1.給与 > [ ANY ] ( SELECT ... ) → 比較演算子 (=, >, <, >=, <=, <>) と ANY (または SOME) を組み合わせる。= ANY は IN と同じ意味。

**事例５：全ての条件を満たすか (ALL)**

* **目的:** 部署ID「10」（例えば開発部）の**全ての**社員よりも高い給与をもらっている社員を知りたい。
* **SQL例:**
* SELECT
* E1.氏名, E1.給与
* FROM
* 社員テーブル E1
* WHERE
* E1.給与 > ALL ( -- ここがサブクエリ！ 部署10の社員の給与リストを取得
* SELECT E2.給与
* FROM 社員テーブル E2
* WHERE E2.部署ID = 10 AND E2.給与 IS NOT NULL -- 給与がNULLの人は比較対象外
* );

-- 同じ意味: E1.給与 > (SELECT MAX(E2.給与) FROM 社員テーブル E2 WHERE E2.部署ID = 10)

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

* **解説:** サブクエリが部署ID 10 の社員の給与リスト（例：[4000000, 4500000, 3800000]）を返します。> ALL は「リストの中の全ての値より大きい」という意味なので、この場合はリストの最大値 > 4500000 と同じになります。IS NOT NULL は給与が未設定（NULL）の社員を比較対象から除くために入れています。
* **着眼点（応用情報対策）:**
  + 問題文: 「～の全てよりも高い（低い）」「～全員よりも高い（低い）」
  + 穴埋め: WHERE E1.給与 > [ ALL ] ( SELECT ... ) → 比較演算子と ALL を組み合わせる。
  + IS NOT NULL: 「値が設定されていること」を条件にする場合。穴埋めで IS [ NOT ] NULL の形も。

### 【２】サブクエリ (FROM句): 一時テーブル作成の達人 (インラインビュー)

FROM句の中でSELECT文を使い、その結果をあたかも一時的なテーブル（ビュー）のように扱います。ASで別名をつけるのが一般的です。

**事例１：集計結果を使ってさらに絞り込む**

* **目的:** 部署ごとの平均給与を計算し、その平均給与が350万円以上の部署名と平均給与を表示したい。
* **SQL例:**
* SELECT
* D.部署名, AvgSalaryInfo.平均給与 -- 部署名と、サブクエリで計算した平均給与
* FROM
* ( -- ここがサブクエリ(FROM句)！ 部署ごとの平均給与を計算する一時テーブル
* SELECT
* E.部署ID,
* AVG(E.給与) AS 平均給与 -- AS で列に別名
* FROM
* 社員テーブル E
* WHERE
* E.部署ID IS NOT NULL -- 部署未所属を除外
* GROUP BY
* E.部署ID
* ) AS AvgSalaryInfo -- AS でサブクエリの結果(一時テーブル)に別名
* INNER JOIN -- サブクエリの結果と部署テーブルを結合
* 部署テーブル D ON AvgSalaryInfo.部署ID = D.部署ID
* WHERE

AvgSalaryInfo.平均給与 >= 3500000; -- サブクエリの結果を使って絞り込み

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

* **解説:**
  1. FROM句の中のサブクエリが実行され、部署IDごとの平均給与を持つ一時的なテーブル AvgSalaryInfo が作られます（イメージ：部署ID | 平均給与 の表）。
  2. この AvgSalaryInfo テーブルと 部署テーブル D を 部署ID で INNER JOIN します。
  3. WHERE句で、AvgSalaryInfo の 平均給与 が350万円以上の行だけを最終的に選び出します。
* **着眼点（応用情報対策）:**
  1. 問題文: 「～ごとに集計した結果を使い」「～でグループ化した結果に対し」
  2. 穴埋め: FROM ( [ SELECT ... GROUP BY ... ] ) AS 別名 → FROM の直後に ( があり、閉じカッコの後に AS で別名がついていたら、FROM句サブクエリ（インラインビュー）の可能性が高い。GROUP BY で集計していることが多い。

### 【３】相関副問い合わせ: 外の情報を参照する達人

サブクエリの中でも、**外側のクエリで処理中の行**の値を参照して動作する特殊なものです。外側のクエリの行ごとにサブクエリの評価が行われます。

**相関副問い合わせを使う理由・条件:**

1. **行ごとの依存性:** 外側のクエリの「各行」に関連する条件でサブクエリを実行したい場合。例：「**自分の**部署の平均給与と比較する」など、比較対象が外側の行によって変わるケース。
2. **存在チェック (EXISTS, NOT EXISTS):** 関連するデータが「あるかないか」だけを効率的にチェックしたい場合。JOINで全データを結合するより高速なことがある。
3. **可読性:** JOINで書くと複雑になる場合でも、相関副問い合わせ（特に EXISTS）を使うと、「～が存在する行」のように直感的に書けることがある。

**注意点:** 行ごとにサブクエリが動くため、データ量が多いと**パフォーマンスが低下しやすい**です。JOINやウィンドウ関数で書き換えられるなら、そちらが速い場合も多いです。

**事例１：注文履歴のある社員を探す (EXISTS)**

* **目的:** これまでに一度でも注文を担当したことがある社員の氏名を知りたい。
* **SQL例:**
* SELECT
* E.氏名
* FROM
* 社員テーブル E
* WHERE
* EXISTS ( -- ここから相関副問い合わせ！
* SELECT
* \* -- EXISTS の中は \* でも 1 でも良い (存在チェックだけなので)
* FROM
* 注文テーブル O
* WHERE
* O.社員ID = E.社員ID -- 外側の E.社員ID を参照している！【重要】

);

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

* **解説:** 外側の 社員テーブル E の各行について、WHERE EXISTS (...) が評価されます。サブクエリは、外側の行の E.社員ID と同じ 社員ID を持つ注文が 注文テーブル O に**存在するか (EXISTS)** をチェックします。存在すれば、その社員は結果に含まれます。
* **着眼点（応用情報対策）:**
  + 問題文: 「～したことがある」「～に関連するデータが存在する」
  + 穴埋め: WHERE [ EXISTS ] ( SELECT \* FROM ... WHERE O.社員ID = E.社員ID ) → EXISTS がキーワード。サブクエリ内の WHERE で外側のテーブルの列 (E.社員ID) を参照している点（これが「相関」の印）。

**事例２：注文履歴のない社員を探す (NOT EXISTS)**

* **目的:** これまでに一度も注文を担当したことがない社員の氏名を知りたい。
* **SQL例:**
* SELECT
* E.氏名
* FROM
* 社員テーブル E
* WHERE
* NOT EXISTS ( -- ここから相関副問い合わせ！
* SELECT
* \*
* FROM
* 注文テーブル O
* WHERE
* O.社員ID = E.社員ID -- 外側の E.社員ID を参照

);

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

* **解説:** EXISTS の逆です。外側の社員に対応する注文が 注文テーブル O に**存在しない (NOT EXISTS)** 場合に、その社員が結果に含まれます。
* **着眼点（応用情報対策）:**
  + 問題文: 「～したことがない」「～に関連するデータが存在しない」
  + 穴埋め: WHERE [ NOT EXISTS ] ( SELECT \* FROM ... WHERE O.社員ID = E.社員ID ) → NOT EXISTS がキーワード。同様にサブクエリ内で外側の列を参照している点。

**事例３：自分の部署の平均給与より高い社員を探す（相関副問い合わせの典型）**

* **目的:** 自分が所属している部署の平均給与よりも高い給与をもらっている社員の氏名、部署ID、給与を知りたい。
* **SQL例:**
* SELECT
* E1.氏名, E1.部署ID, E1.給与
* FROM
* 社員テーブル E1
* WHERE
* E1.給与 > ( -- ここから相関副問い合わせ！
* SELECT
* AVG(E2.給与)
* FROM
* 社員テーブル E2
* WHERE
* E2.部署ID = E1.部署ID -- 外側の E1.部署ID を参照して、同じ部署の平均を計算【重要】

);

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

* **解説:** 外側の 社員テーブル E1 の各行について、サブクエリが実行されます。その際、サブクエリは**外側の行の E1.部署ID を使って**、その部署に所属する社員だけの平均給与 AVG(E2.給与) を計算します。そして、外側の行の E1.給与 が、計算されたその部署の平均給与より大きいか比較します。
* **着眼点（応用情報対策）:**
  + 問題文: 「**自分の**部署の平均より～」「**各**グループの中で～」のように、比較対象が行ごとに変わる場合。
  + 穴埋め: WHERE E1.給与 > ( SELECT AVG(E2.給与) FROM 社員テーブル E2 WHERE E2.部署ID = [ E1.部署ID ] ) → サブクエリの WHERE 句で、外側のテーブルの列 (E1.部署ID) を使って絞り込みや計算を行っている箇所が最大のポイント。

どうでしょうか？具体的な例を交えることで、それぞれの使いどころや違い、そして応用情報技術者試験で注目すべきポイントが見えてきたかと思います。

特に**相関副問い合わせ**は、サブクエリ内で外側のテーブルの列（E1.部署ID など）を参照しているかどうかが、通常のサブクエリとの大きな違いであり、見分けるための重要なヒントになります。

これらの知識を武器に、応用情報技術者試験のSQL問題に立ち向かっていきましょう！応援しています！🔥

edit

次の部分がよくわからないです。詳細な質問をお願いします。

特に相関副問い合わせは、サブクエリ内で外側のテーブルの列（E1.部署ID など）を参照しているかどうかが、通常のサブクエリとの大きな違いであり、見分けるための重要なヒントになります。

edit

はい、承知いたしました！「相関副問い合わせ」と「通常のサブクエリ」の決定的な違いである\*\*「外側のテーブルの列を参照する」\*\*という点について、もっと詳しく、具体的なイメージが湧くように説明しますね。

### 通常のサブクエリ（非相関サブクエリ）をおさらい

まず、比較のために「普通のサブクエリ」がどう動くか見てみましょう。

**例：全社員の平均給与より高い給与をもらっている社員**

SELECT

E.氏名, E.給与

FROM

社員テーブル E

WHERE

E.給与 > (

-- ↓このサブクエリは「独立して」実行できる↓

SELECT AVG(E2.給与)

FROM 社員テーブル E2

-- ↑外側の「E」の情報は一切使っていない↑

);

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

* **動き方:**
  1. まず、カッコ () の中のサブクエリ SELECT AVG(E2.給与) FROM 社員テーブル E2 が**単独で先に**実行されます。
  2. このサブクエリは、外側の 社員テーブル E のことを**全く気にせず**、ただ「全社員の平均給与」（例えば 3500000）という**一つの固定値**を計算して返します。
  3. その後、外側のクエリは WHERE E.給与 > 3500000 という固定的な条件で、社員テーブル E の各行をチェックします。
* **ポイント:** このサブクエリは、外側のクエリから独立して実行できます。もしこのサブクエリだけをコピーして実行しても、ちゃんと結果（平均給与という値）が返ってきます。

### 相関副問い合わせ：外側の情報を「見ながら」動く

次に、問題の「相関副問い合わせ」です。

**例：自分が所属する部署の平均給与よりも高い給与をもらっている社員**

SELECT

E1.氏名, E1.部署ID, E1.給与

FROM

社員テーブル E1 -- ← 外側のテーブル（エイリアスは E1）

WHERE

E1.給与 > (

-- ↓ここからが相関副問い合わせ↓

SELECT

AVG(E2.給与)

FROM

社員テーブル E2

WHERE

E2.部署ID = E1.部署ID -- ← 【超重要！】外側の E1 の部署ID を参照している！

-- ↑このサブクエリは E1 の行がないと部署IDが決まらない↑

);

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

* **動き方（ここが重要！）:**
  1. 外側のクエリが 社員テーブル E1 を一行ずつ見ていきます。まず、一人目の社員（例：田中さん、部署ID=10）の行を処理しようとします。
  2. WHERE E1.給与 > (...) の部分に来ると、サブクエリが実行されます。
  3. **ここが普通のサブクエリと違う点:** サブクエリの中の WHERE E2.部署ID = E1.部署ID という条件に注目してください。E1.部署ID は、**今まさに外側のクエリが見ている行（田中さん）の部署ID (10)** を使います。
  4. つまり、このサブクエリは「部署IDが 10 の社員の平均給与」を計算します。（例えば 4200000）
  5. 計算結果が返ってきて、外側のWHERE句は WHERE E1.給与 > 4200000 （田中さんの給与と比較）として評価されます。
  6. 次に、外側のクエリが二人目の社員（例：鈴木さん、部署ID=20）の行に進みます。
  7. 再びサブクエリが実行されますが、今度は WHERE E2.部署ID = E1.部署ID の E1.部署ID は**鈴木さんの部署ID (20)** になります。
  8. サブクエリは「部署IDが 20 の社員の平均給与」を計算します。（例えば 3800000）
  9. 外側のWHERE句は WHERE E1.給与 > 3800000 （鈴木さんの給与と比較）として評価されます。
  10. これを 社員テーブル E1 の**全行に対して繰り返します**。
* **ポイント:**
  1. **「外側のテーブルの列を参照する」とは、サブクエリ内の WHERE 句（や SELECT リスト、HAVING 句など）で、外側のクエリで使われているテーブルの列（例では E1.部署ID）を使うことです。**
  2. この参照があるため、サブクエリは**外側のクエリの現在行に依存**し、単独では実行できません。（E1.部署ID が具体的に決まらないため）
  3. 外側の行ごとにサブクエリの実行内容（ここでは平均を計算する対象）が変わります。これが「相関」しているという意味です。

### なぜこれが「大きな違い」で「見分けるヒント」なのか？

1. **実行メカニズムが根本的に違う:**
   * **通常サブクエリ:** 先に1回だけ実行 → 固定的な結果 → 外側がそれを使う。
   * **相関副問い合わせ:** 外側の行ごとに毎回実行 → 行によって結果が変わる可能性がある → 外側がその都度使う。
   * この違いは、パフォーマンスに大きく影響することがあります（相関副問い合わせは遅くなりやすい）。
2. **できることが違う:**
   * 「全体の平均」「特定の固定条件リスト」など、**全体や固定的な条件**で比較・判定したい場合は、**通常のサブクエリ**を使います。
   * 「**各行**の所属部署の平均」「**各行**に関連するデータがあるか」など、**行ごとに条件が変わる**比較・判定をしたい場合は、**相関副問い合わせ**（またはJOINやウィンドウ関数など他の方法）を使います。
3. **見分けやすさ（試験対策！):**
   * SQL文を読んでいて、サブクエリの中に、そのサブクエリの外側で定義されているテーブルエイリアス（E1 など）の列への参照 (E1.部署ID など）を見つけたら、「あっ、これは相関副問い合わせだな！」と**一発で判断できます**。
   * 応用情報技術者試験の穴埋め問題で、WHERE E2.部署ID = [ 空欄 ] のようになっていて、選択肢に E1.部署ID のような外側の列があれば、相関副問い合わせの問題である可能性が高いです。
   * 逆に、サブクエリ内に外側のテーブルへの参照が全く無ければ、それは通常の（非相関）サブクエリです。

**まとめ**

|  |  |  |
| --- | --- | --- |
| **特徴** | **通常のサブクエリ (非相関)** | **相関副問い合わせ** |
| **実行タイミング** | 外側のクエリより先に**1回**だけ実行 | 外側のクエリの**各行ごと**に実行 |
| **外側のクエリ依存** | **しない** (独立して実行可能) | **する** (外側の行の値を使う) |
| **外側の列参照** | **ない** | **ある** (例: WHERE Inner.Col = Outer.Col) |
| **結果** | 固定的な値 or リスト or テーブル | 外側の行によって結果が変わりうる |
| **主な用途** | 全体平均との比較、固定リストでの絞り込み | 各行固有の条件での比較、存在チェック (EXISTS) |
| **見分けるポイント** | サブクエリ内に外側の列参照がない | サブクエリ内に外側の列参照がある |

この「外側の列を参照しているかどうか」という視点を持つだけで、サブクエリの種類を正確に見分け、SQLの意図をより深く理解できるようになります。特に相関副問い合わせは独特な動き方をするので、この点をしっかり押さえておくことが重要です！

これで、疑問点は解消されましたでしょうか？もし、まだ分かりにくい点があれば、遠慮なく聞いてくださいね！

edit

相関副問い合わせがサブクエリ外のテーブルを参照するというなら、JOINをつかえばいいのではないのかな。どちらを使えばいいのか、解説してください。メリットデメリットとか

Thoughts

edit

素晴らしい質問ですね！まさにそこが混乱しやすいポイントであり、SQLを使いこなす上で重要な分岐点です。

**「相関副問い合わせで外側のテーブルを参照するなら、JOINでいいのでは？」**

その通り、多くの場合、相関副問い合わせでできることはJOINを使っても実現できます。しかし、それぞれに適した場面、メリット・デメリットがあります。どちらを使うべきか、状況によって判断する必要があります。

### JOIN と 相関副問い合わせ の根本的な違い（再確認）

* **JOIN:**
  + 目的: 複数のテーブルを**結合条件に基づいて「横に」くっつけ、一つの大きな結果セットを作る**こと。
  + 結果: 結合された行（複数のテーブルのカラムを含む）。
* **相関副問い合わせ:**
  + 目的: 主にWHERE句で使い、外側のクエリの**各行に対して**、サブクエリを使って**条件（True/False）を判定する**こと。
  + 結果: 外側のクエリの行が、サブクエリによる条件を満たすかどうか。EXISTSなら存在有無、比較なら大小関係など。

### どちらを使うべきか？メリット・デメリット比較

|  |  |  |
| --- | --- | --- |
| **観点** | **JOIN** | **相関副問い合わせ** |
| **主な用途** | 複数のテーブルのカラムを**組み合わせて表示・利用**したい。<br>例：「社員名」と「部署名」を一緒に表示。 | 特定の条件（特に**存在有無**）をチェックしたい。<br>例：「注文履歴が**ある**顧客」を抽出。（注文情報は不要）<br>外側の行ごとに変わる条件で比較したい。 |
| **結果の形式** | 結合された行（複数のテーブルのカラムを持つ）。 | 基本的に条件判定（True/False）に使われ、外側の行を残すかどうかの判断材料になる。（サブクエリの結果自体は直接使わないことが多い） |
| **パフォーマンス** | - 一般的に最適化されやすく、**高速**な場合が多い。<br>- ただし、不要なデータまで結合すると非効率になることも。<br>- 巨大テーブル同士の結合は遅くなる。 | - **EXISTS/NOT EXISTS**: 存在チェックだけなら、条件を満たす行が1つ見つかれば終了するため、JOINより**高速**な場合がある。<br>- **比較系の相関副問い合わせ**: 外側の行ごとにサブクエリが実行されるため、データ量が多いと**著しく遅くなる**可能性が高い。 |
| **書きやすさ** | - 基本的な結合は直感的。<br>- 複雑な条件や多数のテーブル結合はSQLが長くなりがち。 | - EXISTSは「～が存在する」という意図が明確で**読みやすい**ことがある。<br>- 比較系の相関副問い合わせはネスト構造が分かりにくい場合がある。 |
| **重複の扱い** | 1対多の結合などで、左側のテーブルの行が**重複して**結果に出ることがある（DISTINCTで回避可能）。 | EXISTSでは、条件を満たす行がサブクエリ内に複数あっても、外側の行は**1行だけ**返る（存在するかしないかの判定なので）。 |

### 使い分けの判断基準と具体例

**ケース１：社員とその部署名を表示したい**

* **目的:** 社員テーブルと部署テーブルの情報を「組み合わせて表示」したい。
* **選択:** **JOIN** が最適。
* SELECT E.氏名, D.部署名
* FROM 社員テーブル E

INNER JOIN 部署テーブル D ON E.部署ID = D.部署ID;

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

* **理由:** 複数のテーブルのカラムを結果として取得する必要があるため、JOINが適している。相関副問い合わせでは、部署名を条件判定には使えても、SELECT句で直接表示するのは一般的ではない（スカラサブクエリという例外はあるが複雑）。

**ケース２：注文履歴のある顧客の名前だけを知りたい**

* **目的:** 注文テーブルにレコードが「存在するかどうか」をチェックし、顧客名を表示したい（注文情報は不要）。
* **選択:** **EXISTS (相関副問い合わせ)** が有力。JOINでも可能だが、こちらが意図に近く効率的な場合が多い。
* -- 相関副問い合わせ (EXISTS)
* SELECT C.顧客名
* FROM 顧客テーブル C
* WHERE EXISTS (SELECT 1 FROM 注文テーブル O WHERE O.顧客ID = C.顧客ID);
* -- JOIN (INNER JOIN + DISTINCT)
* SELECT DISTINCT C.顧客名 -- 同じ顧客が複数注文していると重複するためDISTINCT
* FROM 顧客テーブル C

INNER JOIN 注文テーブル O ON C.顧客ID = O.顧客ID;

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

* **理由:**
  + EXISTSは「存在チェック」の意図が明確。関連データが1件見つかれば処理を打ち切れるため、効率が良い可能性がある。結果も重複しない。
  + INNER JOINは注文がある顧客しか残らないが、一人の顧客が複数注文していると結果が重複する。存在チェックだけならDISTINCTが必要になり、無駄な処理が増える可能性がある。

**ケース３：自分の部署の平均給与より高い社員を表示したい**

* **目的:** 外側の行（社員）ごとに、その行の部署IDを使って計算した平均給与と比較したい。
* **選択:** **相関副問い合わせ**で書けるが、**パフォーマンス懸念**。**JOINとFROM句サブクエリ**の組み合わせの方が推奨されることが多い。
* -- 相関副問い合わせ（分かりやすいが遅いかも）
* SELECT E1.氏名, E1.給与
* FROM 社員テーブル E1
* WHERE E1.給与 > (SELECT AVG(E2.給与) FROM 社員テーブル E2 WHERE E2.部署ID = E1.部署ID);
* -- JOIN + FROM句サブクエリ（少し複雑だが速いかも）
* SELECT E.氏名, E.給与
* FROM 社員テーブル E
* INNER JOIN (
* SELECT 部署ID, AVG(給与) AS DeptAvgSalary
* FROM 社員テーブル
* GROUP BY 部署ID
* ) AS DeptAvg ON E.部署ID = DeptAvg.部署ID

WHERE E.給与 > DeptAvg.DeptAvgSalary;

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

* **理由:**
  + 相関副問い合わせは直感的に書けるが、社員一人ひとりに対して平均給与計算のサブクエリが走るため、社員数が多いと非常に遅くなる可能性がある。
  + JOINとFROM句サブクエリを使う方法は、最初に部署ごとの平均給与をまとめて計算し、それを一時テーブルとして元の社員テーブルと結合するため、一般的に効率が良い。

### まとめ：どちらを選ぶか？

1. **基本はJOIN:** 複数のテーブルの情報を組み合わせて使いたい場合は、まずJOINを検討する。
2. **存在チェックならEXISTS:** 関連データがあるかないかだけを知りたい場合は、EXISTS (相関副問い合わせ) が有力。JOINより効率的で意図が明確なことが多い。NOT EXISTS も同様。
3. **行ごとの比較は注意:** 相関副問い合わせで実現できるが、パフォーマンスに注意が必要。JOINと集約関数（FROM句サブクエリなど）や、ウィンドウ関数（より高度な機能）で代替できないか検討する。

**応用情報技術者試験では:**

* EXISTS/NOT EXISTS が出てきたら、それは相関副問い合わせである可能性が極めて高い。
* 単純なデータ結合ならJOINが使われていることが多い。
* 両方の書き方ができる問題（例: ケース３）も出題される可能性があるので、どちらのパターンも読めるようにしておくことが重要。

相関副問い合わせは強力なツールですが、パフォーマンスの罠もあります。JOINとの違いと使い分けを理解することで、より適切で効率的なSQLを書けるようになりますよ！

edit

UNION、JOIN、サブクエリ（WHEREとFROMの二つの方法）、相関副問い合わせのそれぞれの存在意義を教えて。それぞれがないと、このようなことができないというような、その必要性がまだ理解できないでいるのです。教えてください。大先生！！

edit

よっしゃー！大先生に任せなさい！💪😎  
それぞれの構文が「なぜ必要なのか」「これがないと何が困るのか」という存在意義、しっかり解説しますよ！他のもので代用しようとすると、いかに大変か、あるいは不可能かが分かれば、それぞれの価値が見えてくるはずです！

### 1. JOIN の存在意義：テーブルを「横に」繋ぐ基本技！

* **これがないと困ること:**
  + **例：「社員リスト」と「部署リスト」から、「どの社員がどの部署名にいるか」を知りたい時。**
    - 社員テーブルには 部署ID しかなく、部署テーブルには 部署名 しかありません。
    - JOINがなければ、まず社員テーブルから全社員のリスト（氏名と部署ID）を取得し、次に部署テーブルから全部署のリスト（部署IDと部署名）を取得します。
    - そして、プログラム（例えばPythonやJavaなど）の中で、社員一人ひとりの 部署ID をキーにして、部署リストから対応する 部署名 を**自分で探してくっつける**処理を書かなければなりません。
    - データが大量にある場合、この「自分で探してくっつける」処理は非常に面倒で、遅くなります。
  + **JOINがあれば：** SELECT E.氏名, D.部署名 FROM 社員 E JOIN 部署 D ON E.部署ID = D.部署ID; の**一行**で、データベースシステムが最適化された方法で高速に結合してくれます。
* **存在意義:** 関連する情報を複数のテーブルから効率的に取得し、**あたかも最初から一つの大きな表だったかのように**扱えるようにすること。これがリレーショナルデータベースの基本であり、JOINはその心臓部です。プログラム側での面倒な結合処理を不要にします。

### 2. UNION の存在意義：検索結果を「縦に」積み重ねる！

* **これがないと困ること:**
  + **例：「東京支店の顧客リスト」と「大阪支店の顧客リスト」（テーブル構造は同じ）を一つにまとめて表示したい時。**
    - UNIONがなければ、まず東京支店の顧客リストを取得するSQLを実行し、次に大阪支店の顧客リストを取得するSQLを実行します。
    - そして、プログラム側で、**二つのリストを一つに連結する**処理を書く必要があります。もし重複を除きたいなら、さらに自分で重複チェックと削除の処理も書かなければなりません。
  + **UNIONがあれば：** SELECT ... FROM 東京顧客 UNION SELECT ... FROM 大阪顧客; のように書けば、データベースが自動的に二つの結果を**縦に連結し**、（UNIONなら）重複も除去して一つの結果セットとして返してくれます。
* **存在意義:** 構造が同じ複数のデータセットを、**SQLレベルで簡単に統合する**こと。別々のテーブルや、異なる条件で抽出した結果を、プログラム側の処理を介さずに一つにまとめられます。特に重複除去（UNION）や、重複を許す場合（UNION ALL）の選択肢があるのが便利です。

### 3. サブクエリ (WHERE句) の存在意義：動的な条件で絞り込む！

* **これがないと困ること:**
  + **例1：「平均点より高い点数を取った生徒」を知りたい時。**
    - サブクエリがなければ、まず SELECT AVG(点数) FROM 成績; を実行して平均点を調べ（例えば75点）、その結果を見てから、改めて SELECT 氏名 FROM 成績 WHERE 点数 > 75; というSQLを**実行し直す**必要があります。データが変われば平均点も変わるので、その都度SQLを書き換えるか、プログラムで2段階の処理が必要です。
  + **例2：「『飲料』カテゴリに属する商品の注文履歴」を知りたい時。**
    - サブクエリがなければ、まず SELECT 商品ID FROM 商品 WHERE カテゴリ = '飲料'; を実行して飲料のIDリスト（例: 101, 105, 108）を取得し、そのリストを使って SELECT \* FROM 注文明細 WHERE 商品ID IN (101, 105, 108); というSQLを**組み立てて実行する**必要があります。飲料カテゴリの商品が増減するたびに、このリストを作り直す手間がかかります。
  + **サブクエリがあれば：** WHERE 点数 > (SELECT AVG(点数) FROM 成績) や WHERE 商品ID IN (SELECT 商品ID FROM 商品 WHERE カテゴリ = '飲料') のように、**一つのSQL文の中で動的に条件を生成して**絞り込めます。
* **存在意義:** WHERE句で使う条件値を、**別のクエリの結果から動的に取得して利用する**こと。事前に値を調べてSQLに埋め込んだり、複数回のクエリ実行やプログラム側の処理を必要としたりすることなく、一つのSQLで完結させることができます。特に集計結果（平均、最大など）や、変動するリストを条件にしたい場合に不可欠です。

### 4. サブクエリ (FROM句) の存在意義：複雑な集計や加工を段階的に！

* **これがないと困ること:**
  + **例：「部署ごとの平均給与を計算し、その平均給与が350万円以上の部署名とその平均給与」を知りたい時。**
    - FROM句サブクエリがない場合、これをJOINだけでやろうとすると非常に複雑になるか、あるいは一度 GROUP BY で集計した結果を**一時テーブルとしてデータベースに作成**し、その一時テーブルと元のテーブルをJOINする、という複数のステップが必要になることがあります。
    - あるいは、まず部署ごとの平均給与リストを取得し、プログラム側で350万円以上の部署IDを特定し、再度部署テーブルに問い合わせて部署名を取得する、といった面倒な手順が必要です。
  + **FROM句サブクエリがあれば：** SELECT ... FROM (SELECT 部署ID, AVG(給与) AS 平均給与 FROM 社員 GROUP BY 部署ID) AS DeptAvg JOIN ... WHERE DeptAvg.平均給与 >= 3500000; のように、**SQL文の中で「部署ごとの平均給与」という中間的な結果セット（一時テーブルのようなもの）を作り**、それを普通のテーブルのように扱ってさらに処理（JOINやWHERE）を続けられます。
* **存在意義:** **SQLの実行途中で生成した結果セットを、あたかも最初から存在するテーブルかのように再利用する**こと。これにより、複雑なデータ集計や加工処理を、一つのSQL文の中で**段階的に、構造化して**記述できます。一時テーブルを明示的に作成・削除する手間が省け、SQL文として完結させられます。

### 5. 相関副問い合わせ の存在意義：行ごとの関連チェックをスマートに！

* **これがないと困ること:**
  + **例：「一度でも注文を担当したことのある社員」を知りたい時。**
    - EXISTS を使った相関副問い合わせがない場合、一つの方法は INNER JOIN を使うことです: SELECT DISTINCT E.氏名 FROM 社員 E INNER JOIN 注文 O ON E.社員ID = O.社員ID;。しかし、これだと一人の社員が何度も注文している場合に結果が重複するため DISTINCT が必要になり、無駄な結合処理が発生する可能性があります。
  + **例：「注文履歴のない社員」を知りたい時。**
    - NOT EXISTS がない場合、LEFT JOIN を使って SELECT E.氏名 FROM 社員 E LEFT JOIN 注文 O ON E.社員ID = O.社員ID WHERE O.注文ID IS NULL; と書くのが一般的です。これは有効な代替手段ですが、人によっては NOT EXISTS の方が「注文が存在しない」という意図が直接的で分かりやすいと感じるかもしれません。
  + **例：「自分の部署の平均給与より高い社員」を知りたい時。**
    - 相関副問い合わせがない場合、前述のFROM句サブクエリとJOINを使う方法で代替できますが、SQL文は少し長くなります。相関副問い合わせなら、比較対象が行ごとに変わることを直感的に表現できます（ただしパフォーマンスには注意）。
  + **相関副問い合わせがあれば：** WHERE EXISTS (SELECT ... WHERE O.社員ID = E.社員ID) のように、外側の行（社員E）ごとに、関連するデータ（注文O）が**存在するかどうか**を効率的にチェックできます。NOT EXISTS も同様です。比較の場合も WHERE 給与 > (SELECT AVG(...) WHERE ... = E.部署ID) と、行ごとの依存関係を直接的に書けます。
* **存在意義:** 外側のクエリで処理中の**各行に「関連して」**、サブクエリを実行し、条件（特に**存在有無**）を判定すること。EXISTS / NOT EXISTS は、関連データの有無だけをチェックしたい場合に、JOINよりも意図が明確で、かつ効率的な場合が多いです。また、「行ごとに変わる条件」での比較を（パフォーマンスの懸念はあれど）表現する手段を提供します。

どうでしょう？それぞれの構文が、他のものでは代替が難しかったり、非常に面倒になったりする「固有の役割」を持っていることがイメージできたでしょうか？

* **JOIN:** 横につなぐ基本！
* **UNION:** 縦につなぐ！
* **サブクエリ(WHERE):** 動的な条件を作る！
* **サブクエリ(FROM):** 中間結果を再利用！
* **相関副問い合わせ:** 行ごとの関連チェック（特に存在有無）！

これらはSQLの強力な武器であり、それぞれが解決してくれる問題があるからこそ、存在しているのです！