SQL実装ガイド

[【前提】 2](#_Toc165557109)

[【SQL規約】 3](#_Toc165557110)

[1. コーディング規約 3](#_Toc165557111)

[2. テーブル結合の方法 5](#_Toc165557112)

[2.1. JOINの方法 6](#_Toc165557113)

[2.2. UNIONの方法 7](#_Toc165557114)

[3. データ検索の方法 8](#_Toc165557115)

[3.1. EXISTSの適用 9](#_Toc165557116)

[3.2. BETWEEN演算子の適用 10](#_Toc165557117)

[3.3. IN演算子の適用 11](#_Toc165557118)

[3.4. 正規表現の適用 12](#_Toc165557119)

[4. データ集約の方法 14](#_Toc165557120)

[4.1. GROUP BYとDISTINCT 15](#_Toc165557121)

[4.2. HAVING句の利用 17](#_Toc165557122)

[5. 副問い合わせ（サブクエリ）の方法 18](#_Toc165557123)

[5.1. 副問い合わせの適用ケース（テーブル結合時） 19](#_Toc165557124)

[5.2. 副問い合わせの適用ケース（データ集計時） 20](#_Toc165557125)

[5.3. CASE文の適用ケース（データ集計時） 22](#_Toc165557126)

[【有用な関数等の補足説明】 23](#_Toc165557127)

[1. COALESCE関数 24](#_Toc165557128)

[2. 集計関数COUNT+DISTINCTの利用 25](#_Toc165557129)

[3. LIMITとOFFSET 26](#_Toc165557130)

[4. ROW\_NUMBER関数とRANK関数 27](#_Toc165557131)

[5. STRING\_AGG関数 31](#_Toc165557132)

[6. UNNEST関数＋ARRAY関数 32](#_Toc165557133)

## 【前提】

当資料では以下を前提に説明する。

* SQLの基本的な構文（SELECT文やWHERE句、AND、OR、LIKE演算子）は利用できる程度のレベルを想定している。
* 当SQL規約は、記述の平準化を主な目的としている。  
  パフォーマンスよりも上記を考慮してSQLを記述する方針としている。  
  ※パフォーマンスは設計側で考慮するべきとの方針

可読性を考慮するとより簡易な記述方法が存在するケースも考えられるが、他メンバーが適切に流用したりすることも考慮し、平準化を重視して規約を作成している。

## 【SQL規約】

### コーディング規約

当プロジェクトでのSQLのコーディング規約は以下の通りとする。

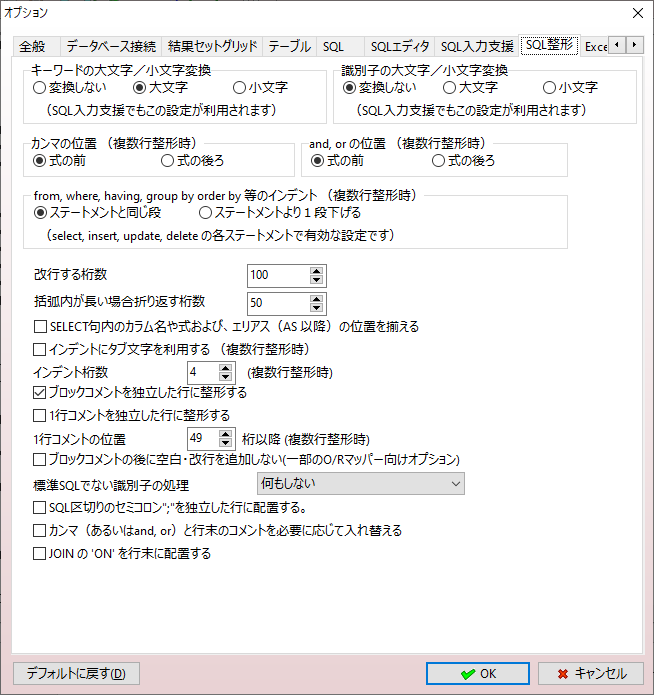
* A5SQLのフォーマッター機能（ctrl+q）を利用する。  
  これにより規約のうちWITH句での分割や別名以外については基本的に順守された状態になる。  
  A5SQLの設定方法については、「（参考）A5SQL「SQL整形」の設定」を参照。
* 予約語/関数は大文字にする。
* 2スペース（または4スペース）でインデントを揃える。  
  ※スペースの数はフォーマッターに依存するため、1SQL内で統一されていれば問題ない
* SQLの特定の予約語の後は改行する。
* 1カラム1行にする。
* 条件のAND / ORは先頭に置く。
* サブクエリは字下げする。
* SQL全体でサブクエリが3段階を超える場合は必ずWITHを使って分割する。
* 別名を付与する際、ASは省略しない
* 別名は意味が分かる名称にする。（「a」や「b」のような意味のない名称は避ける）
* SELECT句でのCASE, COUNTなどの関数を使ったカラムは別名をつける。

（参考）コーディング規約参考URL

<https://qiita.com/taise/items/18c14d9b01a5dfd6d35e>。

（参考）A5SQL「SQL整形」の設定

デフォルトの設定から「キーワードの大文字／小文字変換」を「大文字」に変更することで、コーディング規約ののうちWITH句での分割や別名以外については基本的に順守された状態になる。



### テーブル結合の方法

テーブル結合に関するSQL規約を説明する。説明内容とそのサマリは以下の通り。

1. JOINの方法
   1. 必ずLEFT JOINを用い、RIGHT JOINは適用しないこと
   2. 自然結合（NATURAL）は利用しないこと
2. UNIONの方法
   1. 必ずUNION ALLを用い、UNIONは適用しないこと

#### JOINの方法

OUTER JOINの方法としては、LEFT JOINとRIGHT JOINが存在する。

必ずLEFT JOINを用い、RIGHT JOINは適用しないこと。また「OUTER」の記述は省略すること。

|  |
| --- |
| 【修正前】  WITH test1 AS (   SELECT  '1' AS id  , 'a' AS data1   UNION ALL   SELECT  '2' AS id  , 'x' AS data1 )  , test2 AS (SELECT '1' AS id, 'd' AS data1)  SELECT  \*  FROM  test2   RIGHT OUTER JOIN test1   ON test1.id = test2.id;  【修正後】  WITH test1 AS (   SELECT  '1' AS id  , 'a' AS data1   UNION ALL   SELECT  '2' AS id  , 'x' AS data1 )  , test2 AS (SELECT '1' AS id, 'd' AS data1)  SELECT  \*  FROM  test1   LEFT JOIN test2   ON test1.id = test2.id; |

自然結合（NATURAL）は利用しないこと。

自然結合が利用できる際はON句を省略できるメリットはあるが、結合するテーブルのカラムを全て理解しておく必要があるため、客観的に妥当性が確認しにくい点を考慮して利用不可とする。

#### UNIONの方法

統合結合（UNION）の方法としては、UNIONとUNION ALLが存在する。

必ずUNION ALLを用い、UNIONは適用しないこと。

UNIONを適用することで自動的にDISTINCTされるメリットがあるが、結合後にDISTINCTあるいはGROUP BYなどで集約方法を明示的に行うことで、暗黙的にDISTINCTされたことによる集計誤りを防止できるようにする。

|  |
| --- |
| 【修正前】  WITH test1 AS (   SELECT  '1' AS id  , 'a' AS data1   UNION ALL   SELECT  '2' AS id  , 'x' AS data1 )  , test2 AS (SELECT '1' AS id, 'd' AS data1)  SELECT  \*  FROM  test2   RIGHT OUTER JOIN test1   ON test1.id = test2.id;  【修正後】  WITH test1 AS (   SELECT  '1' AS id  , 'a' AS data1   UNION ALL   SELECT  '2' AS id  , 'x' AS data1 )  , test2 AS (SELECT '1' AS id, 'd' AS data1)  SELECT  \*  FROM  test1   LEFT JOIN test2   ON test1.id = test2.id; |

### データ検索の方法

データ検索（WHERE句）に関するSQL規約を説明する。説明内容とそのサマリは以下の通り。

1. EXISTSの適用
   1. EXISTSが適用できる際は必ず適用すること
2. BETWEEN演算子の適用
   1. BETWEEN演算子が適用できる際は必ず適用すること
3. IN演算子の適用
   1. 同一カラムに対する複数検索条件を指定する際にIN演算子を適用すること  
      ※ただし条件数が概ね10を超えるようなケースでは別テーブルに格納して

EXISTSを適用するなど、設計レベルで検討すること

1. 正規表現の適用
   1. 下記資料を参照すること  
      [\\es44.fs-bxo.nttdata.co.jp\gabu\06\_医薬\10\_千年カルテプロジェクト\02\_MIS\共通\20\_デリバリ\34\_運用改善\01\_案件改善ワーキング\FY2022\007\_正規表現\_プロセス標準化\PostgresSQLでの正規表現によるテキストデータ抽出方法.docx](file:///\\es44.fs-bxo.nttdata.co.jp\gabu\06_医薬\10_千年カルテプロジェクト\02_MIS\共通\20_デリバリ\34_運用改善\01_案件改善ワーキング\FY2022\007_正規表現_プロセス標準化\PostgresSQLでの正規表現によるテキストデータ抽出方法.docx)

#### EXISTSの適用

抽出元テーブルの絞り込み条件に別テーブルを使用する場合、別テーブルから項目を抽出しない場合はINNER JOINではなくEXISTSを使用すること。

EXISTS区内で参照したテーブルのカラムはSELECT句で適用できないため、FROM句のテーブルのカラムのみ適用されることが自明になる点で可読性が高い。

|  |
| --- |
| 【修正前】 WITH test1 AS (   SELECT  '1' AS id  , 'a' AS data1   UNION ALL   SELECT  '2' AS id  , 'x' AS data1 )  , test2 AS (SELECT '1' AS id, 'd' AS data1)  SELECT  test1.id  , test1.data1  FROM  test1   INNER JOIN test2   ON test1.id = test2.id;   【修正後】 WITH test1 AS (   SELECT  '1' AS id  , 'a' AS data1   UNION ALL   SELECT  '2' AS id  , 'x' AS data1 )  , test2 AS (SELECT '1' AS id, 'd' AS data1)  SELECT  test1.id  , test1.data1  FROM  test1  WHERE  EXISTS (SELECT 1 FROM test2 WHERE test1.id = test2.id); |

#### BETWEEN演算子の適用

抽出条件に範囲指定を行う場合、BETWEEN演算子が適用できる際は必ず適用すること。

BETWEEN演算子を適用することで対象のカラムの記載箇所が1か所になるため、範囲指定をしていることが明確になる点で可読性が高い。

|  |
| --- |
| 【修正前】  WITH test1 AS (   SELECT  '1' AS id  , '2023/01/01' AS data1   UNION ALL   SELECT  '11' AS id  , '2024/01/01' AS data1 )  SELECT  \*  FROM  test1  WHERE  test1.data1 >= '2023/01/01'   AND test1.data1 <= '2023/12/31';   【修正後】  WITH test1 AS (   SELECT  '1' AS id  , '2023/01/01' AS data1   UNION ALL   SELECT  '11' AS id  , '2024/01/01' AS data1 )  SELECT  \*  FROM  test1  WHERE  test1.data1 BETWEEN '2023/01/01' AND '2023/12/31'; |

#### IN演算子の適用

同一カラムに対する複数検索条件を指定する際にIN演算子を適用すること。

OR演算子により同一条件での抽出をすることもできるが、同一カラムに対する複数条件での抽出することが明確になる点で可読性が高い。

|  |
| --- |
| 【修正前】  WITH test1 AS (   SELECT  '1' AS id  , '2023/01/01' AS data1   UNION ALL   SELECT  '11' AS id  , '2024/01/01' AS data1 )  SELECT  \*  FROM  test1  WHERE  test1.data1 = '2023/01/01'   OR test1.data1 = '2023/02/01'   OR test1.data1 = '2023/03/01';   【修正後】  WITH test1 AS (   SELECT  '1' AS id  , '2023/01/01' AS data1   UNION ALL   SELECT  '11' AS id  , '2024/01/01' AS data1 )  SELECT  \*  FROM  test1  WHERE  test1.data1 IN ('2023/01/01', '2023/02/01', '2023/03/01'); |

#### 正規表現の適用

千年カルテシステムの本番環境ではローカル文字タイプ（「SHOW LC\_CTYPE」で確認可能）は「Japanese\_Japan.932」ではなく「C」（ロケール設定無効）である。

そのため、「全角を含む」「大文字小文字の区別なし」などの言語に依存する判定が行われない点に留意する必要がある。

* マルチバイト文字の考慮誤りによる抽出漏れの例

全角文字はマルチバイト文字であり大文字小文字の判定ができないため、正規表現で大文字小文字の区別なしで抽出する「~\*」を指定しても、全角小文字の「ａ病」として登録されているid「22」の1レコードは抽出できない。

|  |
| --- |
| WITH test1 AS (   SELECT  '1' AS id  , 'A病' AS data1   UNION ALL   SELECT  '11' AS id  , 'a病' AS data1   UNION ALL   SELECT  '2' AS id  , 'Ａ病' AS data1   UNION ALL   SELECT  '22' AS id  , 'ａ病' AS data1 ) SELECT  \*  FROM  test1  WHERE  --idが1と11と2の3レコードが抽出される  test1.data1 ~\* 'A|Ａ'; |

* マルチバイト文字を考慮した例

半角と全角の大文字「A」に加え、全角の小文字「ａ」を指定すれば、全角半角大文字小文字を区別せず「A」から始まるレコード(全4レコード)を抽出することができる。

|  |
| --- |
| WITH test1 AS (   SELECT  '1' AS id  , 'A病' AS data1   UNION ALL   SELECT  '11' AS id  , 'a病' AS data1   UNION ALL   SELECT  '2' AS id  , 'Ａ病' AS data1   UNION ALL   SELECT  '22' AS id  , 'ａ病' AS data1 ) SELECT  \*  FROM  test1  WHERE  --全レコードが抽出される  test1.data1 ~\* 'A|Ａ|ａ'; |

### データ集約の方法

データ集約に関するSQL規約を説明する。説明内容とそのサマリは以下の通り。

1. GROUP BYとDISTINCTの利用
   1. DISTINCTが適用できる際は必ず適用すること
2. HAVING句の利用
   1. HAVING句が適用できる際は必ず適用すること  
      ※副問い合わせでの集計＋WHEREでの絞り込みと比較して可読性が高い

#### GROUP BYとDISTINCT

DISTINCTは全カラムを対象として重複を除外したい場合に使用する。

以下のテストデータに対して、5レコードが登録(うちidが「3」の2レコードは全カラムで重複)されているものに対して全カラムを対象として重複を除外して抽出したい場合、例1のようにDISTINCTを使用することで重複を除外して抽出できる。

一方、idとtest\_dateの2カラムに加え、data1に対して集約関数であるMIN関数を適用したい場合、例2のようにGROUP BYを使用することで重複を除外しつつ集約関数を用いた抽出ができる。

GROUP BYは重複の判定項目が全カラム以外の場合でも適用できるため、用途が広い。

|  |
| --- |
| 【テストデータ】 WITH test1 AS (   SELECT  '1' AS id  , '2023/01/01' AS test\_date  , 100 AS data1   UNION ALL   SELECT  '2' AS id  , '2023/02/01' AS test\_date  , 50 AS data1   UNION ALL   SELECT  '2' AS id  , '2023/02/01' AS test\_date  , 200 AS data1   UNION ALL   SELECT  '3' AS id  , '2023/03/01' AS test\_date  , 75 AS data1   UNION ALL   SELECT  '3' AS id  , '2023/03/01' AS test\_date  , 75 AS data1 ) 【例1】 SELECT DISTINCT  test1.id  , test1.test\_date  , test1.data1  FROM  test1;   【例2】 SELECT  test1.id  , test1.test\_date  , MIN(test1.data1) AS min\_test\_data  FROM  test1  GROUP BY  test1.id  , test1.test\_date; |

#### HAVING句の利用

以下のテストデータに対して、idごとのdata1の数が2を超過(3以上)するレコードを抽出したい場合、修正前のように集計結果に対してWHERE句で絞り込みを行うより、修正後のようにHAVING句を用いた方が可読性が高い。

|  |
| --- |
| 【テストデータ】 WITH test1 AS (   SELECT  '1' AS id  , 'a' AS data1   UNION ALL   SELECT  '1' AS id  , 'b' AS data1   UNION ALL   SELECT  '1' AS id  , 'c' AS data1   UNION ALL   SELECT  '2' AS id  , 'x' AS data1 ) 【修正前】  SELECT  id  FROM  (   SELECT  id  , count(data1) AS data\_count   FROM  test1   GROUP BY  id  ) AS temp  WHERE  temp.data\_count > 2;   【修正後】  SELECT  id  FROM  test1  GROUP BY  id  HAVING  count(data1) > 2; |

### 副問い合わせ（サブクエリ）の方法

副問い合わせ（サブクエリ）に関するSQL規約を説明する。説明内容とそのサマリは以下の通り。

1. 副問い合わせの適用ケース（テーブル結合時）

大量データのテーブル同士を結合する際は、それぞれのテーブルに対して事前に副問い合わせで条件検索などを行ってレコード数を削減しておくこと  
※可読性よりパフォーマンスを意識しなければならないケース

1. 副問い合わせの適用ケース（データ集計時）

大量データのテーブルから複数の集計結果を算出する際に集約単位が同一である場合は、副問い合わせを利用するのではなく、CASE文などによる集約＋集計関数で対応できないか検討すること

1. CASE文の適用ケース（データ集計時）

集約を同一条件で実施できる場合は、CASE文の方が集約条件の記述が1か所に集約されるため、可読性も高い。部分的に集約条件が同一の場合も、同一の箇所をCASE文で対応した後にJOINや副問い合わせで対応する方法もある。

#### 副問い合わせの適用ケース（テーブル結合時）

千年カルテ環境において、特にMMLの検歴情報モジュールに関するテーブルについてはデータ量が多いため、以下の例のように同モジュール内の他テーブルと結合する際には事前に絞り込みを行い、可能な限りレコード数を少なくした上でテーブル結合を行うこと。

また、結合に使用する項目や抽出したい項目以外の不要な項目は指定しないこと。

|  |
| --- |
| 【例】 WITH text\_mml\_common AS (   SELECT  zip\_no  , file\_no  , body\_no   FROM  mml\_common   WHERE  -- モジュール種別がmmlLbかつ最新フラグがTRUE  mml\_type = 'mmlLb'   AND new\_f = TRUE )  , text\_mmllb\_body\_temp AS (   SELECT  zip\_no  , file\_no  , body\_no  , facility\_id\_mml  , master\_id  , COALESCE(sample\_time, '') AS sample\_time   FROM  mml\_mmllb\_body   WHERE  -- 報告状態が「最終報告」  report\_status = '最終報告'   OR status\_code = 'final' )  , text\_mmllb\_body AS (   SELECT  text\_mmllb\_body\_temp.zip\_no  , text\_mmllb\_body\_temp.file\_no  , text\_mmllb\_body\_temp.body\_no  , text\_mmllb\_body\_temp.facility\_id\_mml  , text\_mmllb\_body\_temp.master\_id  , text\_mmllb\_body\_temp.sample\_time   FROM  text\_mmllb\_body\_temp   WHERE  EXISTS (   SELECT  1   FROM  text\_mml\_common   WHERE  text\_mmllb\_body\_temp.zip\_no = text\_mml\_common.zip\_no   AND text\_mmllb\_body\_temp.file\_no = text\_mml\_common.file\_no   AND text\_mmllb\_body\_temp.body\_no = text\_mml\_common.body\_no  ) )  , text\_mmllb\_labo\_test\_temp AS (   SELECT  zip\_no  , file\_no  , body\_no  , facility\_id\_mml  , master\_id  , labo\_test\_no  , COALESCE(specimen\_name, '') AS specimen\_name  , COALESCE(sp\_code, '') AS sp\_code   FROM  mml\_mmllb\_labo\_test )  , text\_mmllb\_labo\_test AS (   SELECT  text\_mmllb\_labo\_test\_temp.zip\_no  , text\_mmllb\_labo\_test\_temp.file\_no  , text\_mmllb\_labo\_test\_temp.body\_no  , text\_mmllb\_labo\_test\_temp.facility\_id\_mml  , text\_mmllb\_labo\_test\_temp.master\_id  , text\_mmllb\_labo\_test\_temp.labo\_test\_no  , text\_mmllb\_labo\_test\_temp.specimen\_name  , text\_mmllb\_labo\_test\_temp.sp\_code   FROM  text\_mmllb\_labo\_test\_temp   WHERE  EXISTS (   SELECT  1   FROM  milscm\_2023\_009.sny\_202310\_mt\_kensa AS mt\_kensa   WHERE  text\_mmllb\_labo\_test\_temp.facility\_id\_mml = mt\_kensa.facility\_id\_mml   AND text\_mmllb\_labo\_test\_temp.specimen\_name = mt\_kensa.specimen\_name   AND text\_mmllb\_labo\_test\_temp.sp\_code = mt\_kensa.sp\_code  ) )  , text\_mmllb\_item\_temp AS (   SELECT  zip\_no  , file\_no  , body\_no  , facility\_id\_mml  , master\_id  , labo\_test\_no  , COALESCE(item\_name, '') AS item\_name  , COALESCE(item\_code, '') AS facility\_it\_code  , COALESCE(unit, '') AS unit  , COALESCE(value, '') AS value  , COALESCE(num\_value, '') AS num\_value   FROM  mml\_mmllb\_item )  , text\_mmllb\_item AS (   SELECT  text\_mmllb\_item\_temp.zip\_no  , text\_mmllb\_item\_temp.file\_no  , text\_mmllb\_item\_temp.body\_no  , text\_mmllb\_item\_temp.facility\_id\_mml  , text\_mmllb\_item\_temp.master\_id  , text\_mmllb\_item\_temp.labo\_test\_no  , text\_mmllb\_item\_temp.item\_name  , text\_mmllb\_item\_temp.facility\_it\_code  , text\_mmllb\_item\_temp.unit  , text\_mmllb\_item\_temp.value  , text\_mmllb\_item\_temp.num\_value   FROM  text\_mmllb\_item\_temp   WHERE  EXISTS (   SELECT  1   FROM  milscm\_2023\_009.sny\_202310\_mt\_kensa AS mt\_kensa   WHERE  text\_mmllb\_item\_temp.facility\_id\_mml = mt\_kensa.facility\_id\_mml   AND text\_mmllb\_item\_temp.item\_name = mt\_kensa.item\_name   AND text\_mmllb\_item\_temp.facility\_it\_code = mt\_kensa.item\_code   AND text\_mmllb\_item\_temp.unit = mt\_kensa.unit  ) )  , mml\_lb\_join AS (   SELECT  text\_mmllb\_body.facility\_id\_mml  , text\_mmllb\_body.master\_id  , text\_mmllb\_labo\_test.specimen\_name  , text\_mmllb\_labo\_test.sp\_code  , text\_mmllb\_item.item\_name  , text\_mmllb\_item.facility\_it\_code  , text\_mmllb\_item.unit  , text\_mmllb\_body.sample\_time  , text\_mmllb\_item.value  , text\_mmllb\_item.num\_value   FROM  text\_mmllb\_body   INNER JOIN text\_mmllb\_labo\_test   ON (   text\_mmllb\_body.zip\_no = text\_mmllb\_labo\_test.zip\_no   AND text\_mmllb\_body.file\_no = text\_mmllb\_labo\_test.file\_no   AND text\_mmllb\_body.body\_no = text\_mmllb\_labo\_test.body\_no  )   INNER JOIN text\_mmllb\_item   ON (   text\_mmllb\_labo\_test.zip\_no = text\_mmllb\_item.zip\_no   AND text\_mmllb\_labo\_test.file\_no = text\_mmllb\_item.file\_no   AND text\_mmllb\_labo\_test.body\_no = text\_mmllb\_item.body\_no   AND text\_mmllb\_labo\_test.labo\_test\_no = text\_mmllb\_item.labo\_test\_no  ) )  SELECT  \*  FROM  mml\_lb\_join; |

#### 副問い合わせの適用ケース（データ集計時）

以下のテストデータに対して、s\_cdとss\_cdのように複数カラムの特定の値(以下の場合s\_cdが「A01」とss\_cdが「A011」)を対象として集計する場合、例1のような副問い合わせを利用する。

|  |
| --- |
| 【テストデータ】  WITH test1 AS (   SELECT  '1' AS id  , 'A01' AS s\_cd  , 'A011' AS ss\_cd   UNION ALL   SELECT  '1' AS id  , 'A01' AS s\_cd  , 'A012' AS ss\_cd   UNION ALL   SELECT  '2' AS id  , 'A01' AS s\_cd  , 'A012' AS ss\_cd   UNION ALL   SELECT  '3' AS id  , 'A01' AS s\_cd  , 'A011' AS ss\_cd   UNION ALL   SELECT  '4' AS id  , 'B02' AS s\_cd  , 'B021' AS ss\_cd )  【例1】 SELECT  test1.id  , (   SELECT  count(sub1.s\_cd)   FROM  test1 AS sub1   WHERE  sub1.s\_cd = 'A01'   AND test1.id = sub1.id  ) AS s\_cd\_A01\_count  , (   SELECT  count(sub1.ss\_cd)   FROM  test1 AS sub1   WHERE  sub1.ss\_cd = 'A011'   AND test1.id = sub1.id  ) AS ss\_cd\_A011\_count  FROM  test1  GROUP BY  test1.id; |

#### CASE文の適用ケース（データ集計時）

以下のテストデータに対して、s\_cdのように同一カラムに対して複数の値(以下の場合、s\_cdが「A01」と「B02」のもの)を条件として集計する場合、例1のようなSUM関数＋CASE文を用いた方が記述が1か所に集約され、可読性も高い。

|  |
| --- |
| 【テストデータ】   WITH test1 AS (   SELECT  '1' AS id  , 'A01' AS s\_cd  , 'A011' AS ss\_cd   UNION ALL   SELECT  '1' AS id  , 'A01' AS s\_cd  , 'A012' AS ss\_cd   UNION ALL   SELECT  '2' AS id  , 'A01' AS s\_cd  , 'A012' AS ss\_cd   UNION ALL   SELECT  '3' AS id  , 'A01' AS s\_cd  , 'A011' AS ss\_cd   UNION ALL   SELECT  '4' AS id  , 'B02' AS s\_cd  , 'B021' AS ss\_cd )  【例1】 SELECT  test1.id  , sum(CASE WHEN test1.s\_cd = 'A01' THEN 1 ELSE 0 END) AS s\_cd\_A01\_count  , sum(CASE WHEN test1.s\_cd = 'B02' THEN 1 ELSE 0 END) AS s\_cd\_B01\_count  FROM  test1  GROUP BY  test1.id  ORDER BY  test1.id; |

## 【有用な関数等の補足説明】

SQLを作成する上で有用な関数やSQLの記述方法について説明する。  
説明内容とそのサマリは以下の通り。

1. COALESCE関数

第一引数の値がNULLの場合、第二引数の値を返却する関数。

CASE文での記述と比較して、可読性が高い。

1. 集計関数COUNT+DISTINCTの利用

ユニーク患者数の集計などを行う際に有用。

副問い合わせを1つ減らせるため、パフォーマンスが良い。

1. LIMITとOFFSET

集計結果が大量なレコードに対して、データの中身を目視確認したい際に有用。

（SQL動作確認や抽出結果の最低限の確認などに有用）

1. ROW\_NUMBER関数とRANK関数

特定条件での最新のデータを取得する際や特定条件での順序などを参照する際に有用。  
副問い合わせ＋集約関数（MIN,MAX）やORDER BY＋LIMITより、基本的にパフォーマンスが良い上、可読性も高い。

1. STRING\_AGG関数

特定条件でのカラムの値の一覧（＝分布）を確認したりする際に有用。

1. UNNEST関数＋ARRAY関数

横持ちのデータを縦持ちに変換する際に有用。

一般的にはUNION ALLを適用することが多いと考えられるが、基本的にパフォーマンスが良い上、可読性も高い。

（参考）縦持ちのデータを横持ちに変換する際はCASE文が一般的である。

### COALESCE関数

以下のテストデータに対して、s\_cdのNULLを置換したい場合、例1のようなCASE文を用いたNULLの置換を行うよりも、例2のCOALESCE関数を用いた方が可読性が高い。

|  |
| --- |
| 【テストデータ】 WITH test1 AS (   SELECT  '1' AS id  , 'A01' AS s\_cd  , 'A011' AS ss\_cd   UNION ALL   SELECT  '2' AS id  , NULL AS s\_cd  , 'A022' AS ss\_cd   UNION ALL   SELECT  '3' AS id  , NULL AS s\_cd  , NULL AS ss\_cd )  【例1】 SELECT  test1.id  , (   CASE   WHEN test1.s\_cd IS NULL   THEN '0'   ELSE test1.s\_cd   END  ) AS s\_cd  , (   CASE   WHEN test1.ss\_cd IS NULL   THEN '0'   ELSE test1.ss\_cd   END  ) AS ss\_cd  FROM  test1;   【例2】 SELECT  test1.id  , COALESCE(test1.s\_cd, '0') AS s\_cd  , COALESCE(test1.ss\_cd, '0') AS ss\_cd  FROM  test1; |

### 集計関数COUNT+DISTINCTの利用

以下のテストデータに対して、group\_idごとにidのユニーク件数を取得したい場合、例1のように副問い合わせを行わずとも、例2のようにCOUNT+DISTINCTを用いることでパフォーマンスも可読性も良くなる。

|  |
| --- |
| 【テストデータ】  WITH test1 AS (   SELECT  'X01' AS group\_id  , '1' AS id   UNION ALL   SELECT  'X01' AS group\_id  , '1' AS id   UNION ALL   SELECT  'X01' AS group\_id  , '2' AS id   UNION ALL   SELECT  'X11' AS group\_id  , '1' AS id )  【例1】 SELECT  test1\_count.group\_id  , count(\*) AS unique\_id  FROM  (   SELECT  group\_id  , id   FROM  test1   GROUP BY  group\_id  , id  ) AS test1\_count  GROUP BY  test1\_count.group\_id;   【例2】  SELECT  test1.group\_id  , count(DISTINCT id) AS unique\_id  FROM  test1  GROUP BY  test1.group\_id; |

### LIMITとOFFSET

・LIMIT関数の使用例

LIMITを指定することで出力するレコード数を制御できる。

以下のテストデータのように数百万レコードあるテーブルに対して、例1のように条件を指定しない単純な抽出処理を行うと表示までに時間がかかってしまうが、例2のようにLIMITを指定することで高速で表示することができる。

SQLの動作確認や、データのサンプリング調査を行いたい場合にLIMIT関数を用いるとよい。

|  |
| --- |
| 【テストデータ】  WITH test1 AS (   SELECT  'X01' AS group\_id  , '1' AS id   UNION ALL -- 数百万レコード分を想定  SELECT  'X99' AS group\_id  , '99999999' AS id )  【例1】 SELECT  \*  FROM  test1  ORDER BY  test1.group\_id  , id;   【例2】  SELECT  \*  FROM  test1  ORDER BY  test1.group\_id  , id LIMIT 1; |

・LIMIT関数とOFFSET関数の組み合わせた使用例

先述の通りLIMITで出力するレコード数を制御できるが、OFFSETは先頭からスキップしたいレコード数を指定することができる。そのため、LIMIT関数とOFFSET関数を組み合わせて使用することで指定した区間のデータ抽出ができる。

以下のテストデータに対して、group\_idとidを昇順で並び替えを行い10001レコード目から100レコード分の確認を行いたい場合、例1のように特に条件を指定せず全出力されたものに対して確認を行うより、例2のようにLIMITとOFFSETを用いて先頭から10000レコードをスキップして10001レコード目から10100レコード目までを出力した方が表示までの時間も短縮でき、処理負荷を少なくできる。

その他、100件程度の簡単な目視確認などを行いたい際にLIMITを活用するとよい。

なお、LIMITとOFFSETを使用する場合は予めORDER BY句で出力する順番を制御する必要がある。

(ORDER BYを指定しないと出力順がランダムとなり、想定した確認ができなくなるため)

|  |
| --- |
| 【テストデータ】  WITH test1 AS (   SELECT  'X01' AS group\_id  , '1' AS id   UNION ALL -- 数百万レコード分を想定  SELECT  'X99' AS group\_id  , '99999999' AS id )  【例1】 SELECT  \*  FROM  test1  ORDER BY  test1.group\_id  , id;   【例2】  SELECT  \*  FROM  test1  ORDER BY  test1.group\_id  , id LIMIT 100 OFFSET 10000; |

### ROW\_NUMBER関数とRANK関数

・ROW\_NUMBER関数の使用例

以下のテストデータに対して、kensaテーブルにあるIDごとの最新のkensa\_dateに紐づくkensa\_valueを抽出したい場合、例1のようにrow\_numberを指定することでIDごとにkensa\_dateの値の大きいもの(=最新のもの)から順にrow\_noを採番し、その結果に対して抽出条件にrow\_no = 1を指定することで想定したレコードのみを抽出することができる。

このように同一テーブルでソート順に依存した結果における別カラムの値を取得する際にROW\_NUMBER関数を用いるとよい。

(今回の場合、kensa\_dateのソート順に依存した結果におけるkensa\_valueの値を取得した)

|  |
| --- |
| 【テストデータ】  WITH kensa AS (   SELECT  '1' AS id  , '20200101' AS kensa\_date  , '60.0' AS kensa\_value   UNION ALL   SELECT  '1' AS id  , '20200115' AS kensa\_date  , '65.0' AS kensa\_value   UNION ALL   SELECT  '2' AS id  , '20230101' AS kensa\_date  , '80.0' AS kensa\_value   UNION ALL   SELECT  '3' AS id  , '20210101' AS kensa\_date  , '50.0' AS kensa\_value   UNION ALL   SELECT  '3' AS id  , '20210131' AS kensa\_date  , '30.0' AS kensa\_value )  【例1】  SELECT  id  , kensa\_date  , kensa\_value  , row\_no  FROM  (   SELECT  kensa.id  , kensa.kensa\_date  , kensa.kensa\_value  , ROW\_NUMBER() OVER (   PARTITION BY  kensa.id   ORDER BY  kensa.kensa\_date DESC  ) AS row\_no   FROM  kensa  ) AS kensa\_date\_sort  WHERE  kensa\_date\_sort.row\_no = 1; |

・RANK関数の使用例

以下のテストデータに対して、TOP3までのscore(同率の場合は横並び順位として抽出する)のidとscoreを抽出したい場合、例1はORDER BY＋LIMITを用いて上位3レコードに絞って抽出しているが、id「3」と「4」はscore「78」を横並びで上位3番目となっており、片方しか抽出できず、想定した動きとならないためNG。

例2ではRANK関数を用いることで同率の場合を考慮してscore\_rankが採番されるため、score\_rank >= 3と指定することでid「1」と「2」と「3」と「4」の4レコードが抽出でき、想定した動きを満たすことができる。

このように同率順位を考慮した特定順位までの抽出を行う場合にRANK関数の使用を検討するとよい。

|  |
| --- |
| 【テストデータ】  WITH score AS (   SELECT  '1' AS id  , '90' AS score   UNION ALL   SELECT  '2' AS id  , '80' AS score   UNION ALL   SELECT  '3' AS id  , '78' AS score   UNION ALL   SELECT  '4' AS id  , '78' AS score   UNION ALL   SELECT  '5' AS id  , '75' AS score )  【例1】  SELECT  id  , score  FROM  score  ORDER BY  score DESC  LIMIT  3;   【例2】  SELECT  score\_rank.id  , score\_rank.score  FROM  (   SELECT  id  , score  , rank() OVER (ORDER BY score DESC) AS score\_rank   FROM  score  ) AS score\_rank  WHERE  score\_rank.score\_rank <= 3; |

### STRING\_AGG関数

以下のテストデータに対して、idごとのshohou\_nameを目視確認したい場合、例1のようにSTRING\_AGG関数を用いることでshohouテーブルにあるidごとに全てのshohou\_nameをリスト化して抽出することができる。今回の場合、リスト化する際に「\_」を用いて結合を行っているため、例えばid「1」のshohou\_name\_listは「薬剤A\_薬剤B」のように出力される。

このように特定項目に対して目視確認を行う場合にSTRING\_AGG関数の使用を検討するとよい。

|  |
| --- |
| 【テストデータ】  WITH shohou AS (   SELECT  '1' AS id  , '薬剤A' AS shohou\_name   UNION ALL   SELECT  '1' AS id  , '薬剤B' AS shohou\_name   UNION ALL   SELECT  '2' AS id  , '薬剤B' AS shohou\_name   UNION ALL   SELECT  '3' AS id  , '薬剤C' AS shohou\_name   UNION ALL   SELECT  '3' AS id  , '薬剤D' AS shohou\_name )  SELECT  shohou.id  , string\_agg(   shohou.shohou\_name  , '\_'   ORDER BY  shohou.shohou\_name  ) AS shohou\_name\_list  FROM  shohou  GROUP BY  shohou.id; |

### UNNEST関数＋ARRAY関数

横→縦変換は一般的にUNION（UNION ALL）を用いて行うが、UNNEST関数＋ARRAY関数を用いて変換を行うことができる。

UNNEST関数＋ARRAY関数を用いた方が、FROM句が同一であることを明示できるため、可読性が高い。

|  |
| --- |
| 【テストデータ】 WITH height\_weight AS (   SELECT  '1' AS id  , '170' AS height  , '60' AS weight   UNION ALL   SELECT  '2' AS id  , '175' AS height  , '55' AS weight   UNION ALL   SELECT  '3' AS id  , '180' AS height  , '80' AS weight   UNION ALL   SELECT  '4' AS id  , '155' AS height  , '40' AS weight )  【修正前】 SELECT  id  , 'Height' AS category  , height AS item\_value  FROM  height\_weight  UNION ALL  SELECT  id  , 'Weight' AS category  , weight AS item\_value  FROM  height\_weight;   【修正後】 SELECT  id  , UNNEST (ARRAY ['Height', 'Weight']) AS category  , UNNEST (ARRAY [height, weight]) AS item\_value  FROM  height\_weight; |

（参考）縦→横変換にCASE文を用いた例

|  |
| --- |
| 【テストデータ】 WITH weather AS (   SELECT  '1' AS id  , '0001' AS tag  , '2024/01/01' AS text   UNION ALL   SELECT  '1' AS id  , '0002' AS tag  , '晴れ' AS text   UNION ALL   SELECT  '1' AS id  , '0003' AS tag  , '0%' AS text   UNION ALL   SELECT  '2' AS id  , '0001' AS tag  , '2024/02/01' AS text   UNION ALL   SELECT  '2' AS id  , '0002' AS tag  , '雨' AS text   UNION ALL   SELECT  '2' AS id  , '0003' AS tag  , '100%' AS text )  SELECT  id  , max(CASE WHEN tag = '0001' THEN text ELSE NULL END) AS check\_date  , max(CASE WHEN tag = '0002' THEN text ELSE NULL END) AS weather\_data  , max(CASE WHEN tag = '0003' THEN text ELSE NULL END) AS rainy\_percent  FROM  weather  GROUP BY  id; |