セクション 14-ハンズオンスクリプト

■FROM 句におけるサブクエリ

先ほどのコマンドを入力して、実行してみましょう。

()内に SELECT 文を入れます。

```
SELECT
    name
FROM
    (SELECT
        *
    FROM
        users) AS Sub;
```

と入力して、実行します。すると users テーブルからアスタリスクで全てのデータを抽出した上で、さらにその抽出結果から name を抽出することができます。

これは

```
SELECTname FROM users ;
```

と同じ結果となります。

うしろに AS 句を利用して、列名をつけていますが、特に抽出されたリストには利用されていないため、あまり意味がないように思えます。そこで AS Sub を削除して実行してみます。

```
SELECT
    name
FROM
    (SELECT
        *
    FROM
        users);
```

で実行すると、エラーとなってしまい、エラー文には列名が必要ですとのメッセージが記載されています。つまり、FROM 句におけるサブクエリとしては AS による名称付けが必須となっています。その理由は、データ抽出結果を一旦保存するために AS によってデータ抽出結果名称を一時的に設定することが必要となるためです。

再び AS 句を追加した上で、（）内の SELECT 文を GROUP BY によってグループ化してみます。

```
SELECT
    gender
FROM
    (SELECT
        *
    FROM
        users
  GROUP BY gender) AS Sub;
```

と入力して、Gender でグループ化してみます。グループ化した内容をさらに SELECT 文でデータ抽出すてくれる文となりました。

今度はサブクエリ内で COUNT を利用してみます。

```
SELECT
    *
FROM
    (SELECT
        product_id, sum(amount) AS amount_sum
    FROM
        order_details
    GROUP BY product_id) AS sub;
```

と入力して、実行すると、product_id ごとに集約化した注文総量をデータ抽出するサブクエリ

が実行されました。

ただし、これもサブクエリ内のクエリと結果は同じとなります。

```
SELECT
        product_id, sum(amount) AS amount_sum
    FROM
        order_details
    GROUP BY product_id;
```

したがって、このように抜き出して実行しても同じとなります。

FROM 句におけるサブクエリは、VIEW を作成して、VIEW 内のデータをさらにデータ抽出して

確認することと構造は同じです。

先ほどのクエリで VIEW を作成してみます。

```
CREATE VIEW sub AS
    SELECT
        product_id, sum(amount) AS amount_sum
    FROM
        order_details
    GROUP BY product_id;
```

と入力して、VIEW を作成して、SELECT 文で作成した VIEW からデータを抽出します。

```
SELECT * FROM sub;
```

を実行すると、先ほどのサブクエリ文と同じ結果がかえってきました。

つなり、サブクエリを作成して、サブクエリ内からデータを指定して抽出する SELECT 文が VIEW からデータを抽出する SELECT 文と同じ役割になっているわけです。

FROM 句によって新しいデータ抽出が可能になるわけではありませんが、一旦 SELECT したデータを、さらに絞り込んでデータ抽出したい場合などに利用することができます。

今度はサブクエリを選択した SELECT 文に対して条件を付けていきます。その際にカッコ内では GROUP　BY を利用しているため、その後ろの条件としては HAVING を利用することになるように思われるかもしれませんが、WHERE 句になります。なぜなら、（）内のサブクエリはカッコ外のメインのクエリにとっては単なる TABLE 名の指定と同じでしかないためです。

```
SELECT
    *
FROM
    (SELECT
        product_id, COUNT(amount) AS id_count
    FROM
        order_details
    GROUP BY product_id) AS sub
WHERE product_id = 1;
```

と入力して、実行します。このように条件付けを実行したデータ抽出に成功しました。

WHERE 句によって条件付けを実施することができますが、これは通常の SELECT 文の条件付けと同じように設定すること同じことです。したがって、他のデータ抽出条件も同様に利用することが可能です。

■SELECT 句におけるサブクエリ

確認していきます。

```
SELECT
        (SELECT SUM(amount) FROM order_details) AS SUM
FROM
    order_details;
```

発注数量の合計を order_details テーブルから絞り込んだ上で、出力項目を AS 以降の

SUM として表示する内容になります。

実行すると、総数を合計して、全ての列にいれてくれています。これだけだとあまり意味のある

クエリ結果ではありませんが、次のように利用してみると使い方がわかります。

```
SELECT product_id,SUM(amount) AS PRODUCT_SUM,
        (SELECT SUM(amount) FROM order_details) AS TOTAL_SUM
FROM
    order_details
GROUP BY product_id;
```

というように product_id;によってグループ化をして、製品ごとの総発注数と全体の発注数をな

らべて抽出するクエリ結果がでてきます。

GROUP BY の影響はサブクエリの（）内には及ぼさないため、product_id;によってグループ化した SUM の結果と、全体の SUM の結果とを同時に抽出して並べることが可能となります。

このように異なる集約結果を同時に抽出する場合は、GROUP BY の影響範囲をわけるために SELECT 句内のサブスクリプションを利用することができます。

次にサブクエリのカッコ内を GROUP BY によってグループ化してみます。

```
SELECT
        (SELECT SUM(amount) FROM order_details GROUP BY product_id) AS
SUM
FROM
    order_details;
```

と入力して実行すると、これは失敗してしまいます。サブクエリ側に GROUP 化の指定は制限があって利用が上手くいかないようです。

SELECT 句におけるサブクエリの利用は一見意味がないように思えますが、GROUP 化などの影響範囲を分けることができるため、少し特徴のあるリストを作る際に利用することがあります。

■WHERE 句におけるサブクエリ（EXISTS の活用）

```
SELECT
    ＊
FROM
    users
WHERE EXISTS
    (SELECT ＊ FROM users WHERE id =1);
```

このように入力して、実行します。実行結果ですが、users テーブルに入っているデータがすべて抽出されています。

次に

```
SELECT
    ＊
FROM
    users
WHERE EXISTS
    (SELECT ＊ FROM users WHERE id =101);
```

と入力して、実行してみましょう。このように何も取得されませんでした。

これは IN を利用した場合とだいぶ異なります。EXISTS を利用した場合は、サブクエリ内の条件に合致したデータがテーブル内の存在した場合に、そのテーブル内のデータを全て抽出することになります。IN の場合は指定された条件のみのデータが抽出されました。

このコマンドを IN 句にしてみましょう。

```
SELECT
    *
FROM
    users
WHERE id IN (SELECT id FROM users WHERE id =1);
```

と入力して、実行します。このように id が 1 のデータのみが抽出されました。つまり、EXIST 句はそのデータが存在しているか、していないかを確認して、存在している場合は全てのデータを抽出するのが EXISTS になります。一方で IN 句は指定した条件の範囲内のみです。

この EXISTS 句は異なるテーブル間で実施した場合に、テーブルを条件で結合した場合としていない場合で結果がことなります。それを確認していきましょう。

```
SELECT
    *
FROM
    users
WHERE EXISTS
    (SELECT * FROM orders WHERE orders.user_id = users.id);
```

と入力して、orders.user_id = users.id で id が一致した場合を条件にすることで、テーブル間を結合した条件をサブクエリ内に設定します。

この場合は、互いのテーブルのデータが合致したデータを抽出するようになります。このようにデータの有無に応じてデータを抽出するのが EXISTS 句によるサブクエリとなります。

■HAVING 句におけるサブクエリ

先ほどのコマンドを入力していきます。

```
SELECT
        product_id, SUM(amount) AS amount_sum
FROM
        order_details
GROUP BY product_id
    HAVING SUM(amount) > (SELECT SUM(amount)/9 FROM order_details);
```

と入力して、実行します。product_id ごとにグループ化して全体の amount の平均よりも多い、製品ごとの amount を抽出しています。平均値は product_id が九つあるので、9 で割っている式となっています。

サブクエリを単独で実施してみましょう。

```
SELECT SUM(amount)/9 FROM order_details;
```

と入力して、実行します。このようにトータルの amount の平均値がでてきます。これを HAVING 句内で条件に指定しています。

このように HAVING 句でサブクエリを利用することで、クエリ結果を比較に用いることができ、より複雑なクエリを作成することができるようになります。

■ANY/SOME/ALL による論理式

まずは２つの条件式を用いた複雑な条件文をつくってみましょう。

```
SELECT
    *
FROM
    products
WHERE
    price > ANY
            (SELECT
        price
    FROM
        products
    WHERE
        id < 5);
```

と入力して、実行します。これは productsID が５未満の製品の、いづれかの価格よりも大きいデータが抽出されるようになります。リストを見てみると、２と８の ID 行のみが取得されていないようです。実際に中身を確認すると、２と８の ID はそれぞれ 5 と 12 となっており、ID5 未満の数字で最も小さい数字（ここでは ID2 の 12 ドル）より小さい数が対象外となっていることがわかります。

サブクエリのみを実行して、もう少し中身を確認してみましょう。

```
SELECT
    price
FROM
```

```
    products
WHERE
    id < 5;
```

49 ドル、12 ドル、40 ドル、38 ドルが取得されています。したがって、12 ドルよりも大きな価格のデータのみが取得されるサブクエリの条件となっていることがわかります。

```
SELECT
    *
FROM
    products;
```

で全てのデータを取得してみると、2 が 12 ドル、８が 5 ドルとなっていて、これが条件にはずれているためデータが取得されていないことがわかりました。

次に ANY ではなく、SOME を利用してみます。

```
SELECT
    *
FROM
    products
WHERE
    price > SOME
            (SELECT
        price
    FROM
        products
    WHERE
        id < 5);
```

と入力して、実行します。これは ANY と同じ結果となりました。ANY と SOME は全く同じコマンドとなっています。次に ALL にしてみましょう。

```
SELECT
    *
FROM
    products
WHERE
    price > ALL
            (SELECT
            price
        FROM
            products
        WHERE
            id < 5);
```

と入力して、実行します。ID5,6,7 が出力されたと思います。

どのようなことかというと、ID5 未満の数字で最も大きい数字（ここでは ID 1 の 49 ドル）より大きい数が対象となっていることがわかります。

ANY／SOME／ALL を利用することで、サブクエリ結果と比較して条件指定することができるようになることがわかりました。

■ケーススタディ演習１解説

それでは演習 1 の回答を実施します。

サブクエリの実行を検討するために、実際にとってくる users テーブルと、product_reviews テーブルを確認していきましょう。

SELECT * FROM product_reviews;

と入力して、実行するとレビューした製品に対して、ユーザーID が紐づけられていることがわかります。

SELECT * FROM users;

と入力して、実行して実際に取得するリスト元となるユーザーリストを確認します。さきほどの product_reviews;テーブルで製品 ID 5 のデータへのレビューをしているユーザーID を絞り込んで、WHERE 句のサブクエリとして条件づければ実現しそうです。

想定するサブクエリを実行してみましょう。

```
SELECT
        user_id
    FROM
        product_reviews
    WHERE
        product_id = 5;
```

と実行してみると、product_id = 5 へのレビューを実施したユーザーID が取得できます。これを

メインのクエリの WHERE 句の IN に関連付けていけば目的は達成できそうです。

```
SELECT
    *
FROM
    users
WHERE
    id IN (SELECT
            user_id
        FROM
            product_reviews
        WHERE
            product_id = 5)
ORDER BY users.id;
```

と入力して、ORDER BY で順番を ID 順にした上で実行します。上手くいきました。

これで演習内容としては正解ですが、JOIN を利用した場合の回答も実施していきます。

users テーブルと、product_reviews テーブルを JOIN を利用して結合して、product_id = 5

の条件をつければ実現できそうです。

```
SELECT
    users.*
FROM
    users
        INNER JOIN
    product_reviews ON product_reviews.user_id = users.id
WHERE
```

```
    product_id = 5
ORDER BY users.id;
```

と入力して、実行します。すると重複したデータが算出されてしまい、若干サブクエリを利用した結果と異なります。なぜ重複してしまうのか確認していきましょう。

product_reviews テーブルから今回の対象となる user_id を取得します。

```
SELECT
    user_id
FROM
    product_reviews
WHERE
    product_id = 5
ORDER BY product_reviews.user_id;
```

と入力して、実行します。このようにレビューが二回以上登録されているユーザーがいるようです。JOIN 句の場合は、これは別のレビューとして認識しているため INNER JOIN でも重複を排除していないようです。

したがって、出力するデータに DISTINCT を追加することで、サブクエリの実行結果と同じにする必要があります。

```
SELECT
    DISTINCT users.*
FROM
    users
        INNER JOIN
```

```
    product_reviews ON product_reviews.user_id = users.id
WHERE
    product_id = 5
ORDER BY users.id;
```

と入力して実行します。これでサブクエリと JOIN 句で同じ結果を抽出することができました。

これでわかるように、サブクエリの方が効率的にクエリを作ることができます。

■ケーススタディ演習 2 解説

それでは演習 2 の回答を実施します。

まずは前半の「それぞれの性別でグループ化をして注文総数（amount）をジェンダー別に合計してください。」という内容を実行してみます。

どのテーブルが必要であるのかを確認していきましょう。

SELECT * FROM users;

ここの gender をグループ化で利用することになります。

SELECT * FROM order_details;

つぎに order_details テーブルを確認すると集計対象となる amount があります。これを合計する必要があるため、SUM を利用することになります。ここには user_id がないため、users テーブルとの連携ができないため、もう 1 つテーブルが必要です。それが orders テーブルになります。

SELECT * FROM orders;

orders テーブルを確認すると、user_id と order_id とを関連付けることができそうです。ここの ID が order_details テーブルの order_id となっています。

３つのテーブルを結合する SELECT 文を作成します。

```
SELECT
        gender,
        SUM(amount) AS sum
    FROM
         order_details
        JOIN
                orders ON orders.id = order_details.order_id
        JOIN
                users ON users.id = orders.user_id
    GROUP BY users.gender;
```

と入力して、実行すると、gender でグループ化されて注文総数を合計した結果が算出されます。

次に後半の「その上で、さらに平均値、最大、最小に集計結果を分けてデータを抽出してください。」に対応していきましょう。

先ほどのクエリをサブクエリとして FROM 句内に設置して、さらに SELECT 文でデータを集計します。

```
SELECT
        AVG(sub.sum), MIN(sub.sum), MAX(sub.sum)
FROM
    (SELECT
         SUM(amount) AS sum
    FROM
         order_details
```

```
    JOIN
        orders ON orders.id = order_details.order_id
    JOIN
        users ON users.id = orders.user_id
GROUP BY users.gender) AS sub;
```

と入力して実行すると、各集計結果の平均値と最小値と最大値が算出されました。

■ケーススタディ演習 3 解説

それでは演習 3 の回答を実施します。

まずはレビューの列を確認してみましょう。

SELECT * FROM product_reviews;

と入力して実行すると、body 列にコメントがあることがわかります。

プロダクト ID 7 に対して、disappointing のレビューコメントがあるというデータ抽出を行うため

には、LIKE 句を利用します。

```
SELECT
      *
FROM
      product_reviews
WHERE
      product_id = 7
            AND
   body LIKE '%disappointing%';
```

というコマンドで実施できます。いくつかのレビューがヒットしました。

これをサブクエリにして、EXISTS を利用して、users テーブルを検索します。

```
SELECT
      *
FROM
      users
```

```
WHERE
    EXISTS( SELECT
            user_id
        FROM
            product_reviews
        WHERE
            product_id = 7
                AND
                    body LIKE '%disappointing%');
```

と入力して実行すると、行が出てきたため、上手く確認できました。

次のこのコマンドを IN 句に変えて、レビューを行ったユーザーを特定します。

```
SELECT
    *
FROM
    users
WHERE id IN( SELECT
            user_id
        FROM
            product_reviews
        WHERE
            product_id = 7
                AND
                    body LIKE '%disappointing%');
```

と入力して、実行するとプロダクト ID 7 に対して、低い評価を行ったユーザーが特定されました。

■ケーススタディ演習 4 解説

それでは演習 4 の回答を実施します。

初めに、ユーザーの注文データを利用して、ユーザーの個人情報と注文した商品の発注数を合わせて注文履歴リストを作成します。そ項目は Users テーブルから id、name, email, gender, birthday の 5 つを取得します。order_details テーブルから product_id と amount を取得します。

このクエリには、Users テーブルと order_details テーブルと、orders テーブルの 3 つのテーブルを結合してあげる必要があります。

まずは users テーブルと orders テーブルを結合していきます。それぞれ user_id を共通データとして指定して結合します。JOIN のタイプは「これまでに注文がないユーザーについてもリストとして抽出して把握できるようにする」ことが条件であったため、OUTER JOIN で users テーブル側のデータをすべて表示させることが必要となります。users テーブルは左テーブルですので、LEFT JOIN を利用します。

```
SELECT
    *
FROM
    users
        LEFT JOIN
    orders
    ON users.id = orders.user_id;
```

と入力して、実行します。これで結合ができました。

次に orders テーブルに対して、order_details テーブルを結合させます。orders テーブルと order_details テーブルは過不足なくセットで作成されるデータであるため、INNER JOIN を利用して結合していきますが、LEFT JOIN でも結果は変わりません。指定する共通データは orders.id です。

```
SELECT
    *
FROM
```

```
    users
        LEFT JOIN
    orders
  ON users.id = orders.user_id
        INNER JOIN
    order_details
  ON order_details.order_id = orders.id;
```

と入力して、実行します。これで結合ができました。

あとは表示するカラム名を

Users テーブルから id、name, email, gender, birthday の 5 つをとってくることにします。

order_details テーブルから product_id と amount を取得するように絞り込んでリストを見やすくしましょう。

```
SELECT
```

```
    users.id, users.name, users.email, users.gender, users.birthday,
order_details.product_id, order_details.amount
FROM
    users
        LEFT JOIN
    orders
    ON users.id = orders.user_id
        INNER JOIN
    order_details
    ON order_details.order_id = orders.id;
```

となります。

次に、作成した注文履歴リストに対して、注文した商品名と合計金額（税込み／税別）がわかるようにリストにデータを追加してください。結果は user_id で昇順で表示されるようにします。

商品名と金額は products テーブルにあるため、これを結合する必要があります。

```
SELECT
    users.id, users.name, users.email, users.gender, users.birthday,
order_details.product_id, order_details.amount, products.name,
amount * price AS total_fee
FROM
    users
        LEFT JOIN
    orders
    ON users.id = orders.user_id
        INNER JOIN
    order_details
    ON order_details.order_id = orders.id
            INNER JOIN
    products
    ON order_details.product_id = products.id;
```

これで目的のリストができました。さらに消費税がわかるようにもしてみましょう。消費税は10%と仮定して設定していきます。また、結果は user_id で昇順で表示されるようにします。

```
SELECT
    users.id, users.name, users.email, users.gender, users.birthday,
order_details.product_id, order_details.amount, products.name,
amount * price AS total_fee, amount * price * 1.1 AS tax_included
FROM
    users
        LEFT JOIN
    orders
    ON users.id = orders.user_id
        INNER JOIN
    order_details
    ON order_details.order_id = orders.id
            INNER JOIN
    products
```

```
    ON order_details.product_id = products.id
    ORDER BY users.id;
```

で実行すると、合計金額が記載された注文履歴リストが完成しました。次に、ユーザーごとの合計発注金額を抽出してください。その際に、これまでの合計注文数がわかるようにしてください。

これは GROUP BY を利用してユーザーID でまとめればよいということはわかると思います。それでは GROUP BY を付け加えてみましょう。また、金額を合計するには SUM を利用していく必要があります。

```
SELECT
    users.id, users.name, users.email, users.gender, users.birthday,
order_details.product_id, order_details.amount, products.name,
SUM(amount * price) AS total_fee, SUM(amount * price * 1.1) AS
tax_included
FROM
    users
```

```
        LEFT JOIN
    orders
    ON users.id = orders.user_id
        INNER JOIN
    order_details
    ON order_details.order_id = orders.id
                INNER JOIN
    products
    ON order_details.product_id = products.id
      GROUP BY users.id
    ORDER BY users.id;
```

これでユーザーごとの全ての注文に対する合計金額が算出されました。Product_id と発注数と商品名の列がユーザーごとの 1 行目を表示しており、価格と不整合が発生しているため、リストから除外します。

```
SELECT
```

```
    users.id, users.name, users.email, users.gender, users.birthday,
SUM(amount * price) AS total_fee, SUM(amount * price * 1.1) AS
tax_included
FROM
    users
        LEFT JOIN
    orders
    ON users.id = orders.user_id
        INNER JOIN
    order_details
    ON order_details.order_id = orders.id
            INNER JOIN
    products
    ON order_details.product_id = products.id
        GROUP BY users.id
    ORDER BY users.id;
```

となります。

最後にこれまでの合計注文した数がわかるように COUNT を利用して order の数を数えます。

```
SELECT
    users.id, users.name, users.email, users.gender, users.birthday,
COUNT(order_id) AS order_count, SUM(amount * price) AS total_fee,
SUM(amount * price * 1.1) AS tax_included
FROM
    users
        LEFT JOIN
    orders
    ON users.id = orders.user_id
        INNER JOIN
    order_details
    ON order_details.order_id = orders.id
            INNER JOIN
    products
    ON order_details.product_id = products.id
```

```
    GROUP BY users.id
ORDER BY users.id;
```

最後に HAVING 句を利用して、ユーザーごとの合計発注金額が products テーブルにある製品価格(price)の合計金額よりも少ないデータを抽出します。

```
SELECT
    users.id, users.name, users.email, users.gender, users.birthday,
SUM(amount * price) AS total_fee, SUM(amount * price * 1.1) AS
tax_included
FROM
    users
        LEFT JOIN
    orders
    ON users.id = orders.user_id
        INNER JOIN
    order_details
```

```
    ON order_details.order_id = orders.id
                INNER JOIN
    products
    ON order_details.product_id = products.id
        GROUP BY users.id
    HAVING SUM(amount * price) < ( SELECT SUM(price) FROM
products)
    ORDER BY users.id;
```

と追加して、実行すると全ての製品の合計金額よりも低いユーザー毎の合計金額となるデータが抽出されました。