

内容

はじめに	1
0. SQL のコマンド／命令	2
0.1. SQL の形式	2
0.2. SELECT コマンド	3
1. テーブル結合と条件毎の集計 (JOIN、CASE)	5
2. 集計結果を更に計算 (副問い合わせ)	6
3. 集計結果の金額編集 (編集関数)	8
4. 問い合わせ中の行に関係づけた副問い合わせ (相関副問い合わせ)	9
5. 関連テーブルに存在しないキーを抽出する (相関副問い合わせ、EXISTS)	10
6. 作業テーブルと問い合わせ結果の再利用 (WITH)	11
7. 集約関数や配列を利用した 1 : N の関連付け	12
補足 性能を考慮したクエリ	14
補足 SELECT 命令の処理順	17

【改変履歴】

2023/11/1 「項番 7」、「補足 性能を考慮したクエリ」を追加

はじめに

SQL は RDBMS メーカーが独自に開発してきた RDB 操作のコマンドを ISO / IEC で標準化し拡張を続けて、2019 年には「ISO / IEC 9075-15 : 2019 パート 15」(SQL:2019) が公開されています。当初は ANSI (アメリカ規格協会) が標準化を進めたため、SQL の準拠水準として ANSI xx(xx は年)や日本語化した JIS xx と表記されている場合もあります。

SQL の内容は DDL (Data Definition Language)、DML (Data Manipulation Language)、DCL (Data Control Language) として整理され、標準化した内容は各メーカーが自社製品に取り込んでいます。但し、実際には過去製品のサポート事情もあってか完全互換にはなっておらず一部のデータ型や関数に差異があります。

RDB は SQL (あるいはロード・アンロードやコピー等の独自コマンド) を使わないとデータを更新したり照会することができません。逆に SQL を使えばアプリケーションプログラムを作らなくても複数テーブルの関係付けや加工・集計した高度な問い合わせが可能です。SQL は RDBMS に同梱されている UI から直接 実行するだけでなく ETL や BI 等のツール、アプリケーションに組み込んで利用することもできます。

SQL はテーブルの関係図から簡単に書け、開発、試験・検証から運用局面まで出番は多いのですが、問い合わせ条件を間違えると存在するはずのデータを取りこぼしたり逆に余分なデータを抽出してしまうことがあります。特に複数テーブルの結合等を行うと複雑で判りづらくなりますが、単純なクエリー(Query)を複数組み立てて 1 つの処理にすると分かり易く、基本的なパターンに慣れれば簡単に拡張ができます。

0. SQL のコマンド／命令

SQL は RDB の操作言語で、業務処理では主に INSERT、SELECT、UPDATE、DELETE というデータ操作言語 (DML) を使います。各々のコマンドは CRUD (データのライフサイクル-Create、Read、Uppdate、Delete) に対応しています。

0.1. SQL の形式

(1) DML の一般形式

INSERT INTO テーブル(列1, 列2 …) VALUES(値1, 値2 …)

INSERT INTO テーブル VALUES(値1, 値2 …) -- 列名を省略可 (全列、定義順で値指定)

[SELECT に関しては次項以降で説明します]

UPDATE テーブル SET 列x=値, 列y=値 WHERE 条件

DELETE FROM テーブル WHERE 条件

※値は文字型の場合は「'」アポストロフィで囲み、数字はアポストロフィ無で記述します

※各コマンドには他にも各種のオプション句があります

(2) コメント

行コメント -- 連続したハイフンを2つの後ろは行末迄コメントになります

ブロック /**/ 「/」と「*」で囲まれるとコメントになります

※古いバージョンだったり RDBMS によっては異なる場合もあります

(3) 命令の終端

セミコロン「;」が SQL の命令の終わりです。多くの場合、この後から新しい命令が書けます。

0.2. SELECT コマンド

SELECT は指定した条件に合うテーブルの内容を返してきます（他に日付・時刻等のシステム情報も取れますが割愛）。SELECT コマンドを他のコマンドの一部として記述し、得た内容を他のテーブルの結合（JOIN）相手としたり WHERE 条件の参照先として使うことができます。この他のコマンドに含まれる SELECT が「副問い合わせ」です。

更新系のコマンドは単一の実テーブルが対象になりますが、SELECT は複数のテーブルを結合して加工や編集を行ったり、更にはその結果を別の問い合わせの入力にすることができます。

次項より、少し高度な SELECT コマンドの事例を紹介します（内容は下表）。

#	機能	内容
1	テーブル結合と集計	結合(OUTER JOIN)して別テーブルの列を取込み(存在しない場合 null で)、データを集計(GROUP BY/SUM)して、条件(CASE)により設定する項目を変えます。
2	集計結果を更に計算	#1 と同様のデータ集計結果 (SELECT-副問い合わせで取得) と別のテーブルを結合(OUTER JOIN)し、更に集計(GROUP BY/SUM)する。
3	集計結果の金額編集	#2 の結果を人が見やすいように編集(TO_CHAR())して出力します。
4	問い合わせ中の行に関係付けた副問い合わせ	SELECT しているテーブルの列をキーに別テーブルを検索(相関副問い合わせ)し、別テーブルにキーが存在したら出力の対象にします。また、SELECT 句の中で別テーブルを集計(副問い合わせ)して出力項目の一つにします。
5	関連テーブルに存在しないキーを抽出する	#4 と逆に相関副問い合わせで存在しなかったら (NOT EXISTS)出力の対象にします。
6	作業テーブルと問い合わせ結果の再利用	副問い合わせの結果を保存し(WITH)、複数箇所から保存しておいた内容を参照します。
7	集約関数を利用した 1 : N の関連付け	1 : N の関係にある複数のテーブルを、N件側を1列に集約して結合します (LISTAGG/STRING_AGG 他)。 <div style="text-align: right;">【2023/11 追加】</div>

- 事例で使うテーブルの構成 以降の事例では、以下のテーブル構成を想定しています。



① 仕訳共通テーブル "public.t_siwake_kyotu"

列	タイプ	照合順序	Null 値を許容	デフォルト
trdate	character(8)		not null	
denpyono	character(5)		not null	
tantou	character(4)			
tekiyo	character varying(50)			
sysdate	character(8)		not null	

インデックス:

"pk_t_siwake_kyotu" PRIMARY KEY, btree (denpyono)

参照元:

TABLE "t_siwake_meisai" CONSTRAINT "fk_denpyono" FOREIGN KEY (denpyono) REFERENCES t_siwake_kyotu(denpyono)

② 仕訳明細テーブル "public.t_siwake_meisai"

列	タイプ	照合順序	Null 値を許容	デフォルト
denpyono	character(5)		not null	
taisyakukb	character(1)		not null	
gyo	numeric(1,0)		not null	
kamokucd	character(6)		not null	
kingaku	numeric(13,0)		not null	
aitekamokucd	character(6)		not null	

インデックス:

"pk_t_siwake_meisai" PRIMARY KEY, btree (denpyono, gyo, taisyakukb)

外部キー制約:

"fk_aite_kamokucd" FOREIGN KEY (aitekamokucd) REFERENCES m_kamoku(kamokucd)

"fk_denpyono" FOREIGN KEY (denpyono) REFERENCES t_siwake_kyotu(denpyono)

"fk_kamokucd" FOREIGN KEY (kamokucd) REFERENCES m_kamoku(kamokucd)

③ 科目マスタテーブル "public.m_kamoku"

列	タイプ	照合順序	Null 値を許容	デフォルト
kamokucd	character(6)		not null	
kamokunm	character varying(30)		not null	
dennyukb	character(1)		not null	
taisyakukb	character(1)		not null	
chohyokb	character(1)		not null	

インデックス:

"pk_m_kamoku" PRIMARY KEY, btree (kamokucd)

参照元:

TABLE "t_siwake_meisai" CONSTRAINT "fk_aite_kamokucd" FOREIGN KEY (aitekamokucd) REFERENCES m_kamoku(kamokucd)

TABLE "t_siwake_meisai" CONSTRAINT "fk_kamokucd" FOREIGN KEY (kamokucd) REFERENCES m_kamoku(kamokucd)

1. テーブル結合と条件毎の集計 (JOIN、CASE)

--仕訳明細から科目毎の借方計・貸方計を算出し、科目マスタから科目名を取得します

```
SELECT
    s.kamokucd
  , k.kamokunm
  , CASE WHEN s.taisyakub = '1' THEN sum(kingaku) ELSE 0 END as karikei --②
  , CASE WHEN s.taisyakub = '2' THEN sum(kingaku) ELSE 0 END as kasikei --③
  , s.taisyakub
FROM t_siwake_meisai as s -- ①
LEFT OUTER JOIN m_kamoku as k on s.kamokucd=k.kamokucd -- ②
GROUP BY s.kamokucd, k.kamokunm, s.taisyakub ORDER BY s.kamokucd -- ③
;
```

[全体の説明]

① t_siwake_meisai テーブルを入力にします。参照するときの名前(相関名)を”s”とします

“as”句は殆どのRDBMSで省略可能で、しばしば省略されて[FROM t_siwake_meisai s]になります

② t_siwake_meisai の科目コードと貸借区分、m_kamoku の科目名で纏め、科目コード順に並べて処理を行います

[この項のポイント]

① 左側に記述したテーブル(相関名”s”)に m_kamoku (相関名”k”)を付加)を kamokucd をキーに外部結合します。外部結合なので右側のテーブル(相関名”k”)にキーが存在しない左側テーブルの行も出力されます。※INNER join であればキーが存在した行だけが出力されます

② ②の纏め単位で、相関名”s”の貸借区分が '1' のとき金額を集計し、“as ~”で列名”karikei”とします

③ ②の纏め単位で、相関名”s”の貸借区分が '2' のとき金額を集計して”kasikei”の列名を付けます

[実行結果]

```
postgres=# select * from t_siwake_meisai;
denpyono | taisyakub | gyo | kamokucd | kingaku | aitekamokucd
-----+-----+-----+-----+-----+-----
00001    | 1         | 0   | 110101   | 1000000 | 000000
00001    | 1         | 1   | 110102   | 10000000 | 000000
00001    | 2         | 0   | 120100   | 1000000 | 000000
00001    | 2         | 1   | 120200   | 10000000 | 000000
00002    | 1         | 0   | 110101   | 2000000 | 000000
00002    | 1         | 1   | 110102   | 20000000 | 000000
00002    | 2         | 0   | 120100   | 2000000 | 000000
00002    | 2         | 1   | 120200   | 20000000 | 000000
```

(8 行)

```
postgres=# SELECT
postgres=#   s.kamokucd
postgres=# , k.kamokunm
postgres=# , CASE WHEN s.taisyakub = '1' THEN sum(kingaku) ELSE 0 END as karikei --②
postgres=# , CASE WHEN s.taisyakub = '2' THEN sum(kingaku) ELSE 0 END as kasikei --③
postgres=# , s.taisyakub
postgres=# FROM t_siwake_meisai as s -- ①
postgres=# LEFT OUTER JOIN m_kamoku as k on s.kamokucd=k.kamokucd -- ②
postgres=# GROUP BY s.kamokucd, k.kamokunm, s.taisyakub ORDER BY s.kamokucd -- ③
postgres=# ;
```

```

kamokucd | kamokunm | karikei | kasikei | taisyakub
-----+-----+-----+-----+-----
110101   | 現金     | 3000000 | 0        | 1
110102   | 預金     | 30000000 | 0        | 1
120100   | 商品     | 0        | 3000000 | 2
120200   | 製品     | 0        | 30000000 | 2
```

(4 行)

2. 集計結果を更に計算（副問い合わせ）

--仕訳明細から科目毎に借方計と貸方計を算出し、次に貸借の差（残高）を求めます

```
SELECT
    sub.kamokucd, k.kamokunm
    , sum(sub.karikei) as 借計
    , sum(sub.kasikei) as 貸計
    , CASE WHEN k.taisyakub='1' THEN sum(sub.karikei)-sum(sub.kasikei) --②
      ELSE sum(sub.kasikei)-sum(sub.karikei) END as 貸借計
FROM (
    SELECT s.kamokucd as kamokucd
        , CASE WHEN s.taisyakub = '1' THEN sum(kingaku) ELSE 0 END as karikei
        , CASE WHEN s.taisyakub = '2' THEN sum(kingaku) ELSE 0 END as kasikei
        , s.taisyakub
    FROM t_siwake_meisai s
    GROUP BY s.kamokucd, s.taisyakub
    ORDER BY s.kamokucd
) as sub
--①

LEFT OUTER JOIN m_kamoku k on sub.kamokucd=k.kamokucd

GROUP BY sub.kamokucd, k.kamokunm, k.taisyakub
ORDER BY sub.kamokucd
;
```

[ポイント]

- ① FROM の中に副問い合わせ（SELECT）を入れて結果に”sub”という相関名を付ける
副問い合わせの中では仕訳テーブルの借方／貸方(taisyakub)を参照して、借方発生か貸方発生かを判定しています
 - ② 外側の SELECT では副問い合わせで作成した「科目」×「仕訳の貸・借」毎の合計に対して、科目マスタに設定されている貸借区分（借方に残高が残る科目は '1'）をみて科目毎の残高を計算します
- ※外側の SELECT の結果を副問い合わせにして更に加工したり、別のテーブルと結合することも可能です

[実行結果]

```
postgres=# SELECT
postgres=#     sub.kamokucd, k.kamokunm
postgres=#     , sum(sub.karikei) as 借計
postgres=#     , sum(sub.kasikei) as 貸計
postgres=#     , CASE WHEN k.taisyakub='1' THEN sum(sub.karikei)-sum(sub.kasikei) --②
postgres=#     ELSE sum(sub.kasikei)-sum(sub.karikei) END as 貸借計
postgres=# FROM (
postgres=#     SELECT s.kamokucd as kamokucd
postgres=#     , CASE WHEN s.taisyakub = '1' THEN sum(kingaku) ELSE 0 END as karikei
postgres=#     , CASE WHEN s.taisyakub = '2' THEN sum(kingaku) ELSE 0 END as kasikei
postgres=#     , s.taisyakub
postgres=# FROM t_siwake_meisai s
postgres=# GROUP BY s.kamokucd, s.taisyakub
postgres=# ORDER BY s.kamokucd
postgres=# ) as sub
postgres=# LEFT OUTER JOIN m_kamoku k on sub.kamokucd=k.kamokucd
postgres=# GROUP BY sub.kamokucd, k.kamokunm, k.taisyakub
postgres=# ORDER BY sub.kamokucd
postgres=# ;
```

kamokucd	kamokunm	借計	貸計	貸借計
110101	現金	3000000	0	3000000
110102	預金	30000000	0	30000000
120100	商品	500000000	3000000	497000000
120200	製品	0	30000000	-30000000

(4 行)

```
postgres=# select * from t_siwake_meisai;
denpyono | taisyakub | gyo | kamokucd | kingaku | aitekamokucd
-----+-----+-----+-----+-----+-----
00001    | 1         | 0   | 110101   | 1000000 | 000000
00001    | 1         | 1   | 110102   | 10000000 | 000000
00001    | 2         | 0   | 120100   | 1000000 | 000000
00001    | 2         | 1   | 120200   | 10000000 | 000000
00002    | 1         | 0   | 110101   | 2000000 | 000000
00002    | 1         | 1   | 110102   | 20000000 | 000000
00002    | 2         | 0   | 120100   | 2000000 | 000000
00002    | 2         | 1   | 120200   | 20000000 | 000000
00001    | 1         | 2   | 120100   | 500000000 | 000000
```

(9 行)

```
postgres=# select * from m_kamoku where kamokucd in ('110101','110102','120100','120200');
kamokucd | kamokunm | dennyakb | taisyakub | chohyokb
-----+-----+-----+-----+-----
110101   | 現金     | 1         | 1         | 1
110102   | 預金     | 1         | 1         | 1
120100   | 商品     | 1         | 1         | 1
120200   | 製品     | 0         | 1         | 1
```

(4 行)

3. 集計結果の金額編集（編集関数）

-- 数値編集関数で CASE 式を囲み、貸借計の計算結果を 3 桁単位に編集

```
SELECT
    sub.kamokucd, k.kamokunm
    , sum(sub.karikei) 借計
    , sum(sub.kasikei) 貸計
    , to_char(
        CASE WHEN k.taisyakub='1' THEN sum(sub.karikei)-sum(sub.kasikei)
        ELSE sum(sub.kasikei)-sum(sub.karikei) END
        , '999,999,999,999,999') 貸借計
FROM (
    SELECT s.kamokucd as kamokucd
        , CASE WHEN s.taisyakub = '1' THEN sum(kingaku) ELSE 0 END as karikei
        , CASE WHEN s.taisyakub = '2' THEN sum(kingaku) ELSE 0 END as kasikei
        , s.taisyakub
    FROM t_siwake_meisai s
    GROUP BY s.kamokucd, s.taisyakub
    ORDER BY s.kamokucd
) as sub

LEFT OUTER JOIN m_kamoku k on sub.kamokucd=k.kamokucd

GROUP BY sub.kamokucd, k.kamokunm, k.taisyakub
ORDER BY sub.kamokucd
;
```

[ポイント]

① CASE 式は値に置換されて関数に渡されます。

[実行結果]

kamokucd	kamokunm	借計	貸計	貸借計
110101	現金	3000000	0	3,000,000
110102	預金	30000000	0	30,000,000
120100	商品	500000000	3000000	497,000,000
120200	製品	0	30000000	-30,000,000

(4 行)

4. 問い合わせ中の行に関係づけた副問い合わせ（相関副問い合わせ）

--主テーブルの SELECT ループ内で 1 件毎、一致(相関)する副テーブルを検索する

```

SELECT
  k.kamokucd, k.kamokunm
  , (SELECT sum(kingaku) FROM t_siwake_meisai
      WHERE kamokucd = k.kamokucd      --④
      AND taisyakukb = '1'
    ) 借計
  , (SELECT sum(kingaku) FROM t_siwake_meisai
      WHERE kamokucd = k.kamokucd      --⑤
      AND taisyakukb = '2'
    ) 貸計
FROM m_kamoku k                        --①
WHERE (
  k.kamokucd = (
    SELECT kamokucd FROM t_siwake_meisai s
    WHERE k.kamokucd = s.kamokucd      --③
    LIMIT 1
  )
)
ORDER BY k.kamokucd
;
```

[ポイント]

- ① 主テーブル（一番外側の SELECT）を科目マスタとし、相関名を”k”とします
- ② 主テーブルからの抽出条件として仕訳明細への登録を確認します
- ③ 仕訳明細の科目コードと科目マスタの科目コードが一致することを条件にします
 ※このとき、相関名”k”を使うことで主テーブルのループ中の科目コードが比較対象になります
 ∴ 「相関副問い合わせ」と呼ぶ
- ④ 相関名”k”の科目コードと一致し、貸借区分='1'（借）の仕訳明細の金額を集計します
- ⑤ ----- ” ----- 、貸借区分='2'（貸） ----- ” -----

[実行結果]

kamokucd	kamokunm	借計	貸計
110101	現金	3000000	
110102	預金	30000000	
120100	商品	500000000	3000000
120200	製品		30000000

(4 行)

※この例は「テーブル結合」と同一の結果が得られますが、科目マスタの全件+科目コード×仕訳明細全件のアクセスが発生するため非常に効率が悪くなります

5. 関連テーブルに存在しないキーを抽出する (相関副問い合わせ、EXISTS)

--仕訳明細の登録がない勘定科目を抽出する

```
SELECT * FROM m_kamoku k
WHERE NOT EXISTS (
    SELECT kamokucd FROM t_siwake_meisai s
    WHERE s.kamokucd = k.kamokucd
)
;
```

[ポイント]

- ① 副問い合わせの結果が存在しなければ (空集合) 外側の WHERE 条件が真になります
- ② 副問い合わせ抽出条件は、外側でループ中の科目マスタ (相関名"K") の科目コードと副問い合わせで参照している仕訳明細の科目コードが一致することです

※この例は、相関副問い合わせでなければならない事例です

```
postgres=# SELECT * FROM m_kamoku k
postgres=# WHERE NOT EXISTS (
postgres(#     SELECT kamokucd FROM t_siwake_meisai s
postgres(#     WHERE s.kamokucd = k.kamokucd
postgres(# )
postgres=# ;
```

kamokucd	kamokunm	dennyukb	taisyakub	chohyokb
000000	諸口	0	3	0
100000	流動資産	0	1	1
110000	当座資産	0	1	1
110100	現金及び預金	0	1	1
110103	普通預金	1	1	1
110104	定期預金	1	1	1
110105	他預金	1	1	1
110200	受取手形	0	1	1
110201	受取手形 (関係会社のものを除く)	1	1	1
110207	子会社受取手形	1	1	1
110208	その他の関係会社受取手形	1	1	1
110209	貸倒引当金 (受取手形)	1	2	1
110300	売掛金	0	1	1
110301	売掛金 (関係会社のものを除く)	1	1	1
110307	子会社売掛金	1	1	1
110308	その他の関係会社売掛金	1	1	1
110309	貸倒引当金 (売掛金)	1	2	1
110400	有価証券	1	1	1
110500	自己株式	1	1	1
110600	親会社株式	1	1	1
120000	たな卸資産	0	1	1
120300	副産物及び作業くず	0	1	1
120400	半製品	0	1	1
120500	原材料	0	1	1

(以下 略)

※仕訳明細にする科目コード ('110101', '110102', '120100', '120200') が出力対象外になっている

6. 作業テーブルと問い合わせ結果の再利用 (WITH)

-- ①「副問い合わせ」したい内容(科目、集計金額)を WITH 句で作業テーブルとして生成
 WITH **sum_kamoku** AS(

```
SELECT k.kamokucd as kamokucd, kamokunm
      , CASE WHEN s.taisyakub = '1' THEN sum(kingaku) ELSE 0 END as kari
      , CASE WHEN s.taisyakub = '2' THEN sum(kingaku) ELSE 0 END as kasi
      , k.taisyakub
FROM m_kamoku k LEFT OUTER JOIN t_siwake_meisai s ON k.kamokucd = s.kamokucd
GROUP BY k.kamokucd, kamokunm, s.taisyakub
```

)
 -- ②生成した作業テーブルを多重に参照して科目コードの中分類に集計する

```
SELECT *
FROM(
  SELECT kamokucd
        , kamokunm
        , (
          SELECT sum(kari) FROM sum_kamoku
          WHERE SUBSTR(kamokucd, 1, 2) = SUBSTR(L2.kamokucd, 1, 2)
        ) kari
        , (
          SELECT sum(kasi) FROM sum_kamoku
          WHERE SUBSTR(kamokucd, 1, 2) = SUBSTR(L2.kamokucd, 1, 2)
        ) kasi
        , taisyakub
FROM sum_kamoku L2
WHERE L2.kamokucd LIKE ('%0000')
AND L2.kamokucd NOT LIKE ('%00000')
) L3
ORDER BY kamokucd
limit 5;
```

[ポイント]

- ① 仕訳明細の金額を科目コード(細分類)単位の貸借に集計し、作業表を作成します
 - ② ①の作業表から科目の上2桁が同一の科目を集計し、xx0000の科目コードを抽出します
- ※この処理は副問い合わせでもできますが、WITH句を使うことで記述が簡単になります

[実行結果]

kamokucd	kamokunm	kari	kasi	taisyakub
110000	当座資産	33000000	0	1
120000	たな卸資産	50000000	33000000	1
140000	その他の流動資産	0	0	1
210000	有形固定資産	0	0	1
230000	無形固定資産	0	0	1

(5行)

7. 集約関数や配列を利用した 1 : N の関連付け

集約関数はグルーピングした複数行から集約した一つの値を求める関数で SUM、COUNT、MIN、MAX 等の数値計算がよく使われますが、文字列を一つの値（列）に集約する関数も SQL:2016 で「LISTAGG: 行のグループの値を区切り文字で区切られた文字列に変換する関数」として標準化されています。これも製品の実装が先行したため RDBMS によって関数名が異なっている場合があります、PostgreSQL の場合は STRING_AGG 関数がこれにあたります。

(1) STRING_AGG の例

下図は仕訳共通に関係する仕訳明細を string_agg(借)と string_agg(貸)のセルに集約しています

伝票 (String_agg)						
trdate	denpyono	tantou	tekiyo	sysdate	string_agg (借)	string_agg (貸)
20230601	00001	1000	摘要	20230811	1,0,110101,1000000 1,1,110102,10000000 1,2,120100,500000000	2,0,120100,1000000 2,1,120200,10000000
20230602	00002	2000	摘要	20230811	1,0,110101,2000000 1,1,110102,20000000	2,0,120100,2000000 2,1,120200,20000000
20230801	00003	1000	損益計算書,出力用	20230802	1,0,810109,10000000 1,1,820100,1000000000 1,2,820201,10000000000 1,3,820300,100000000	2,0,810101,10000000000 2,1,810200,1000000000 2,2,820209,100000000 2,3,820400,1000000000

(3 行)

<上記出力のためのクエリ>

```
select k.*
, (select STRING_AGG(
    taisyakukb||','||gyo||','||kamokucd||','||kingaku,chr(10)
    order by taisyakukb, gyo
  )
  from t_siwake_meisai m
  where m.denpyono=k.denpyono
    and taisyakukb='1'
  ) as STRING_AGG (借)
, (select STRING_AGG(
    taisyakukb||','||gyo||','||kamokucd||','||kingaku,chr(10)
    order by taisyakukb, gyo
  )
  from t_siwake_meisai m
  where m.denpyono=k.denpyono
    and taisyakukb='2'
  ) as STRING_AGG (貸)
from t_siwake_kyotu k
;
```

この例では区切り文字に改行コード chr(10) [html では
 になります] を使って複数行に見せていますがセル内は一つの値です。また STRING_AGG 内の「||」は Oracle と互換の文字列連結演算子で CONCAT(taisyakukb, ',', gyo, ',', ... kingaku)でも同じ結果が得られます

(2) その他の集約関数と配列

AISI/ISO SQL には定義されていませんが、PosgreSQL やクラウドデータベース等の配列が使える RDBMS は配列化する集約関数 Array_agg や、配列を生成する Array コンストラクタが用意されています。

```
-- Array_agg
¥C 伝票 (Array_agg)
select k.*
      , ARRAY_AGG(
          taisyakukb||','||gyo||','||kamokucd||','||kingaku
          order by taisyakukb, gyo
        )
from t_siwake_kyotu k join t_siwake_meisai m
     on m.denpyono=k.denpyono
group by trdate,k.denpyono,tantou,tekiyo,sysdate
;

-- array
¥C 伝票 (Array)
select k.*
      , ARRAY(
          select taisyakukb||','||gyo||','||kamokucd||','||kingaku
          from t_siwake_meisai m
          where m.denpyono=k.denpyono
        )
from t_siwake_kyotu k
;

```

上記のクエリを html 形式で出力すると以下の出力が得られます。見た目の結果は同じですが Arrat_agg は group by で集約し、Array はクエリの結果をそのまま配列化します。

伝票 (Array_agg)

trdate	denpyono	tantou	tekiyo	sysdate	array_agg
20230601	00001	1000	摘要	20230811	{"1,0,110101,1000000","1,1,110102,10000000","1,2,120100,500000000","2,0,120100,1000000","2,1,120200,10000000"}
20230602	00002	2000	摘要	20230811	{"1,0,110101,2000000","1,1,110102,20000000","2,0,120100,2000000","2,1,120200,20000000"}
20230801	00003	1000	損益計算書,出力用	20230802	{"1,0,810109,100000000","1,1,820100,10000000000","1,2,820201,10000000000","1,3,820300,100000000","2,0,810101,10000000000","2,1,810200,1000000000","2,2,820209,100000000","2,3,820400,1000000000"}

(3行)

伝票 (Array)

trdate	denpyono	tantou	tekiyo	sysdate	array
20230601	00001	1000	摘要	20230811	{"1,0,110101,1000000","1,1,110102,10000000","2,0,120100,1000000","2,1,120200,10000000","1,2,120100,500000000"}
20230602	00002	2000	摘要	20230811	{"1,0,110101,2000000","1,1,110102,20000000","2,0,120100,2000000","2,1,120200,20000000"}
20230801	00003	1000	損益計算書,出力用	20230802	{"2,0,810101,10000000000","1,0,810109,10000000","2,1,810200,10000000000","1,1,820100,100000000","1,2,820201,10000000000","2,2,820209,100000000","1,3,820300,100000000","2,3,820400,1000000000"}

(3行)

補足 性能を考慮したクエリ

SQLは同一の結果を得るために複数の書き方ができますが、書き方は性能に影響します。例えば、「集約関数を利用した1:Nの関連付け」のクエリは以下のように書くことができます。

(1) SQLのパターン

① joinを使わず、select句の中に単独の項目を取得するselect命令を書く

```
select k.*
, (select STRING_AGG(
    taisyakukb||','||gyo||','||kamokucd||','||kingaku, chr(10)
    order by taisyakukb, gyo
)
from t_siwake_meisai m
where m.denpyono=k.denpyono
and taisyakukb='1'
) as STRING_AGG (借)
, (select STRING_AGG(
    taisyakukb||','||gyo||','||kamokucd||','||kingaku, chr(10)
    order by taisyakukb, gyo
)
from t_siwake_meisai m
where m.denpyono=k.denpyono
and taisyakukb='2'
) as STRING_AGG (貸)
from t_siwake_kyotu k
;
```

見た目が分かり易い反面、主テーブル（仕訳共通）1件に対して関連テーブル（仕訳明細）を2回（貸方・借方）全件SELECTしてしまう（共通全件×明細全件×2）可能性があります。

② joinをした結果（借方）にjoin（貸方）する

```
select k2.*
, STRING_AGG(
    m2.taisyakukb||','||m2.gyo||','||m2.kamokucd||','||m2.kingaku, chr(10)
    order by m2.taisyakukb, m2.gyo
) as STRING_AGG (貸)
from(
select k.*
, STRING_AGG(
    taisyakukb||','||gyo||','||kamokucd||','||kingaku, chr(10)
    order by taisyakukb, gyo
) as STRING_AGG (借)
from t_siwake_kyotu k LEFT OUTER JOIN t_siwake_meisai m
on m.denpyono=k.denpyono
where taisyakukb='1'
group by trdate, k.denpyono, tantou, tekiyo, sysdate
) as k2 LEFT OUTER JOIN t_siwake_meisai m2
on m2.denpyono=k2.denpyono
where m2.taisyakukb='2'
group by k2.trdate, k2.denpyono, k2.tantou, k2.tekiyo, k2.sysdate, k2.STRING_AGG (借)
;
```

仕訳共通に対して最初に借方の仕訳明細を結合しその結果に貸方の仕訳明細を結合する2段階にしています。データを結合してから加工することでIOを減らせる可能性があります。

③ 借方と貸方の両方を集約後に join

```
select k.*
  , STRING_AGG (借)
  , STRING_AGG (貸)
from t_siwake_kyotu k
  LEFT OUTER JOIN (
  select
    m1.denpyono
  , STRING_AGG(
    m1.taisyakukb||','||m1.gyo||','||m1.kamokucd||','||m1.kingaku,chr(10)
    order by m1.taisyakukb, m1.gyo
  ) as STRING_AGG (借)
  from t_siwake_meisai m1
  where taisyakukb='1'
  group by m1.denpyono
) as m1 on m1.denpyono = k.denpyono
  LEFT OUTER JOIN (
  select
    m2.denpyono
  , STRING_AGG(
    m2.taisyakukb||','||m2.gyo||','||m2.kamokucd||','||m2.kingaku,chr(10)
    order by m2.taisyakukb, m2.gyo
  ) as STRING_AGG (貸)
  from t_siwake_meisai m2
  where taisyakukb='2'
  group by m2.denpyono
) as m2 on m2.denpyono = k.denpyono
group by trdate, k.denpyono, tantou, tekiyo, sysdate, STRING_AGG (借) , STRING_AGG (貸)
order by k.denpyono
;
```

集約した借方の仕訳明細と集約した貸方の仕訳明細を結合しています。この例もデータを結合してから加工することで IO を減らせる可能性があります。

※上記の SQL 例の性能に関して全て「可能性」なのは RDBMS が統計情報を用いて最適化を行いアクセス経路を決定するためです。決定した経路はコマンドで確認することができます。

(2) 性能測定 (EXPLAIN コマンド)

EXPLAIN コマンドは ANSI/ISO で標準化されたものではなくオプションや使い方が異なる場合がありますが、大方の RDBMS が備えている機能です。このコマンドでクエリを実行する際のアクセスプラン (実行計画) やインデックスの使用有無を確認することができます。

PostgreSQL の場合は、「ANALYZE オプションを付けると EXPLAIN は実際にその問い合わせを実行し、計画ノードごとに実際の行数と要した実際の実行時間を表示します。」

(PostgreSQL ドキュメント要約¹)

¹ PostgreSQL 15.4 EXPLAIN の利用 <https://www.postgresql.jp/document/15/html/using-explain.html>

前項の各クエリの前に EXPLAIN (ANALYZE オプション付き) を付加して実行すると、以下のような内容が表示されます。

① select 句の中に取得項目の select 命令を書いた場合の EXPLAIN ANALYZE

```

postgres=# EXPLAIN ANALYZE
postgres=# select k.*
postgres=# , (select STRING_AGG(
postgres=#     taisyakukb||','||gyo||','||kamokucd||','||kingaku,chr(10)
postgres=#     order by taisyakukb, gyo
postgres=# )
postgres=#     from t_siwake_meisai m
postgres=#     where m.denpyono=k.denpyono
postgres=#     and taisyakukb='1'
postgres=# ) as STRING_AGG (借)
postgres=# , (select STRING_AGG(
postgres=#     taisyakukb||','||gyo||','||kamokucd||','||kingaku,chr(10)
postgres=#     order by taisyakukb, gyo
postgres=# )
postgres=#     from t_siwake_meisai m
postgres=#     where m.denpyono=k.denpyono
postgres=#     and taisyakukb='2'
postgres=# ) as STRING_AGG (貸)
postgres=# from t_siwake_kyotu k
postgres=# ;

                                QUERY PLAN
-----
Seq Scan on t_siwake_kyotu k  (cost=0.00..5975.78 rows=310 width=298) (actual time=0.142..0.246 rows=3 loops=1)
  SubPlan 1
    -> Aggregate  (cost=9.61..9.62 rows=1 width=32) (actual time=0.046..0.046 rows=1 loops=3)
          -> Bitmap Heap Scan on t_siwake_meisai m  (cost=4.19..9.54 rows=2 width=18) (actual time=0.018..0.019 rows=3 loops=3)
                Recheck Cond: ((denpyono = k.denpyono) AND (taisyakukb = '1'::bpchar))
                Heap Blocks: exact=3
                -> Bitmap Index Scan on pk_t_siwake_meisai  (cost=0.00..4.19 rows=2 width=0) (actual time=0.014..0.014 rows=3 loops=3)
                      Index Cond: ((denpyono = k.denpyono) AND (taisyakukb = '1'::bpchar))
  SubPlan 2
    -> Aggregate  (cost=9.61..9.62 rows=1 width=32) (actual time=0.025..0.026 rows=1 loops=3)
          -> Bitmap Heap Scan on t_siwake_meisai m_1  (cost=4.19..9.54 rows=2 width=18) (actual time=0.008..0.008 rows=3 loops=3)
                Recheck Cond: ((denpyono = k.denpyono) AND (taisyakukb = '2'::bpchar))
                Heap Blocks: exact=3
                -> Bitmap Index Scan on pk_t_siwake_meisai  (cost=0.00..4.19 rows=2 width=0) (actual time=0.006..0.006 rows=3 loops=3)
                      Index Cond: ((denpyono = k.denpyono) AND (taisyakukb = '2'::bpchar))
Planning Time: 0.578 ms
Execution Time: 0.371 ms
(17 行)

```

※最終行に計画生成と最適化に掛かった時間と、実行時間を表示 (以下、各 2 回の試行結果纏め)

1 回目	2 回目
Planning Time: 0.578 ms	0.476 ms
Execution Time: 0.371 ms	0.455 ms

② 集約した結果 (借方) に join (貸方) する

Planning Time: 0.741 ms	0.584 ms
Execution Time: 0.330 ms	0.396 ms

③ 借方と貸方の両方を集約後に join

Planning Time: 0.530 ms	0.566 ms
Execution Time: 0.346 ms	0.327 ms

- ・所要時間はデータ量やインデックスの状態等で大きく変動し、実行の都度変わります。
- ・ANALYZE オプションを付けると実際に処理が行われる (付けなければ推定値を表示) ので更新処理に対して実行するときは注意してください。

補足 SELECT 命令の処理順

SELECT 命令を構成する各句の処理順²は以下になります。

1. FROM (ON JOIN) (結合する条件と対象テーブルを決め) 基データを決定します
2. WHERE 基データを条件で選択します
3. GROUP BY 基データを纏めます
4. HAVING 纏めた基データから条件で選択します
5. SELECT 出力するデータの形式に成形します
6. ORDER BY 出力するデータを並べ替えます
7. LIMIT 出力データをカウントし、指定件数で打ち切ります

(1) 一般的な注意点

処理順から各句で取り扱う対象が変わります。例えば、SELECT 句が処理の対象にするのは GROUP BY、HAVING の後なので集約前のデータは CASE 式等で扱えません。また、読み込みの順は ORDER BY の順でデータではありません。

但し、処理順とは関係なく文解釈が前もって行われるので、SELECT の中で付けた項目につけた as (別名) が GROUP BY 句等で使えます (RDBMS による)。

(2) 性能面・共有資源利用上の注意点

FROM で基になる行が展開され扱うデータの総量が決まり、その中から WHERE 句の条件に該当する全ての行が排他的対象になる (並走する更新系業務と衝突) 可能性があります。

また、性能面で改善が必要になった場合は、WHERE 条件を以下の観点で見直してください。

- ① インデックスになっている項目を使う。後方一致の LIKE 句を使わない
- ② 最初にデータ数の絞り込み効果の高い項目を使う
- ③ 大量のトランザクションデータ同士のテーブル結合は行わない

(副問い合わせで集約後に結合を行う)

※RDBMS は SQL 実行時に統計情報で最適化し、実行の順番を変更したりインデックスを使わない場合があります。添付のツール等で統計情報やアクセスプランを確認してください。

² 内部処理の順は RDBMS の製品/バージョンで異なる (アクセスプランの最適化等) 可能性がありますが、処理結果はここで記述した内容と論理的に一致するはずですが

<https://docs.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql?view=sql-server-ver15#logical-processing-order-of-the-select-statement>

<https://oracle.readthedocs.io/en/latest/sql/basics/query-processing-order.html>

<https://www.ibm.com/support/pages/logical-processing-phases-sql-select-statements>