できるDigital Camp

集中演習

230問のドリルと演習用ファイル付き

SQL入門

Google BigQueryではじめるビジネスデータ分析

木田和廣

株式会社プリンシプル　取締役副社長

前提知識もインストールも不要。すぐに実践できる

大規模データからの抽出、整形が思い通りに！

業務での分析に活用できます

売上分析　統計分析　CRM分析　Web解析

インプレス

できるDigital Camp

集中演習

230問のドリルと演習用ファイル付き

SQL入門

Google BigQueryではじめるビジネスデータ分析

木田和廣
株式会社プリンシプル　取締役副社長

前提知識もインストールも不要。すぐに実践できる

大規模データからの抽出、整形が思い通りに！

業務での分析に活用できます

売上分析　統計分析　CRM分析　Web解析

インプレス

AUTHOR PROFILE

木田和廣（きだ かずひろ）

株式会社プリンシプル 取締役副社長
Web解析チーフ・エバンジェリスト

早稲田大学政治経済学部卒業。商社の海外駐在員などを経て、2004年にWeb解析業界でのキャリアをスタートする。

2009年からGoogleアナリティクスに基づく解析コンサルティングに従事し、個人資格「GAIQ」を継続保有。2015年にはGoogleアナリティクス公式コミュニティで日本初の「トップコントリビューター」に認定される。同年にGoogleアナリティクス解説書のベストセラー『できる逆引き Googleアナリティクス Web解析の現場で使える実践ワザ 240』を執筆、2017年にはその『増補改訂2版』を発刊する。

BIツールのTableauにも習熟し、2016年に市販書籍としては日本初の入門書『できる100の新法則 Tableau ビジュアルWeb分析』を上梓。2017年には上級資格「Tableau Desktop Certified Professional」を取得する。

アナリティクスアソシエーション（a2i）や個別企業でのセミナー登壇、トレーニング講師実績も多数。統計検定2級保有。

株式会社プリンシプル
https://www.principle-c.com/

はじめに

これまで、データベースを操作する言語である「ＳＱＬ（エスキューエル）」は、もっぱらエンジニアが持つべき素養と思われていました。

しかし、状況は急速に変わり、今やデータ分析に関わるすべてのビジネスパーソンが持つべき素養になってきています。その変化を引き起こしたのは、誰もが簡単にデータを取得・分析でき、ビジネスの意思決定に生かせるようになった「データ分析の民主化」です。Tableauなどのセルフサービス型BIツールの人気も、同じトレンドから発生したものです。

近年の企業においては、CRM（顧客管理）、SFA（営業支援）、Web解析、Web広告、メール配信、A/Bテスト、アンケートなどのツールやシステムから出力可能な、大量かつ多様なデータが蓄積されています。しかし、そうしたデータを分析するにあたり、データの抽出や整形といった作業をシステム部門に都度依頼しているのでは、業務上求められるスピードが出せなくなっている現実があります。

この現実を打破するには、企画職やマーケティング職のみならず、営業職を含めたエンジニアではないすべての職種で、自らデータを整形してクイックに分析したり、BIツールで視覚化する前段階の処理を行ったりする必要があります。これが、ビジネス素養としてのSQLの重要性が高まっている背景です。

本書は前提知識がゼロでも学べるよう、Googleが提供する「BigQuery（ビッグクエリ）」というデータベースサービスを利用することで、SQLの学習にあたって最初の障壁となる「データベースソフトウェアのインストール」のステップを回避しました。固有のデータベースに依存しない標準SQLに準拠しつつ、基礎的なSQL文の学習からスタートし、テーブルの結合やサブクエリ、そして最終的には、難易度は高いものの分析には欠かせないウィンドウ関数までをカバーしています。

SQLは実際に手を動かすことで上達しますが、そのためのドリルは全230問を用意しました。特に巻末の演習ドリルは、データ分析の実務で直面する可能性の高いシナリオに沿って設計し、実践的なスキルが得られるように工夫しています。

本書を手に取っていただいたみなさんが、業務で使えるSQLを身につけ、「データ分析力が飛躍的にアップした」と実感してもらえればうれしく思います。

2021年１月　木田和廣

CONTENTS

著者プロフィール …… 002
はじめに …… 003
本書の読み方 …… 009

CHAPTER 1
はじめてのSQL …… 011

1-1 SQLを学ぶ意義 …… 012
1-2 データベースとSQL …… 023
1-3 覚えておきたい用語 …… 027

CHAPTER 2
BigQueryの利用開始 …… 033

2-1 アカウントの開設 …… 034
2-2 アカウントの構造とプロジェクトの作成 …… 039
2-3 料金体系と無料枠 …… 044
2-4 演習用ファイルの内容とスキーマの定義 …… 046
2-5 データセットとテーブルの作成 …… 050
2-6 コンソールの操作 …… 059
2-7 SQL文の記述作法 …… 074

CHAPTER 3

SQLの基本構文 …… 083

3 - 1　列の取得と並べ替え …… 084
3 - 2　列の作成・演算と別名の付与 …… 091
3 - 3　行の絞り込み …… 098
3 - 4　確認ドリル …… 121

CHAPTER 4

グループ化とデータの集計 …… 127

4 - 1　グループ化 …… 128
4 - 2　集計関数 …… 136
4 - 3　集計結果の絞り込み …… 149
4 - 4　柔軟なグループ化 …… 154
4 - 5　確認ドリル …… 165

CHAPTER 5

テーブルの結合と集合演算 …… 173

5-1 複数のテーブルが存在する理由 …… 174
5-2 テーブルの結合 …… 179
5-3 複数条件や3テーブル以上の結合 …… 188
5-4 同一テーブルの結合とnull値の置換 …… 200
5-5 テーブルの集合演算 …… 210
5-6 仮想テーブルの作成 …… 225
5-7 確認ドリル …… 237

CHAPTER 6

サブクエリ …… 245

6-1 サブクエリの記述と戻り値の利用 …… 246
6-2 サブクエリの実践的な活用例 …… 264
6-3 UNNEST関数によるGA4テーブルの変換 …… 288
6-4 確認ドリル …… 297

CHAPTER 7

さまざまな関数 …… 307

7-1 数値を扱う関数 …… 308
7-2 文字列を扱う関数 …… 317
7-3 正規表現を利用する関数 …… 327
7-4 日付・時刻を扱う関数 …… 335
7-5 統計集計関数 …… 350
7-6 確認ドリル …… 357

CHAPTER 8

ウィンドウ関数 …… 365

8-1 ウィンドウ関数とは …… 366
8-2 ランキング・行番号の取得 …… 380
8-3 最初の値・別レコードの値の取得 …… 391
8-4 累計・移動平均の取得 …… 412
8-5 確認ドリル …… 421

PRACTICE

演習ドリル

P-0 演習ドリルへの取り組み方 …… 432
P-1 第1群 …… 435
P-2 第2群 …… 441
P-3 第3群 …… 451
P-4 第4群 …… 458
P-5 第5群 …… 469

おわりに …… 474
索引 …… 475

本書は2021年1月現在での情報を掲載しています。
本文中の製品名やサービス名は、一般に各開発メーカーおよびサービス提供元の商標または登録商標です。なお、本文中には™および®マークは明記していません。

本書の読み方

解説セクション

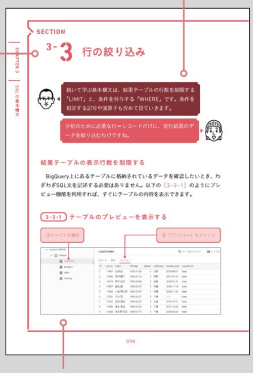
本書はCHAPTER（章）とSECTION（節）の2つの階層で構成され、SECTIONごとに解説が進んでいきます。SECTIONの開始ページにある番号は「章 - 節」の関係になっています。

著者（講師）との会話

本書の著者と、みなさんと一緒に学習に取り組む仲間たちの会話です。これから解説することや素朴な疑問、理解のヒントについて言及しています。

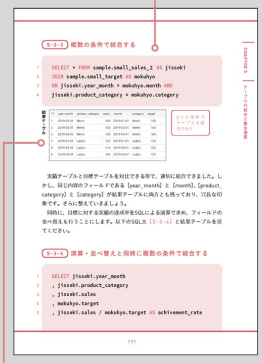
SQL文

SQLのコードです。命令語には色を付け、行番号も付記しています。見出しの番号は「章 - 節 - 節内の通し番号」です。



操作手順・図表

BigQueryの画面を使った操作手順や、本文を補足する図や表です。見出しの番号はSQL文などと共通になっています。



結果テーブル

SQL文をBigQueryで実行したときに返される結果テーブルです。多くのSQL文は、本書の演習用ファイルから同じ結果を得られます。

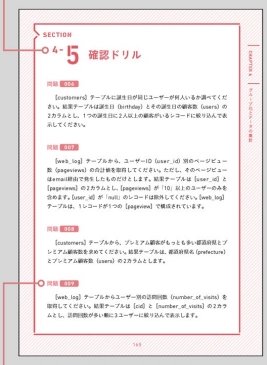
STEP UP

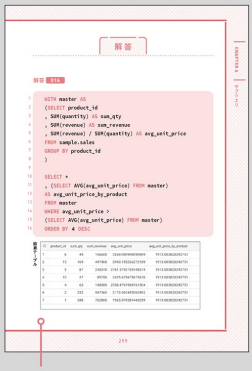
ステップアップ

本文で解説した方法とは別の方法や、より複雑な分析で役立つSQL文の記述方法など、発展的な知識を紹介しています。

問題セクション

CHAPTER 3〜8の末尾には「確認ドリル」があり、その章で学習した内容の理解度をすぐにチェックできます。また、本書の最後には特別な章として、それまでの章で学んだことを実際のビジネスシーンで生かせるかどうかを問う「演習ドリル」を用意しています。





問題

演習用ファイルを対象に、目的の結果を得るためのSQL文を解答してください。問題番号はすべての章を横断した通し番号です。

解答

「確認ドリル」の解答となるSQL文と、その解説です。巻末の「演習ドリル」の解答は、下記の「できるネット」に掲載しています。

演習用ファイルのダウンロードについて

本書での学習に利用する「演習用ファイル」のダウンロード方法や、巻末の「演習ドリル」の解答は、Webメディア「できるネット」にて公開するサポートページに掲載しています。みなさんのPCでの動作確認や理解度のチェックにお役立てください。

※演習用ファイルのダウンロードには「CLUB Impress」への会員登録（無料）が必要です

https://dekiru.net/dc_sql

CHAPTER 1

はじめての SQL

1-1　SQLを学ぶ意義
1-2　データベースとSQL
1-3　覚えておきたい用語

これからみなさんが学ぶSQLについて、土台となる知識を身につけましょう。本書では「ビジネスに役立つ知見を得るためのデータ分析」を目的にSQLを学びますが、なぜ、その目的のためにSQLが重要なのかを理解してください。また、データベースとSQLの関係や、後続の章で何度も登場する用語についても整理します。

SECTION

1-1 SQLを学ぶ意義

最初に「なぜSQLを学ぶ必要があるのか？」「学ぶべき分野の中で、なぜSQLなのか？」について整理します。

僕は営業部門のマーケティングチームに所属していますが、そもそもプログラミングを学んだことがありません。それでもSQLを使えるようになるでしょうか？

大丈夫です。本書は前提知識がゼロでも学べるように設計しました。プログラミング経験も問いません。

私たちが扱うデータがSQLのスキルを要求する

「ビッグデータの時代」と言われて久しいですが、日頃からデータ分析に携わるビジネスパーソンのみなさんは、すでに**業務上で扱うデータのビッグデータ化**を実感している人も多いと思います。

もし「自分の取り扱うデータは、あまりビッグデータ化していない」という人は、変化に気付いていないだけかもしれません。ビックデータ化は、**どのようなビジネスでも進行している不可避のトレンド**です。

一般にビッグデータ化とは、次に示した「3つのV」が、より大きい方向にシフトする事象を指しています。これらはビッグデータの性質や特徴を表すうえで、よく登場するキーワードです。

- Velocity ⇒データが集まる速度・更新される速度
- Variety ⇒データの種類・多様性
- Volume ⇒データの量

このうち、理解しやすいVolumeについて考えてみましょう。以下の表［1-1-1］は、定番のWeb解析ツール「Googleアナリティクス」から参照できる「メディア」レポートのデータを、サンプルとして抜粋したものです。

1-1-1 Googleアナリティクスの「メディア」レポート

メディア	セッション	ページビュー数	コンバージョン	コンバージョン率
organic	54,874	176,146	1,010	1.84%
cpc	48,551	91,276	976	2.01%
social	25,847	65,134	284	1.10%
direct	12,457	27,530	148	1.19%
referral	11,984	23,608	114	0.95%
email	9,574	28,818	191	1.99%
合計	163,283	412,512	2,723	1.67%

メディアとは、ユーザーの訪問経路を表すGoogleアナリティクスのディメンション＝分析軸の1つです。organic（自然検索）、cpc（リスティング広告）、social（SNS）などの値があります。

Webサイトにおける目標の達成のことを「コンバージョン」と呼びますが、上記の表のうち、もっともコンバージョン率が高いのは「cpc」です。よって、これしかデータがないのなら、得られる知見は「コンバージョンをアップさせるために広告の予算を増やしましょう」となります。

しかし、たったこれだけのデータで判断するのはあまりにも乱暴で、ビジネス上の意味がある知見とはいえません。実際、単に広告の予算を増やしたからといって、コンバージョンが増える可能性は低いでしょう。

そこで、上記の表に「デバイスカテゴリ」という列を加えたと仮定しましょう。デバイスカテゴリは、desktop（パソコン）、mobile（スマートフォン）、tablet（タブレット）という3種類の値を持ちます。6種類のメディア、つまり6行しかなかったデータは、6×3で18行に増えます。

さらに、曜日（7種類）、時間（24種類）、都道府県（47種類）の列も加えたと仮定すると、6×3×7×24×47という掛け合わせになり、最大142,128行もの表になります。デバイスカテゴリ、曜日、時間、都道府県はいずれも、Googleアナリティクスにおける標準のディメンションです。

Googleアナリティクスなら、僕も自社サイトのページビューやコンバージョンの月次報告のために使っています。ディメンションの掛け合わせで、それほど大量のデータが入手できるんですね！

このようにしてデータの粒度を細かくすると、表の行数、つまりデータのVolumeが増えていきます。そして、**データが大型化するほど、得られる知見は詳細に**なっていきます。

［1-1-1］の表では「cpcのコンバージョン率が高い」ということしか分かりませんでした。しかし、複数のディメンションを掛け合わせた142,128行もの表を分析すれば、例えば「大都市圏で土日の午後22時から深夜0時までにスマートフォンを利用しているユーザーに対するcpcのコンバージョン率が高い」といった知見が得られるわけです。

ここまで詳細であれば、その広告の予算を増やす判断が妥当であると理解できます。また、そうしたユーザーが何を求めて広告をクリックし、コンバージョンしてくれたかを考えることで、広告のキャッチコピーやLP（ランディングページ）での訴求内容、CTA（Call To Action：ユーザーに行動を促すためのボタンなどの要素）の改善にもつながるでしょう。

データから得られる知見が詳細になれば、それだけ精度の高いビジネス上の「打ち手」が考えられるようになり、売上や利益にも貢献するはずです。

こうしたビッグデータ化の流れは、Googleアナリティクスの新バージョンである「**Googleアナリティクス4 プロパティ**」(旧称：アプリ＋ウェブプロパティ）にも引き継がれます。Googleアナリティクス4では、無償版であっても、データベースサービスである「Google BigQuery」へのデータのエクスポートが標準でサポートされるようになります。

BigQueryについてはCHAPTER 2で解説しますが、「SQLをBigQueryで学ぶ」が本書のテーマでもあり、みなさんにとって不可欠なツールになると確信しています。以下の［1-1-2］で示したGoogle公式ヘルプには、Googleアナリティクス4とBigQuery、SQLの関係が端的にまとまっているので、現時点ではまだ意味が理解しにくいかもしれませんが、一読してみてください。

1-1-2 Googleアナリティクス4のBigQuery Export

Google アナリティクス 4 プロパティの
BigQuery Export

BigQuery は、大規模なデータセットに対しパフォーマンスの高いクエリを実行できるクラウドデータ ウェアハウスです。

すべてのイベントを Google アナリティクス 4 プロパティから BigQuery にエクスポートし、SQL タイプの構文を使ってそのデータにクエリを発行できます。BigQuery では、アナリティクスのデータとの統合を目的に、データを外部ストレージにエクスポートすることや、外部データをインポートすることが可能です。

https://support.google.com/analytics/answer/9358801?hl=ja

そして、Googleアナリティクス4からBigQueryにエクスポートされたデータにアクセスすると、私たちは「ヒット単位」の情報を入手できます。「ヒット」とは、ユーザーがWebサイト利用をしたときにGoogleアナリティクスサーバーに送信される、すべての信号を指しています。つまり、「これ以上粒度を細かくできない、もっとも詳細なレベルのデータ」ということになります。

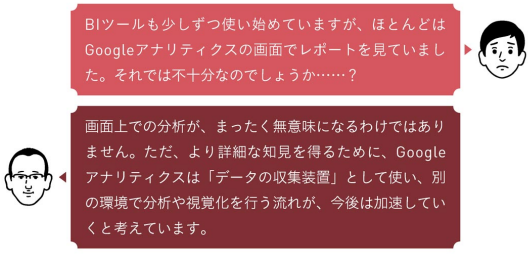
次ページの［1-1-3］に示したのは、Googleアナリティクス4からBigQueryに出力した1つのヒットで、ユーザーの初回のWeb利用を表す「first_visit」というイベントが記録された行です。1つのイベント（＝ヒット）に、

複数のパラメータが記録されているのが分かります。こうしたデータが文字通り無数に発生し、蓄積されていくのが、これからのWeb解析の常識になっていきます。

1-1-3 Googleアナリティクス4のヒットデータの例

行	event_date	event_timestamp	event_name	event_params.key	event_params.value.string_value
1	20200731	1596157508337999	first_visit	page_location	https://www.principle-c.com/column/seo/seo-crawl-data/
				session_engaged	null
				ga_session_number	null
				page_title	あなたの知らないクロールの統計情報の世界 | 株式会社プリンシプル
				engaged_session_event	null

私たちは、これほどまでに細かくなった粒度のデータと対峙し、ビジネスにおいて意味のある知見を導き出さなくてはなりません。そのためには、データベースを操る言語であるSQLが、どうしても必要になります。私たちが扱うデータが、私たちにSQLの知識・スキルを要求するのです。



SQLが使えれば「他人依存」から脱却できる

Googleアナリティクスに限らず、現在ではマーケティング、営業、経営企画、人事など、あらゆる部門のビジネスパーソンが、業務上で複数のツール

やサービス、各種システムを利用していると思います。マーケティングならGoogleやFacebook、Twitterなどの広告運用ツール、営業ならSalesforceなどのCRMツールが代表例でしょうか。

それらには、ツールの画面内でデータを見やすく表示する「ダッシュボード」機能が備わっていることがありますが、**ツールの標準機能としてのダッシュボードが、業務上の分析において必要条件を満たしていることは、ほぼありません**。なぜなら、そうした「お仕着せ」のダッシュボードでは対応できない次のような要望が、現場から次々に出てくるからです。

- 時系列分析をもっと時間を広げて行いたい
- 別のディメンションを適用してデータを見たい
- 前年同期比のデータが見たい

こうしたニーズに対応するため、ツール側では「生データのCSVダウンロード」機能を提供していることがあります。生データは加工されていない、もっとも粒度の細かいデータなので、ユーザー側に分析が委ねられます。

生データのCSVファイルは、ツールの種類や利用状況にもよりますが、数十万行、時には数百万行にも及びます。Excelで扱うには難しいボリュームです。また、「Tableau（タブロー）」や「Power BI」などのBIツールで視覚化するにしても、生データとダイレクトに連携すると複雑なフィルタ操作などが必要になったり、動作が重くなったりする原因になります。

そこで、SQLの出番です。**CSVファイルの分析や、BIツールに取り込む前のデータの抽出・整形といった前処理**で、SQLのスキルは大いに役立ちます。逆に、分析者がSQLのスキルを持たない場合、ツールから生データをダウンロードしても、宝の持ち腐れとなってしまう可能性が高いでしょう。

ユーザー側に自由に分析できるデータを与えるという意味で、生データのダウンロードに対応しているか否かは、ツールの選定基準の1つといえます。

また、分析したいデータが社外のツールではなく、自社がオンプレミスで運用している基幹システムに格納されていることもあるでしょう。この場合、分析を担うビジネスパーソンは社内のエンジニアに依頼して、希望するデータを抽出してもらうことになります。

しかし、分析のためのデータ抽出は、その性質上、1回では終わらないのが普通です。抽出したデータを分析してはじめて、データの過不足に気付いたり、もう少し細かいデータが必要になったりする性質があるためです。

すると、分析者に何の落ち度がなくても、エンジニアにデータ抽出を何回か依頼することになります。もちろん、分析者に何らかの誤解や、データを熟知していないことによる勘違いがあれば、その回数分のデータ抽出を依頼しなければなりません。こうしたやりとりは、分析者・エンジニアの双方にとって、生産性を下げる原因となるでしょう。

このようなエンジニアへの「他人依存」を避け、**分析のスピードを速めるための根本的かつ本質的な解決策は、分析者自らがデータ抽出を行うこと**です。そのための最大の武器がSQLのスキル、というわけです。

BigQueryを利用すれば、エンジニアほどの知識を持たない人でも、CSVファイルのデータからテーブルを作成し、SQLを実行して希望のデータ抽出・加工・分析が行えます。少しのやる気さえあれば、誰でもデータベースとSQLを使える環境は、すでに整っているのです。

SQLは環境依存のない汎用的なスキル

「BIツールに習熟していれば、SQLは必要ないのではないか？」

筆者には、このような質問が時々寄せられます。確かに、BIツールは内部で分類や集計などのSQL的な処理を行っているので、高度に習熟していれば、かなり柔軟に業務上必要とされる分析を視覚的に行うことができます。筆者も日常的にTableauを業務に使っているので、質問者の意図は理解できます。

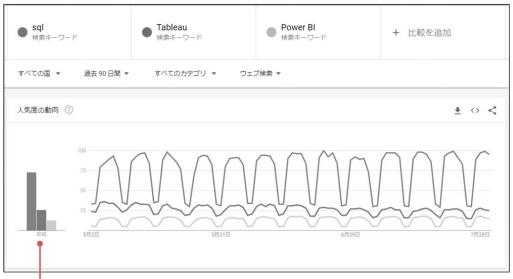
しかし、筆者はBIツールが使えても、SQLは分析者が持つべきスキルである

と確信しています。その理由の1つは、企業や部署によって採用されているBIツールが異なる場合があるからです。

あらゆるBIツールに精通しているビジネスパーソンはさすがに少数だと思われ、多くの人はツールによって得意・不得意があるでしょう。よって、転職前の企業で使っていたBIツールには習熟していたが、転職後の企業では導入されておらず、別のBIツールを利用することになった、ということはいかにも起こりえます。結果、少なくとも一時的に、分析力は低下してしまいます。

一方、SQLは、それが実行できないデータベースはないといっても過言ではありません。**BIツールよりもSQLのほうが汎用的なスキル**だといえます。参考までに、Googleトレンドで「SQL」「Tableau」「Power BI」の検索ボリュームを比較すると以下の［1-1-4］のようになり、社会全体での関心の度合いとしても、SQLのほうが幅広いことが理解できます。

1-1-4 Googleトレンドにおける検索ボリューム



「SQL」の検索ボリュームがもっとも大きく、次いで「Tableau」「Power BI」の順となっている

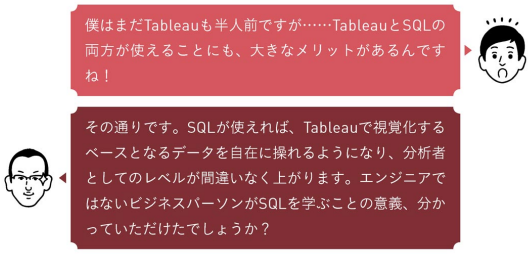
BIツールが使えても、分析者はSQLのスキルを持つべきである理由は、もう1つあります。かなり高度なゴールではありますが、それらの両方が使えるよとで、より「筋のよい」分析が可能になる、というものです。

例えば、BIツールとしてTableau Desktopに習熟し、データプレパレーション（加工・変換）ツールとして「Tableau Prep」と「Exploratory」が使え、さらにSQLのスキルも持つ分析者がいるとしましょう。その人が数百万行からなるWeb広告のレポートについて月別・広告グループ別の分析をする場合、次のように適材適所でのアウトプットが可能になります。

- 月別の集計はSQLで行い、データマートとして利用
- 月別目標CV金額データとのJOINはTableau Prepで行う
- 月別CV金額の時系列予測はExploratoryで行う
- 以上の結果のダッシュボード作成はTableau Desktopで行う

「データマート」とは、データベースから特定の目的のために取り出した部分のことで、まさにSQLによって実現できます。「JOIN」はデータベースを結合する操作のことで、CHAPTER 5で詳しく学びます。

このようなアウトプットができると、もっとも効率がよく、インパクトの大きな分析が最短の手間で可能になります。**BIツールしか使えない分析者とは「引き出し」の多さに違いが出る**といえるでしょう。



STEP UP

筆者のSQL活用事例

筆者もエンジニアではなく、みなさんと同じ現役の分析者です。デジタルマーケティングを支援するコンサルタントとして、お客さまとなる企業のデータをお預かりし、分析結果をレポートすることを仕事としています。

これからSQLを学ぶみなさんにとって、現時点ではちょっと難しい話かもしれませんが、SQLの有効性を示す一端として、筆者が直近の業務でSQLを最大限に活用した事例を紹介しましょう。

要件としては、コストがかかる販売促進キャンペーンを「その施策を行ったからこそ購入するであろう顧客」だけにターゲットを絞って行いたいので、その条件に合致する顧客を抽出してほしい、というものでした。クライアントから提供していただいた元データは、次のように多岐にわたります。

▶顧客別購入データ

購入した1つの商品が1レコードになっているファイルです。これまでに3回購入したことがある顧客が、初回に1商品、2回目に2商品、3回目に3商品を購入した場合、この顧客のレコード数は「6」となります。商品はコードで記述されており、それだけでは何を購入したかは分かりません。顧客はコードで識別されており、年齢・性別などは分かりません。

▶商品マスタ

商品コードと商品カテゴリ、商品名を紐づけているファイルです。

▶顧客マスタ

顧客コードとデモグラフィックな属性が記録されています。

▶過去の施策対象顧客リスト

年に10回程度、定期的に行う施策の対象となったユーザーのリストが10個程度のCSVファイルとして提供されました。

上記のデータを1顧客1行にまとめ、「過去に施策実施対象となり、かつ施策実施日から2週間以内に購入したことがある」顧客を「反応履歴あり」とし、目的変数としました。説明変数は、次に挙げる30余りです。

- 年齢
- 性別
- 居住都道府県
- 初回購入日からの経過日数
- 購入回数
- LTV
- 初回注文の金額
- 初回購入商品のカテゴリ（初回に複数カテゴリの商品を購入していた場合は、もっとも金額が大きい商品のカテゴリ）
- もっとも購入した商品カテゴリ

これらのデータをすべてBigQuery上に展開し、SQLを実行して抽出・整形していきました。そして、完成した1顧客1行で1つの目的変数、30余りの説明変数を持つデータに対して、「BigQuery ML」のロジスティック回帰アルゴリズムで目的変数を予測します。「ML」は「Machine Learning」、つまり機械学習のことで、BigQuery MLはその名の通り、BigQuery上でMachine Learningを実現する機能のことです。

そのうえでクライアントには、販売促進キャンペーンを「反応履歴あり」の予測が「TRUE」の顧客だけに行ってもらいました。予測はおおむね正しく、販売促進キャンペーンを行ったユーザーの購入率は十分に高い結果を示し、筆者にとっても成功案件となりました。

SECTION

1-2 データベースとSQL

「データベース」と聞いて、みなさんはどのようなものを思い浮かべますか？

私は商品企画をしています。データベースというと、商品やお客さまのデータ、過去のアンケートデータなどをまとめたもの、というイメージです。

僕はWebマーケティング担当なので、ECでの販売データや、自社サイトのアクセスデータなどが思い浮かびますね。

ふたりとも、正しい認識をお持ちですね。ここではSQLの成り立ちと、その実行対象となる「リレーショナルデータベース」について、おさらいしておきます。

複数の「テーブル」で構成されたデータベースを操作

SQLをひと言で説明すると、**リレーショナルデータベースの管理や操作を行うための言語**となります。SQLは「Structured Query Language」の略とされ、構造化された（Structured）問い合わせ（Query）のための言語（Language）と説明できますが、諸説あり定まっていないようです。

SQLの実行対象となるリレーショナルデータベース（Relational Database｜RDB）は、**複数の「表」の形式でデータを管理するデータベースシステム**を指します。概念的には次ページの［1-2-1］のようになり、これらの1つ1つの表のことを「テーブル」と呼びます。

リレーショナルデータベースにおいては、例えば［user_id］［product id］などの共通するデータにより、テーブル同士に関係性を持たせることができます。データを複数のテーブルで管理することのメリットについて、詳しくはCHAPTER 5で学びますが、この関係性が「リレーショナル」データベースと呼ばれるゆえんです。

1-2-1 リレーショナルデータベースの概念図

販売テーブル

order_id	user_id	product_id	date_time	quantity	revenue
158784	15598	11	2018-10-06 09:13:04	1	1200
148763	14184	3	2019-09-06 22:22:42	2	3600
191638	11026	1	2017-07-09 15:26:59	1	2000
117808	17690	2	2019-09-13 19:45:32	1	1980

複数のテーブル（表）の関係性でデータベースが構成されている

顧客テーブル

user_id	name	birthday
11478	石塚 拓	1978-11-26
14486	長坂 賢介	1992-07-18
14118	野中 裕之	1994-05-08
13507	高松 龍	1983-02-07

商品テーブル

product_id	product_name	cost
1	レアチーズケーキ	400
2	ベイクドチーズケーキ	500
3	ショートケーキ	600
4	モンブラン	500

表形式でデータを扱うという意味ではExcelに似ていますが、RDBではより大規模なデータを、SQLを用いて柔軟に扱うことが可能になります。

SQLは、もともとはIBMやOracleといったリレーショナルデータベース管理システム（Relational Database Management System｜RDBMS）を販売する企業により、自社のデータベース製品を操作するための言語として開発されました。その後、ISO（国際標準化機構）やANSI（米国国家規格協会）が「標準SQL」として規格を定めています。標準SQLの規格は数年に一度改訂され、

その年号をとって「SQL:2011」「SQL:2016」などと呼ばれます。

データベース製品によっては、「拡張SQL」などと呼ばれる独自の文法が存在しており、本書の学習環境として利用するBigQueryも、かつては独自の文法にのっとっていました。しかし現在では、そうした独自文法をレガシー（過去の遺産的）な文法と位置づけ、標準SQLへの準拠を強めています。同様のデータベースは増加傾向にあり、本書で**標準SQLを学ぶことで、BigQueryをはじめとした幅広いデータベースへの対応が可能**です。

データ分析用途で学ぶ範囲とそうでない範囲

SQLは本来、次のような命令を含む巨大な言語体系です。

1. ユーザーに操作権限を与えたり、はく奪したりする
2. テーブルを作成したり、削除したりする
3. テーブルにデータを書き込んだり、更新したり、取得したりする

そして、本書では上記**3**の「取得したり」に関連するスキルのみを、集中的に学びます。「そこだけ？」と思われるかもしれませんが、これは本書がエンジニアではなく、データ分析力を向上させたい非技術部門のビジネスパーソンを対象としているからです。

もし、企業の技術部門に所属するエンジニアのみなさんがSQLのスキルを身につけたいのであれば、上記のすべてを学ぶ必要があります。具体的な業務としては、社内のデータベースにアクセス可能なユーザー（社員）の管理、基幹システムなどと連携するテーブルの作成・削除、他部門の要求に応じたデータの追加・更新・抽出などが挙げられるでしょう。すでに存在するSQL解説書の多くも、エンジニアを対象とし、上記のすべてに関係するスキルを網羅的に学べる内容となっています。

一方、本書が対象とするビジネスパーソンのみなさんにとって、SQLを学ぶ目的はデータ分析にあります。分析対象となるデータベースにアクセスするのは自分だけですし、**テーブルの作成は、CRMやWeb解析などのシステムから出力したデータをBigQueryにアップロードするだけで完了**します（CHAPTER

2を参照)。また、テーブル自体を新たに作成することはあっても、既存のテーブルにあとからデータを追加したり、更新したりすることはありません。

ただし、上記3の「取得したり」に関連するスキルについては、深い部分まで解説します。その代表例が、別名「分析関数」とも呼ばれる「ウィンドウ関数」や、Googleアナリティクス4からBigQueryにエクスポートされるテーブルの「ネストされたレコード」を取り扱う際に必要となる「UNNEST関数」です。SQLをデータ分析の目的で使うには欠かせませんが、反面、難解な範囲ではあるので、順を追って本書を読み進めて、ぜひ最後までマスターしてください。

前提知識ゼロで学べるのが本書の特徴ですが、データ分析の業務でしっかり使えることをゴールに置いています。SQLのスキルをさらに磨き、分析目的でも使いたいエンジニアのみなさんも大歓迎です！

なお、SQLはWebサイトやアプリにも組み込まれており、動的にページを生成したり、サイト内・アプリ内検索を実現したりする用途で利用されています。そうした用途では、ユーザビリティを向上するために「0.1秒でも早く結果を返す」ためのSQLを記述するスキルが求められます。

しかし、データ分析の用途では、分析結果を出すスピードは求められるにしても、SQLの実行結果が返ってくるスピードは求められないのが普通です。したがって、本書におけるSQLの記述は「もっとも早く結果が返るようにチューニングする」ことにはこだわらず、**「とにかく分析結果が出せればよい」ことを念頭に置いている**と、覚えておいてください。

ひとくちにSQLと言っても、どのような業務で使うかによって、学ぶべき範囲や重視するポイントが異なるのですね！

SECTION

1-3 覚えておきたい用語

以降では、データベースとSQLに関連する専門用語が多数登場します。ここで主な用語を覚えてください。

新しいことを覚えるのはワクワクしますね！

データベースの構造についての用語を理解する

CHAPTER 1の最後として、本書を読み進めるうえで必要となる、データベースとSQLの基本用語を理解しておきましょう。BigQueryに限らず、**どのようなデータベースを扱うときにも役立ち、ほとんどのデータベースで共通の用語**として使われます。

▶テーブル

データを格納する表のことです。表は、縦方向に並ぶ「行」と、横方向に並ぶ「列」からできています。例えば「ユーザーごとの年齢」の表には、[氏名] [年齢] という2つの列があり、[氏名] 列には「山田太郎」「鈴木花子」などが、[年齢] 列には「28」「24」といった値を格納する行が並んでいるはずです。そのような表をテーブルと呼びます。

▶カラム／フィールド

テーブル内の「列」のことを「カラム」と呼びます。[氏名] [年齢] [性別] という3つの列を持つテーブルは、「3カラムで構成されたテーブル」などと表現します。

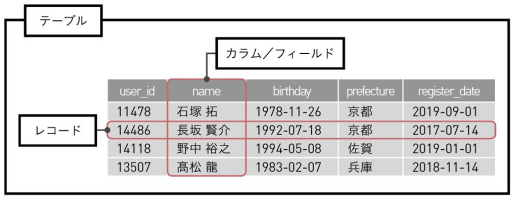
この列を指して「フィールド」と呼ぶこともあり、一般的には同義として扱われます。本書では「テーブルの構造としての列」を指す場合には「カラム」、「列

に格納された値」を指す場合には「フィールド」と記載しています。例えば『[revenue] フィールドの値を1.1倍して [revenue_with_tax] という新しいカラムに格納してください』といった具合です。

▶レコード

テーブル内の「行」のことを「レコード」と呼びます。「山田太郎」「鈴木花子」という2つの行が格納されている場合、「このテーブルには2レコードが存在する」などと表現します。テーブル、カラム／フィールド、レコードを図示すると、以下の [1-3-1] のようになります。

1-3-1 テーブル、カラム／フィールド、レコードの関係



▶データ型

Excelなどとは異なり、データベースではカラムごとに、そのカラムに格納するデータの「型」を最初に定める必要があります。この型のことを「データ型」と呼びます。なお、最初に定めた型と異なるデータは格納できません。データ型には、後述する文字列型、整数型、日付型などがあります。

▶スキーマ

テーブルを構成する、通常複数の列の属性をまとめたものを「スキーマ」と呼びます。次ページの [1-3-2] は、本書の演習用ファイルからBigQuery上に作成した [sales] テーブルのスキーマです。

1-3-2 スキーマの例

フィールド名	タイプ	モード
order_id	INTEGER	NULLABLE
user_id	INTEGER	NULLABLE
product_id	INTEGER	NULLABLE
date_time	DATETIME	NULLABLE
quantity	INTEGER	NULLABLE
revenue	INTEGER	NULLABLE
is_proper	BOOLEAN	NULLABLE

テーブルを構成する列の属性がまとめられている

▶クエリ

データベースに対する問い合わせのことを「クエリ」と呼びます。具体的には、データベース内の任意のテーブルを対象に、データの取得や演算、関数の利用、テーブルの結合といった処理を命令する、SQLによって記述されたコードとなります。このコードは「SQL文」とも呼び、クエリとほぼ同義です。

▶ビュー

テーブルに対してSQLを実行すると、返ってくる結果も、通常は表の形をしています。その表を再利用したい場合、毎回SQLを書くのは手間がかかります。そこで、SQLの実行結果の表を記録しておくのが「ビュー」です。

テーブルは実体を持った表であるのに対し、ビューはあくまでもSQLの結果が表の状態で保存されているものです。そのため、ビューに対してSQLを実行すると、結果が変わることがあります。ビューについて詳しくはSECTION 5-6で解説します。

▶null（ヌル）

「値がない」ことを表し、『アンケートの必須回答項目ではない［年齢］フィールドには、一定数の「null」がある』などのように使います。計算の対象にならないのが一般的で、例えば「10, 20, null」の平均値は「10」ではなく「15」になります。

データがどのように扱われるかを決めるデータ型

データベースにおいては、カラムごとに格納するデータの型を定める必要があります。続いて、主なデータ型について見ていきましょう。

▶文字列型

BigQueryでは「STRING型」と呼ばれます。「山田太郎」「北海道」「Organic」などの文字列は、文字列型としてテーブルに格納されます。

▶整数型

BigQueryでは「INT64型」と呼ばれます。小数点以下を持たない整数としてテーブルに格納されます。

▶浮動小数点型

BigQueryでは「FLOAT64型」と呼ばれます。小数部分を持つ数値としてテーブルに格納されます。

取得できるのが近似値であり、計算においてごくわずかな誤差が発生するデメリットがありますが、結果が返るのが早いというメリットもあります。次ページの［1-1-3］では、BigQuery上に作成したテーブルで［revenue］の値を1.1倍した［revenue_with_tax］を計算していますが、小数点以下13位に「2」があり、わずかな誤差が発生していることが分かります。

▶数値型

BigQueryでは「NUMERIC型」と呼ばれます。最大で38桁の整数部分と、9桁の小数点以下部分を持ち、浮動小数点型では誤差が出てしまう計算を正確に表せます。

▶ブール型

BigQueryでは「BOOL型」と呼ばれます。「TRUE」と「FALSE」（大文字と小文字の区別なし）で表されます。

1-1-3 浮動小数点型の誤差の例

▶日付型

BigQueryでは「DATE型」と呼ばれます。「YYYY-MM-DD」の形式で、年・月・日を表します。例えば「2021-2-5」は「2021年2月5日」となります。

▶日時型

BigQueryでは「DATETIME型」と呼ばれます。「YYYY-MM-DD HH:mm:ss.ddddd」の形式で、年・月・日・時・分・秒・サブ秒を表します。タイムゾーンに依存せず、カレンダーや時計に表示される日時を表現するときに使います。

▶時刻型

BigQueryでは「TIME型」と呼ばれます。「HH:mm:ss.ddddd」の形式で、時・分・秒・サブ秒を表します。タイムゾーンに依存せず、時計に表示される日時を表現するときに使います。

▶タイムスタンプ型

BigQueryでは「TIMESTAMP型」と呼ばれます。「YYYY-MM-DD HH:mm:ss.ddddd タイムゾーン」の形式で、絶対的な時刻を表します。

なお、上記以外にも「バイト型」「配列型」「構造体型」「地理型などのデータ型がありますが、本書では扱いません。

こうしてみると、私がこれまでに扱ってきたデータも、実はデータ型に沿ったものであることが分かりますね。「1から5」といった整数型やタイムスタンプ型のデータは、アンケートツールから出力したデータでよく見かけます。

その通りです。Excelではデータ型を意識することはなく、1つの列に文字列も数値も入りますが、データベースではカラムのデータ型は必須の考え方です。アンケートツールから出力したデータもよく見ると、列ごとにデータ型がそろっていることが分かるはずです。

CHAPTER 2

BigQueryの利用開始

2-1　アカウントの開設
2-2　アカウントの構造とプロジェクトの作成
2-3　料金体系と無料枠
2-4　演習用ファイルの内容とスキーマの定義
2-5　データセットとテーブルの作成
2-6　コンソールの操作
2-7　SQL文の記述作法

本書では、みなさんが十分に「手を動かして」SQLを習得できるように、学習環境としてGoogle BigQueryを利用することを前提とします。BigQueryを無料で使えるアカウントを開設し、著者が用意した4つの演習用ファイルからテーブルを作成したうえで、具体的な利用方法を解説していきます。

SECTION

2-1 アカウントの開設

それでは、実際にBigQuery上にテーブルを作成し、SQLを記述する環境の準備を始めましょう。

普段からGmailやGoogle Workspaceは使っているのですが、BigQueryは一度も見たことがありません。使いこなせるでしょうか？

BigQueryはインストール不要ですし、実質的に無料で利用できます。エンジニア以外の分析者にとって、実はもっとも手が届きやすいSQLの実行環境です。本書に多数掲載しているドリルを通じて、基本的な操作方法は自然と身につくはずですよ。

BigQueryは「データ保持」兼「SQL実行」のための環境

SQLは、手を動かして学習しないと身につかない典型的なスキルです。よって、本書では「SQL文」(データベースへの命令のセット）を実際に記述して実行し、結果を確認するための学習環境として、Googleが提供するデータベースサービスである「BigQuery（ビッグクエリ）」を利用します。

BigQueryは、Googleアナリティクス360（有償版のGoogleアナリティクス）や無償版のGoogleアナリティクス4（SECTION 1-1を参照）からのデータエクスポート先となるほか、BigQueryに蓄積されたデータをGoogleスプレッドシートやGoogleデータポータルから利用することもできます。つまり、Googleが提供する複数のサービスにおける、データ保持・加工のハブとしての役割を持っています。

一方で、Googleのサービスとは無関係な、ユーザーの独自データをアップロードして利用することもできます。「Tableau」「PowerBI」「Qlik Sense（クリックセンス）」「Looker（ルッカー）」といった、主要なBIツールからのデータ接続も可能です。そのため、BigQueryでSQLを記述し、データを直接分析するといった用途以外にも、分析用の中間テーブルをBigQuery上に作成し、そのデータにBIツールから接続するといった使い方も可能になっています。

BigQueryの実体はクラウド上にあるため、PCに何らかのソフトウェアを新たにインストールする必要はありません。**必要なのはインターネットに接続されたPCとWebブラウザーのみ**です。Googleのサービス群との相性を考慮すると、ブラウザーはGoogle Chromeがいいでしょう。

BigQueryの利用料金は基本的には有料で、主にデータ量などに応じた従量課金となっています。しかし、十分な無料枠が用意されており、本書執筆時点、かつ本書での学習に利用する範囲なら、実質無料で利用できます。料金体系と無料枠の詳細は、SECTION 2-3で解説します。

「BigQueryサンドボックス」ならクレジットカードは不要

本書執筆時点において、BigQueryの利用を無料で開始する方法には、次の2つがあります。

1. Google Cloud Platformの無料トライアルを開始する
2. BigQueryサンドボックスを利用する

上記1のGoogle Cloud Platform（GCP）とは、BigQueryを含むGoogleのさまざまなクラウドサービスの集合体で、この無料トライアルの体裁でアカウントを開設するのが通常の方法となります。しかし、初回にクレジットカード情報の登録が必要です。もちろん無料枠の範囲で課金されることはありませんが、若干の煩わしさを感じる人もいるでしょう。

そのため、少しでも手間を省けるという意味で、本書では上記2の方法をおすすめします。BigQueryサンドボックスとは「**BigQueryを無料で試せるオプション**」であり、クレジットカード情報を登録することなくアカウントの開設

が可能です。以下のGoogle Cloudブログの記事では、新規ユーザーや学生を想定した環境であることなどが説明されています。

クエリを無料で試せるBigQueryサンドボックス
https://cloud.google.com/blog/ja/products/gcp/query-without-a-credit-card-introducing-bigquery-sandbox

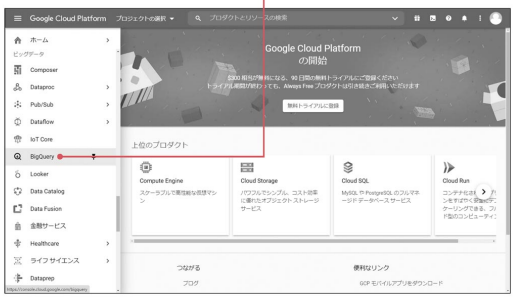
実際にBigQueryサンドボックスのアカウントを開設するには、以下のURLにある「サンドボックス スタートガイド」から操作します。みなさんが普段から使っているGoogleアカウントでサインインしたうえで、以降の手順を参考に操作を進めてください。

BigQueryサンドボックスの使用
https://cloud.google.com/bigquery/docs/sandbox/?hl=ja

2-1-1 BigQueryサンドボックスの利用を開始する

Google Cloud Platform のページが表示された

②左側のメニューから［BigQuery］を選択



BigQuery サンドボックスが利用できるようになった

［プロジェクトを作成］をクリックすると、新しいプロジェクトを作成できる

プロジェクトの作成方法は SECTION 2-2 を参照する

BigQueryサンドボックスでは、とても簡単にBigQueryのアカウントを開設できますが、注意点として**すべてのテーブルに60日間の有効期限がある**ことが挙げられます。本書での学習が60日間を超えた場合、作成したテーブルが警告もなく削除されるため、影響があるかもしれません。

一方、GCPの無料トライアルには、そのような有効期限がありません。BigQueryサンドボックスの画面には、GCPの無料トライアルに移行するためのボタンが表示されているので、有効期限が心配な場合は登録を進めてください。また、学習を進めるうちに「BigQueryをこのまま実務でも使いたい」という気持ちが固まった場合も、GCPの無料トライアルに移行していただくといいでしょう。

意外と簡単に利用を始められるのですね！ とはいえ、まだ何も表示されていませんが……

最初に「プロジェクト」を作成する必要があるので、次節の操作に進みましょう。あと、忘れずにブックマークしておいてくださいね。今後、BigQueryには何度もアクセスすることになります。

SECTION

2-2 アカウントの構造とプロジェクトの作成

BigQueryの画面と操作について見ていきます。まずはアカウントの構造を理解してください。

GoogleアナリティクスやGoogle広告のアカウントは、階層構造になっていますよね。

はい、基本的にはそれらと同じ考え方です。ここでは「プロジェクト」の新規作成についても解説します。

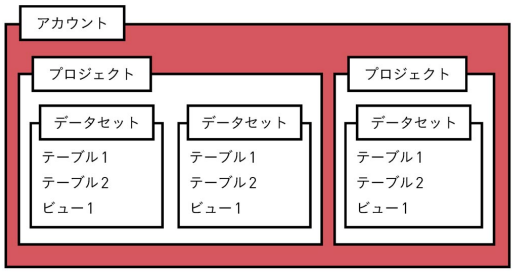
データの関連性を意識して階層を使い分ける

BigQueryのアカウントは、次ページの図［2-2-1］に示すように**「アカウント」「プロジェクト」「データセット」という3つの階層**で構成されています。開設直後のアカウントには、プロジェクトもデータセットも存在しておらず、新しいプロジェクトを作成するところから始めます。

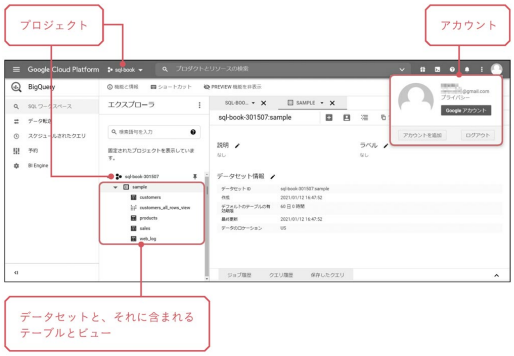
1つのアカウントの中には複数のプロジェクトを作成でき、1つのプロジェクトの中には複数のデータセットを作成できます。データセットの中には、SECTION 1-3で解説したテーブルとビューを複数作成できます。これらの単位で、関連性のあるデータをまとめて保持することにより、目的のデータにたどり着きやすくなる、という仕組みです。

アカウントの開設後、実際にプロジェクトやデータセットを作成した画面は、次ページの［2-2-2］のようになります。Googleアカウントと紐づいたBigQueryのアカウント内に、[sql-book] プロジェクトとデータセット [sample] が格納され、その中に [customers] などのテーブルとビューが存在していることが分かります。

2-2-1 BigQueryのアカウント構造



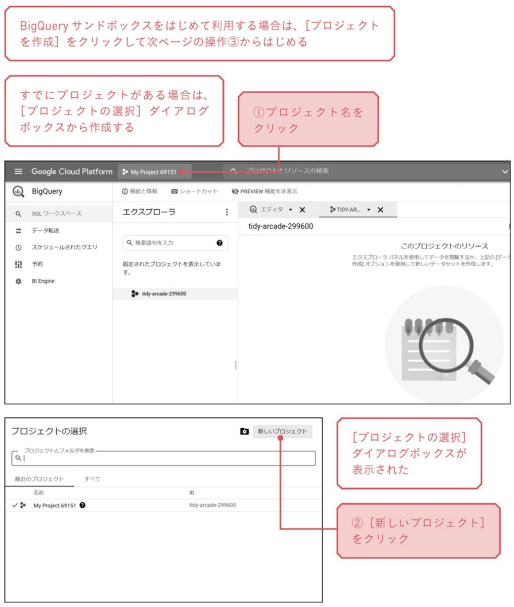
2-2-2 アカウント／プロジェクト／データセットの例



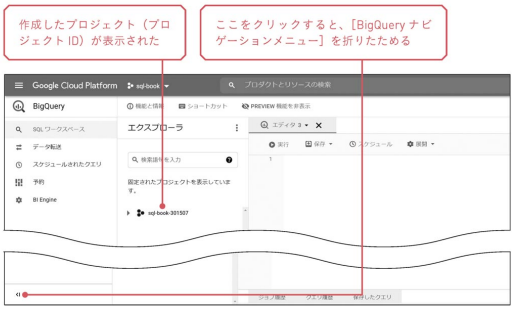
プロジェクトを作成する

本書の演習用ファイルをBigQueryで使えるようにするために、新しいプロジェクトを作成しましょう。以下の［2-2-3］の手順で進めます。

2-2-3 新しいプロジェクトを作成する



Google Cloud Platform
新しいプロジェクト
MANAGE QUOTAS
プロジェクト名 *
sql-book
編集
場所 *
組織なし
参照
作成
キャンセル
③プロジェクト名を入力
ここでは「sql-book」と指定する
④［編集］をクリック
⑤プロジェクト ID を入力
プロジェクト ID は「sql-book-（数字）」などと指定する
［場所］を変更する必要はない
⑥［作成］をクリック
作成したプロジェクトに切り替える
はじめてプロジェクトを作成したときは、この操作は必要ない
⑦プロジェクト名をクリック
BigQuery
プロジェクトの選択
新しいプロジェクト
プロジェクトとフォルダを検索
最近のプロジェクト
すべて
名前
ID
sql-book
sql-book-301507
My Project 69181
tidy-arcade-299600
キャンセル
開く
⑧作成したプロジェクトを選択
⑨［開く］をクリック



［新しいプロジェクト］画面の［プロジェクト名］で使用できる文字は、英数字、シングルクォーテーション、ハイフン、スペース、エクスクラメーションマークです。プロジェクト名としては、**どのようなデータが格納されているのかを端的に表す短い文字列**が適しています。本書での学習のため、ここでは「sql-book」としました。

［編集］をクリックすると入力できる［プロジェクトID］は、BigQueryが内部的にプロジェクトを識別するIDです。アルファベットの小文字、数字、ハイフンのみを入力でき、先頭はアルファベットとし、末尾はハイフンで終わらないようにします。ほかのプロジェクトとは異なる一意のIDである必要があるため、「sql-book-（数字）」などとして、重複しないものを指定してください。また、一度指定したIDは変更できません。

［場所］については［組織なし］のままで大丈夫です。作成後、新しいプロジェクトに切り替えることを忘れないようにしましょう。

通常の用途で、操作のためにコマンドを記述したりすることはありません。Googleのほかのビジネスツールと、操作感は大差ないはずです。

SECTION

2-3 料金体系と無料枠

サンドボックスを卒業して、正規にBigQueryを利用するときに備えて、利用料金について確認しておきましょう。みなさん、気になるところですよね。

SECTION 2-1でおっしゃっていた「実質無料」とは、どういう意味なのでしょうか？

BigQueryには無料枠が用意されており、本書の学習で利用するテーブルやSQL文であれば、その無料枠に収まります。よって、実質無料というわけです。

データ保持とSQL文の実行にかかるデータ量で課金される

BigQueryの料金体系には、いくつものパターンがあります。データの変更頻度や従量・定額、データを格納する物理的な場所（ロケーション）による違いがあるなど、すべてを解説・比較しようとすると、かなり複雑です。

ここでは本書が対象とする読者のみなさんに向け、SQLの学習目的、および業務上のデータ分析を行う用途で理解すべき範囲に絞って解説します。まず、BigQueryの料金は次の2つの合計で決まると考えてください。

- 月ごとのデータ保持にかかる料金　⇒ストレージの料金
- SQL文の実行にかかかる料金　⇒クエリの料金

そして、ロケーションが北米（マルチリージョン）の場合、ストレージとクエリの料金、および無料枠は次のようになります。

●ストレージの料金
⇒1GBあたりUS$0.020（約2.2円）／毎月10GBまで無料
●クエリの料金
⇒1TBあたりUS$5.00円（約540円）／毎月1TBまで無料

みなさんに使っていただく本書の演習用ファイル（SECTION 2-4を参照）をすべてBigQueryにアップロードしても、データ量の合計は約300KBにしかなりません。ストレージの無料枠である10GBのうち0.003%しか利用しないため、ストレージの料金が発生することはありません。

また、演習用ファイルから作成した、もっともデータ量の大きなテーブルに対してSQL文を10,000回実行したとしても約3GB、クエリの無料枠1TBのうちの0.3%です。本書での学習用途なら実質無料と述べたのは、このような根拠に基づいています。

なお、BigQueryの「クエリエディタ」でSQL文を実行するときには、**その処理に必要なデータ量が事前に表示**されます。学習後、業務上のデータ分析を行うときにも役立ちますので、覚えておいてください。また、料金の詳細は以下の公式ヘルプを参照してください。

2-3-1 クエリの実行前にデータ量を確認する

料金（BigQuery）
https://cloud.google.com/bigquery/pricing?hl=ja

SECTION

2-4 演習用ファイルの内容とスキーマの定義

本書では、演習用ファイルとして4つのCSVファイルを用意しました。これらからBigQueryのテーブルを作成する前に、筆者がどのような意図で作成したデータなのかを紹介しておきたいと思います。

私たちがSQLを学ぶためのサンプルになるデータですね。「意図」といいますと……？

多くのビジネスパーソンが実務で扱う可能性が高いであろうデータに似せたサンプルを意図しました。業務と無関係なサンプルでは、臨場感がないですからね。

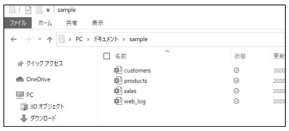
各テーブルの性質やカラム名、データ型を確認する

みなさんに実践的なSQLのスキルを身につけてもらえるように、本書ではSQL文を実行する対象であるテーブルのサンプルデータとして、**CSV形式の演習用ファイル**を用意しました。P.010に掲載している本書のサポートページを参考に、PCにダウンロードしておいてください。

演習用ファイルには、Webサイトを持つ架空の小売業者を想定したデータが記録されており、次の4つのファイルに分かれます。

- customers.csv ⇒顧客データ
- products.csv ⇒商品データ
- sales.csv ⇒販売データ
- web_log.csv ⇒Webログデータ

2-4-1 ダウンロードした演習用ファイル



演習用ファイルをPCにダウンロードし、任意のフォルダーに保存しておく

これらをBigQueryにアップロードし、それぞれからテーブルを作成していくわけですが、**アップロード時にはカラム名（フィールド名）やデータ型（タイプ）といったスキーマを定義**する必要があります。以下の［2-4-2］を含む4つの表で各テーブルの元となるファイル名や性質、アップロード時に指定するカラム名、データ型を示すので、あらかじめ確認しておいてください。

2-4-2 customersテーブルの内容とスキーマ

ファイル名	customers.csv	
性質	顧客マスタ	
想定	顧客情報を格納するテーブル	
行数	497	
カラム名	データ型	説明
user_id	INTEGER	顧客を識別するID（顧客ごとに固有）
name	STRING	顧客の氏名
birthday	DATE	顧客の誕生日
gender	INTEGER	顧客の性別（1：男性、2：女性）
prefecture	STRING	顧客の住所の都道府県名
register_date	DATE	顧客の会員登録日
is_premium	BOOLEAN	プレミアム顧客かどうか？ （TRUE：プレミアム顧客、FALSE：一般顧客）

2-4-3 productsテーブルの内容とスキーマ

ファイル名	products.csv
性質	商品マスタ
想定	商品属性を格納するテーブル
行数	15

カラム名	データ型	説明
product_id	INTEGER	商品を識別するID（商品ごとに固有）
product_name	STRING	商品の名称
product_category	STRING	商品のカテゴリー
cost	INTEGER	商品の仕入れ値（単価）

2-4-4 salesテーブルの内容とスキーマ

ファイル名	sales.csv
性質	販売データ
想定	販売情報を格納するテーブル
行数	1,177

カラム名	データ型	説明
order_id	INTEGER	注文を識別するID（注文ごとに固有）
user_id	INTEGER	顧客を識別するID
product_id	INTEGER	販売された商品を識別するID
date_time	DATETIME	販売が実行された年月日時間
quantity	INTEGER	販売された個数（数量）
revenue	INTEGER	販売された金額
is_proper	BOOLEAN	定価販売だったかどうか？ （TRUE：定価販売、FALSE：値引き販売）

2-4-5 web_logテーブルの内容とスキーマ

ファイル名	web_log.csv
性質	Webアクセスログ
想定	Webページの利用状況ログを格納するテーブル
行数	2,604

カラム名	データ型	説明
cid	STRING	Webブラウザーを識別するID
user_id	INTEGER	顧客を識別するID
device	STRING	Webサイトを利用したデバイス
session_count	INTEGER	Webサイトへの訪問回数を示す整数
media	STRING	Webサイト訪問時に利用したメディア（検索エンジン、SNS経由など）
date_time	DATETIME	ページビューが発生した年月日時間
page	STRING	閲覧されたWebページ（/prod/prod_id=2なら商品ID2の紹介ページ）

僕が関わっているWebサイトではECも展開しているので、どれもなじみのあるデータです。

私はECよりも実店舗のPOSデータを見ることが多いですが、実質的に［sales］テーブルと同じですね。

4つの表で示したカラム名とデータ型は、次節で演習用ファイルをアップロードして、BigQuery上にテーブルを作成するときに必要です。アップロード作業をするときに適宜参照してください。

SECTION

2-5 データセットとテーブルの作成

演習用ファイルの内容を理解できたところで、それらからBigQueryのテーブルを作成していきましょう。事前に新しいデータセットも作成しておきます。

特に気をつけるところはありますか？

テーブル作成時の設定ですね。本書では自動検出に頼らない手動での設定をおすすめしますが、カラム名とデータ型の指定、ヘッダー行のスキップが間違えやすいところです。注意しつつ進めてください。

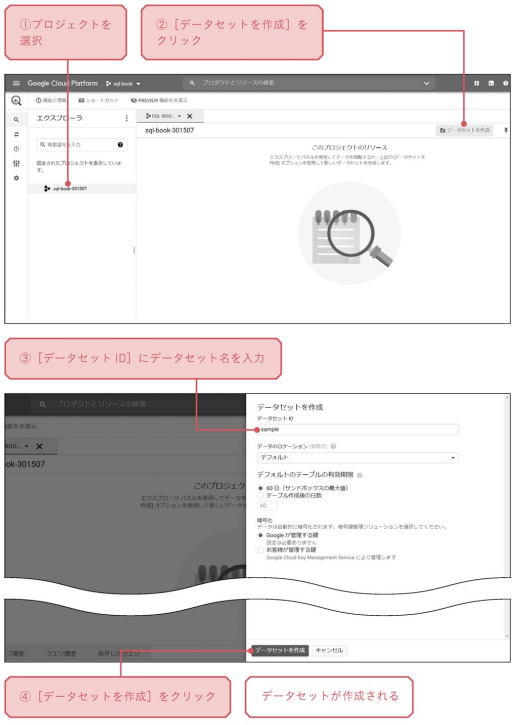
テーブルを格納するデータセットを準備する

演習用ファイルのアップロードに先立ち、プロジェクト内に新しいデータセットを作成しましょう。データセットには複数のテーブル、そしてSQLの実行結果であるビューが格納されます。本書での学習においては、**1つのデータセットに4つのCSVファイルをアップロードし、4つのテーブルを作成**します。手順は次ページの［2-5-1］の通りです。

SECTION 2-2の例では、プロジェクト名を「sql-book」として作成しました。本節で作成するデータセット名は「sample」とし、テーブル名とカラム名は、前節のSECTION 2-4で示した表と同じにしていきます。

いずれも、みなさんが任意の名前を付けても構わないのですが、その場合は以降の解説において、該当するデータセット名やテーブル名を読み替えてもらう必要が生じます。多少面倒だと思いますので、できれば本書で示している名前にそろえることをおすすめします。

2-5-1 データセットを作成する



スキーマを定義してテーブルを作成する

データセットが作成できたら、演習用のCSVファイルをアップロードしてテーブルを作成します。ここでのポイントはスキーマの定義です。

CSVファイルをアップロードする場合、CSVファイルの1行目にカラム名となるべきデータがあり、それらが英数字の場合は、BigQuery上のテーブルのカラム名としてそのまま使うことができます。しかし、本節では勉強も兼ねて、**アップロード時にカラム名をあらためて指定し、同時にデータ型も手動で指定する手順で解説**します。

BigQueryにはカラム名だけでなく、データ型も自動検出してスキーマを定義するオプションがありますが、意図しないカラム名やデータ型でテーブルが作成されるリスクがあります。その場合はテーブルを作り直すことになり、二度手間になりがちです。手数は増えてしまいますが、意図通りのテーブルを作成できる手動での方法を推奨することとしました。

また、スキーマの定義においては、各カラムに対して「モード」も選択することになります。本書の演習用ファイルでは、すべて「NULLABLE」としてください。これは『「null」を許容する』という意味になります。

仮にほかのモードを選択した場合、いずれかのカラムに「null」が存在するとエラーになり、テーブルの作成に失敗してしまいます。演習用ファイルの1つ「customers.csv」でいえば、user_idは顧客ごとに固有のIDであるため「null」は存在しませんが、birthdayは誕生日の未記入を想定した「null」が存在します。カラムごとにモードを選択することの重要性は本書での学習においては低いので、一律でNULLABLEを選択してください。

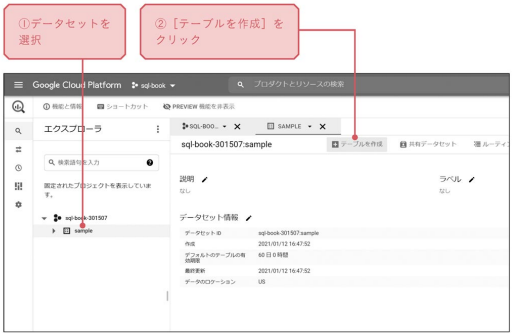
次ページの表[2-5-2]に［customers］テーブルを作成するときのスキーマを示すので、この通りに設定してください。残り3つのテーブルのカラム名とデータ型は、SECTION 2-4で示したとおりです。また、次ページの[2-5-3]でテーブルを作成する一連の手順を紹介します。

4つのCSVファイルで同じ作業を繰り返すことになるので、慌てずに進めてください。

2-5-2 customersテーブルのスキーマ

カラム名	データ型	モード
user_id	INTEGER	NULLABLE
name	STRING	NULLABLE
birthday	DATE	NULLABLE
gender	INTEGER	NULLABLE
prefecture	STRING	NULLABLE
register_date	DATE	NULLABLE
is_premium	BOOLEAN	NULLABLE

2-5-3 テーブルを作成する

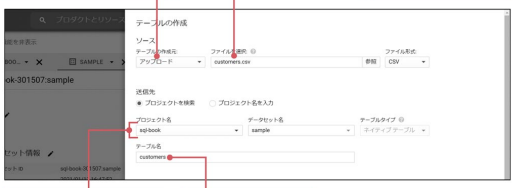


［テーブルの作成］ウィンドウが表示された

アップロードする CSV ファイルの指定やテーブル名、スキーマの設定を行う

③［テーブルの作成元］で［アップロード］を選択

④アップロードする CSV ファイルを選択



⑤［プロジェクト名］と［データセット名］を選択

⑥［テーブル名］を入力

⑦スキーマの［フィールドを追加］をクリック

⑧［名前］［型］［モード］にカラム名、データ型、モードを指定

1 行が 1 つのカラムのスキーマを表している

CSVファイルに含まれるデータのヘッダーをスキップする

⑨［詳細オプション］をクリックして展開

⑩［スキップするヘッダー行］に「1」と入力

⑪［テーブルを作成］をクリック

選択したプロジェクト、データセットの配下にテーブルが作成された

同じ作業を繰り返し、演習用ファイルから4つのテーブルを作成する

テーブル作成時にエラーが表示されたときの対処法

［customers］テーブルに続き、［products］［sales］［web_log］のテーブルを作成できたでしょうか。もしかするとテーブルの作成を繰り返す中で、次ページの［2-5-4］に示した「ジョブの実行エラー」という画面が表示されたかもしれません。このエラーの対処法についても触れておきましょう。

BigQueryにおけるジョブとは、テーブルの作成・削除、SQL文の実行などの処理のことを指します。ジョブの実行中にエラーが発生すると、その旨とエラーの原因が表示され、テーブルの作成などが中止されます。

［2-5-4］の例では、1つ目のメッセージ内に［user_id］を整数型（INT64）として取り込めませんでした。場所は0文字目です」とあります。演習用ファイルの1行目（ヘッダー行）にはカラム名が記載されているので、ヘッダー行まで整数型のデータとして取り込もうとしたために表示されたエラーであると判断できます。ヘッダー行は［2-5-3］の操作⑨〜⑩で示したように、スキップしなければいけません。

ほかにも、データ型の指定が間違っていてもエラーとなります。例えば、［web_log］テーブルの［cid］は数値に見えますが、ドットを含むIDであるため文字列型（STRING）として取り込むのが適切です。SECTION 2-4の表を参照しつつ、正しく指定してください。

ジョブをやり直してテーブルを再作成するには、エラー画面の**［ジョブ履歴を開く］からリトライするのが簡単**です。［テーブル作成］ボタンをクリックする前の状態に戻ることができます。

慣れないうちはエラーに遭遇することもあるかもしれませんが、［ジョブ履歴］を参照し、何が原因かを落ち着いて確認すれば問題ありません。

2-5-4 実行エラーのジョブをやり直す

実行エラーとなったジョブをやり直せる

業務上のCSVファイルをアップロードするときの注意点

本書での学習後には、みなさんが業務において入手したCSVファイルからBigQueryのテーブルを作成したいこともあるでしょう。そのようなシチュエーションでの注意点を補足します。

まず、BigQueryにアップロードするCSVファイルは**「UTF-8」という方式でエンコード**されている必要があります。アップロード時にエラーが表示されるときは、テキストエディタなどで変換・保存してください。

また、本節で解説したようにCSVファイルを直接アップロードできるのは、ファイルサイズが10MB未満、レコード数が16,000行未満の場合に限られます。それ以上の場合は、Google Cloud Storageにいったん CSVファイルをアップロードしたうえで、BigQueryに取り込む手順を踏む必要があります。Cloud Storageの利用には、データ量に応じた料金が必要になるので、以下の公式ヘルプを参照してください。

Cloud Storage
https://cloud.google.com/storage?hl=ja

SECTION

2-6 コンソールの操作

テーブルの準備ができたところで、BigQueryのUIを見ていきましょう。特にマスターしたいのが、SQL文を記述する領域である「クエリエディタ」です。

コンソールを構成する領域を理解する

BigQueryの操作画面は「コンソール」と呼びます。コンソールの主なUIは、次ページの［2-6-1］にも示した次の領域で構成されています。

▶プロジェクト

操作の対象となるプロジェクト名が表示されます。本書での学習においては、演習用のテーブルが存在するプロジェクトが選択されていることを常に確認してください。次ページの［2-6-2］の画面で切り替えできます。

▶クエリエディタ／下部パネル

SQL文を記述したり、その実行結果やジョブの履歴などが表示されたりする領域です。以降であらためて解説します。

▶ナビゲーションパネル

現在選択されているプロジェクトの配下にあるデータセット、テーブル、ビューの一覧が、階層化されて表示されます。

▶BigQueryナビゲーションメニュー

BigQueryにおけるメニューです。本書での学習に関連するのは［SQLワークスペース］のみで、その他の項目については扱いません。

2-6-1 BigQueryのコンソール

2-6-2 ［プロジェクトの選択］ダイアログボックス

クエリエディタにSQL文を記述して実行する

クエリエディタは、SQL文を記述するための機能が集約されている、もっとも使用頻度の高い領域です。まず、はじめてのSQL文として、以下のSQL文[2-6-3]をクエリエディタに記述してみてください。

2-6-3 はじめてのSQL文

```
1  SELECT user_id FROM sample.sales
```

すると、[クエリエディタ]の表記が[未保存のクエリ]に変わり、以下の画面[2-6-4]のようになったと思います。この状態を元に、クエリエディタの各部の働きを解説します。

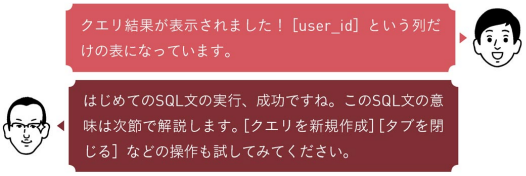
[クエリを新規作成]ボタン(❶)は、SQL文を記述したいときに利用します。クリックすると、新しいタブで別のクエリエディタが表示され、タブを切り替えながらSQL文を記述できるようになっています。

[実行]ボタン(❷)をクリックすると、SQL文が実行されます。キーボードのCtrl+Enterキーでもクエリを実行できます。

2-6-4 クエリエディタ

［タブを閉じる］ボタン（❸）をクリックすると、記述途中、もしくは実行済みのSQL文を削除して、クエリエディタのタブを閉じます。現在のSQL文を消去する旨のメッセージが表示されるので、問題なければ［終了］をクリックして閉じます。なお、後述するビューやテーブルのプレビューなどもタブとして表示されます。

SQL文を実行したときに使用するデータ量は、クエリエディタの右上部分（❹）で確認できます。実際に［2-6-3］のSQL文を実行し、クエリエディタの下に［クエリ結果］が表示されることを確認してください。

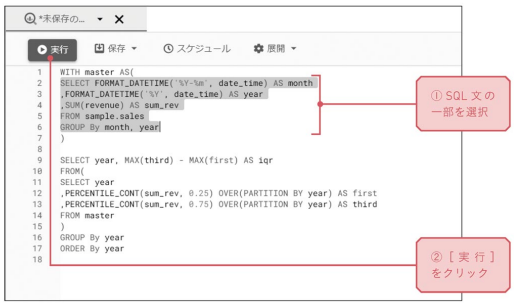


以下の［2-6-5］は、間違ったSQL文を記述した例です。赤色の「！」マークとともに、エラーメッセージ「Syntax error」（構文エラー）と理由が記載されています。この状態では［実行］ボタンをクリックしても、SQL文は実行されません。

2-6-5 間違ったSQL文とエラーメッセージ

なお、CHAPTER 5以降に登場する複雑なSQL文を記述したときなど、SQLの実行結果を部分的に確認したいケースは、実務でもたびたび発生します。そのようなときは、以下の［2-6-6］のように操作してください。選択した範囲のSQL文のみが実行されます。キーボードでは Ctrl +Eで実行できます。

2-6-6 SQL文の一部を実行する



SQL文の実行結果を確認する

SQL文を実行した結果は、次ページの［2-6-7］に示したように、クエリエディタ下部の［クエリ結果］に表示されます。**SQL文の実行結果は表形式のデータ**となり、この表のことを本書では「**結果テーブル**」と呼びます。

BigQueryにおける結果テーブルは、デフォルトでは最大100行が一度に表示されます。この行数（レコード数）は［ページあたりの行数］のプルダウンから変更が可能です。

なお、下部パネルが展開されていると、［クエリ結果］が隠れて見えない場合があります。その場合は前掲の［2-6-1］を参考に、下部パネルを最小化してください。

2-6-7 クエリ結果

クエリ結果には、結果テーブルが表示される［結果］タブ以外にも、複数のタブが存在します。その中でも覚えておきたいのが、次ページの［2-6-8］に示した［ジョブ情報］タブです。

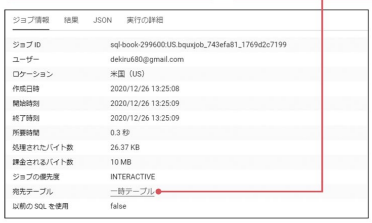
［ジョブ情報］タブには、SQL文の実行作業としてのジョブの詳細が表示されます。この中の［宛先テーブル］にある［一時テーブル］というリンクをクリックすると、結果テーブルに関する情報を確認できます。**一時テーブルはクエリの実行後、最大24時間保持**されます。

例えば、CHAPTER 7〜8に登場する「関数」を利用して、新しいカラムを作成するようなSQL文を実行したとき、結果テーブル上で関数が作成した新しいカラムがどのようなデータ型なのかを確認したいことがよくあります。［一時テーブル］のリンク先にある［スキーマ］タブで、カラムのデータ型（タイプ）を確認できるので、覚えておいてください。具体的には、次ページの［2-6-9］のように表示されます。

2-6-8 クエリ結果のジョブ情報

クエリの作成日時や所要時間など、ジョブの詳細情報を確認できる

［一時テーブル］から結果のテーブルの情報を確認できる



2-6-9 結果テーブルのスキーマ

結果テーブルのスキーマを確認できる

SQL文の実行結果はテーブルとして返ってきて、それは一時テーブルとして一定時間、BigQueryに保持されているということですね。

正解です。さらに、このSQL文や一時テーブルを保存して再利用することもできます。解説を進めましょう。

クエリを保存・利用する

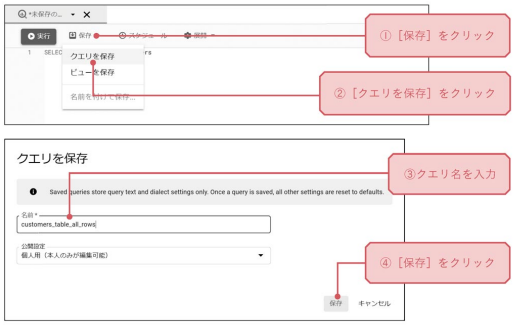
クエリエディタに記述したSQL文は、名前を付けた「クエリ」として保存できます。操作手順を以下の［2-6-10］に示します。

自分ひとりでの分析では、クエリを保存しておく必要性を感じないかもしれません。しかし、他者からデータ分析を請け負い、成果物を納品するような業務では、保存しておいたクエリが後々役立つことがあります。

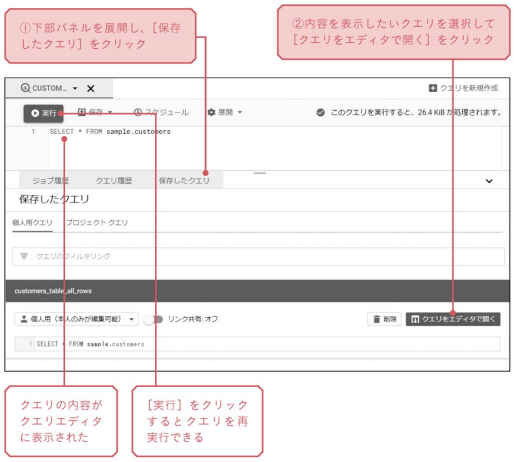
なぜなら、例えば「分析結果に疑問があるので確認したい」「別のプロジェクトでもSQL文を再利用できそうなので納品してほしい」といった要望が、事前にクエリの納品を依頼されていなくても、筆者の経験上、たびたび発生するからです。心当たりのあるみなさんは、多少面倒でもクエリに分かりやすい名前を付けて保存しておくことをおすすめします。

なお、保存したクエリは次ページの［2-6-11］に示すように、下部パネルの［保存したクエリ］から確認できます。クエリの再実行や、SQL文の記述内容の検証に利用しましょう。

2-6-10 クエリを保存する



2-6-11 保存したクエリを表示する



ビューを保存・利用する

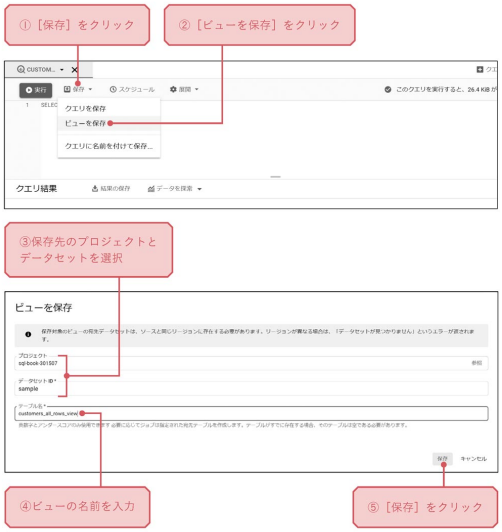
SECTION 1-3で述べたように、ビューとはSQL文の実行結果となる表形式のデータのことを指します。いわば「仮想テーブル」であり、BigQueryには、その仮想テーブルをビューとして保存しておく機能があります。保存方法は次ページの［2-6-12］の通りで、ビューの名前には英数字と「_」（アンダースコア）のみが使えます。

ビューの保存は、実態としてはSQL文の保存です。そのため、保存したビューは別のクエリからも参照でき、使い回すことが可能になります。こうした特徴の活用例はSECTION 5-6で紹介します。

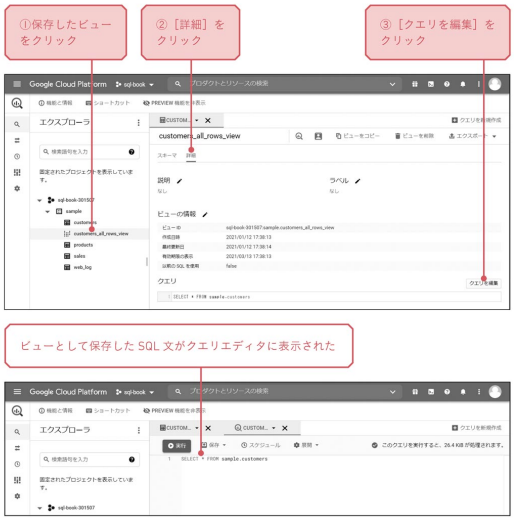
保存したビューは、次ページの［2-6-13］で示すように、ナビゲーションパネルの一覧から呼び出すことができます。既存のテーブルと同じように、データセットの配下に並んでいますが、テーブルとはアイコンが異なるため区別できます。クリックすると、クエリエディタであたかもテーブルのように利用でき、クエリを再編集することも可能です。

なお、記述したSQL文をビューとして保存したいとき、そのSQL文を事前に実行しておく必要はありません。

2-6-12 ビューを保存する



2-6-13 保存したビューを利用する



SQL文の実行結果をテーブルとして保存する

SQL文の実行結果を、新しいテーブルとして保存することも可能です。ビューの保存が仮想テーブル、実態としてはSQL文を保存するのに対し、次ページの［2-6-14］に手順を示した［結果の保存］は、［クエリ結果］に表示された結果テーブルそのものを保存します。

テーブルの保存場所は複数から選択でき、BigQueryに保存する場合、レコード数やファイルサイズの制限はありません。ただし、Googleドライブに保

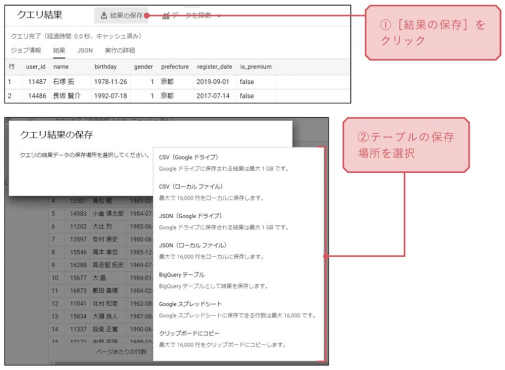
存する場合はサイズが1GBまで、Googleスプレッドシートやローカルドライブ（自分のPC）に保存する場合は16,000レコード、かつサイズが10MBまでという制限があります。

ちなみに、結果テーブルをBigQuery上のテーブルとして保存するメリットは、**目的特化型のテーブルが作れる**ことにあります。

例えば、すべての商品の直近10年間の売上を記録した「売上実績」テーブルがあるとします。商品Aを販売する事業部が2020年の売上を分析したい場合、その売上実績テーブルに対して毎回クエリを実行するのは、あまりにも非効率です。

そこで、売上実績テーブルを「商品A、かつ2020年の販売」にだけ絞り込むSQL文を実行し、結果テーブルをBigQuery上に保存すれば、その事業部の目的に特化したテーブルが完成します。これにより、結果を得るスピードも、BigQueryの費用も抑えられるというわけです。

2-6-14 クエリの結果をテーブルとして保存する

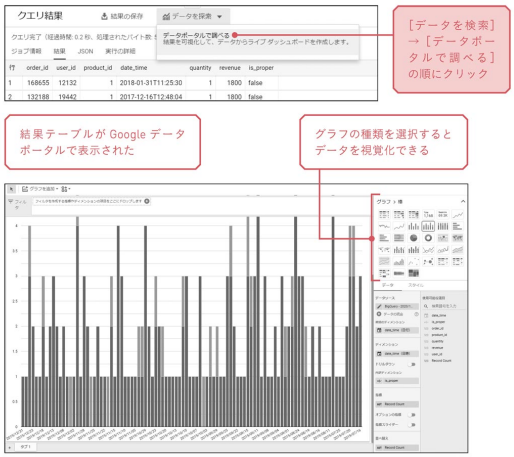


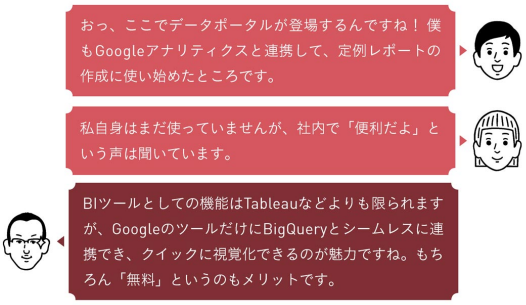
結果テーブルのデータを視覚化する

SQL文の実行結果は、常にテーブルとして出力されます。結果テーブルには貴重なデータが含まれていますが、どこまでいっても表であり、ビジュアルとして分かりやすいものではありません。

BigQueryでの分析結果をグラフとして表現したい、インパクトのある形で伝えたいときには、データ視覚化・ダッシュボード構築を司るサービスである「Googleデータポータル」との連携を試してみてください。以下の［2-3-15］のように操作すると、結果テーブルを即座にデータポータルで開き、グラフとして表現できます。

2-6-15 Googleデータポータルと連携する





クエリやジョブの履歴を確認する

BigQuery上で実行したSQL文の履歴は自動的に記録され、次ページの［2-6-16］に示すように、下部パネルの［クエリ履歴］から表示できます。また、テーブルの作成やコピー、SQL文の実行結果の保存などの処理も記録され、［2-6-17］のように［ジョブ履歴］から参照できます。

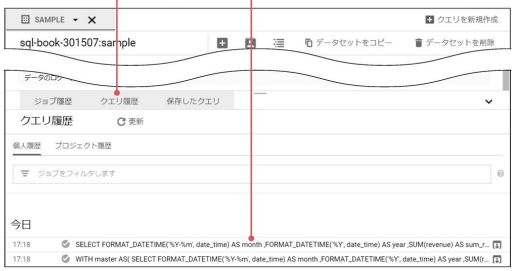
ジョブ履歴の活用例としては、前節のSECTION 2-5でも見たように、CSVファイルからテーブルを作成するときに失敗し、その再作成をすばやく行う用途が考えられます。カラム名に2バイト文字が含まれていた、レコードごとに「,」（カンマ）の位置がずれていた、などのミスはよくありますが、ジョブ履歴から処理をやり直すことで手間を省くことができます。

なお、クエリ履歴とジョブ履歴の保存期間はともに最大6か月で、保存件数は両者の合計で1,000件までです。

2-6-16 クエリ履歴を表示する

［クエリ履歴］をクリック

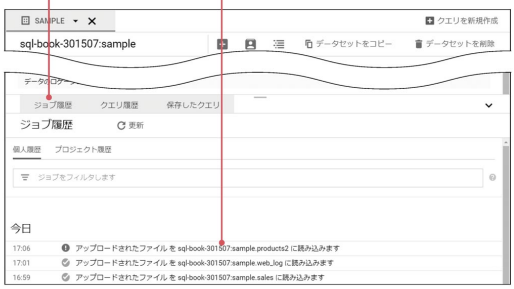
クエリを選択してクエリエディタに表示できる



2-6-17 ジョブ履歴を表示する

［ジョブ履歴］をクリック

ジョブを選択して過去の処理を読み込める



SECTION

2-7 SQL文の記述作法

前節ではBigQueryのUIを学びながら、はじめてのSQL文を記述・実行できました。CHAPTER 2の最後となる本節では、SQL文を記述するための作法として、基本的なルールを理解してください。

プログラミング言語っぽくなってきましたね。

SQLの実践的な使い方は次章以降でたっぷり学ぶので、そのための準備体操だと思ってください。

SQLは7種類のパーツで構成されている

SQLの文法は英語に類似しており、大きく以下の7つのパーツで構成されています。これらはBigQueryのクエリエディタで自動的に色分けされますが、本書では❶のみに色を付ける形で記載していきます。

7つのパーツを実際のSQL文で示すと、次ページの［2-7-1］のようになります。また、以降でそれぞれのルールを個別に解説します。

❶SQLの命令 ⇒「SELECT」「FROM」「WHERE」など
❷SQLの関数・引数 ⇒「COUNT」「SUM」など
❸カラム名 ⇒「user_id」など
❹テーブル名 ⇒「customers」など
❺カラムやテーブルの別名 ⇒「AS」に続けて指定
❻定数や固定文字列 ⇒「東京」「1」など
❼コメント ⇒処理の内容などを記録したメモ

2-7-1 SQLのパーツ

```
1  SELECT
   ❶
2  COUNT (user_id) AS user_number
   ❷     ❸          ❺
3  FROM sample.customers
          ❹
4  WHERE prefecture "東京" AND gender = 1
                          ❻
5  -- AND birthday >= "1990-01-01"
                  ❼
```

SQLの命令の基本と記述ルール

SQLの命令のうち、もっとも基本となるのが「SELECT（セレクト）」と「FROM（フロム）」です。**SELECTはデータを取得するカラムを指定する命令、FROMはテーブルを指定する命令**となります。FROMのあとには、データセット名とテーブル名を「.」（ドット）でつないで記述します。

ここで、前節の［2-6-3］で示したSQL文を再掲しましょう。

```
1  SELECT user_id FROM sample.sales
```

このSQL文を読み解くと、[sample] データセットの [sales] テーブルから、[user_id] フィールドの値を取得せよ」という意味になります。結果テーブルは、そのような表になっていたかと思います。

SQLの命令の記述ルールとしては、次の3つにまとめられます。

- 半角英字で記述する
- 大文字・小文字は混在してもよい
- 後ろに半角スペース、または改行を挿入する

つまり、SELECTであれば「SELECT」「Select」「select」のいずれの記述でも正しく認識されます。本書ではSQLの命令が目立つように、すべて大文字で統一しました。

ちなみに、筆者は大文字・小文字を切り替えるのが面倒なので、普段はSQLの命令も含め、すべて小文字で記述しています。みなさんも好みで構いません。

SQLの関数・引数の記述ルール

SQLには関数という、Excelの関数と同様に機能する命令があります。例えば「MAX関数」は、対象のデータ範囲から最大値を取得します。関数の中には引数を利用するものもあり、例えば「MAX(quantity)」のように、関数に続けて半角カッコ内に記述します。前掲のSQL文［2-7-1］にはCOUNT関数が引数を伴って記述されていました。

記述ルールとしては、SELECTやFROMなどの命令と同様に半角英字で記述し、大文字・小文字の混在は許容されます。本書では基本的に、関数は大文字で、引数は小文字で統一しました。

カラム名やテーブル名の記述ルール

カラム名の記述ルールは単純で、テーブル内のカラム名と一致させるだけです。半角英字で記述し、大文字・小文字の混在は許容されます。

一方、テーブル名は大文字と小文字が厳密に区別されます。また、対象とし

たいプロジェクト名、データセット名、テーブル名を、以下のルールに従って記述する必要があります。

- プロジェクト名、データセット名、テーブル名を「.」(ピリオド)でつなぎ、「`」(バッククォート)で囲んで記述する
- 現在選択中のプロジェクト配下にあるデータセットとテーブルを指定する場合は、プロジェクト名と「`」の省略が可能

例えば、プロジェクト「AAA」、データセット「Bbb」、テーブル「ccc」を対象するには、原則として以下のSQL文[2-7-2]のように記述します。ただし、現在選択しているプロジェクトがAAAである場合、SQL文[2-7-3]のように省略した記述が可能になります。

2-7-2 プロジェクト名を含めてテーブル名を指定する

```
1  SELECT user_id FROM `AAA.Bbb.ccc`
```

2-7-3 プロジェクト名を省略してテーブル名を指定する

```
1  SELECT user_id FROM Bbb.ccc
```

なるほど。テーブル名の指定には注意が必要ですね。

プロジェクト名まで毎回記述するのは手間なので、本書での学習においてはプロジェクト「sql-book」を常に選択しておくようにしてください。

カラムやテーブルの別名の記述ルール

SQLでは、計算結果を出力したカラムに別名を付けたり、参照するテーブルに別名を付けたりすることがあります。まずは「なぜ別名を付けるとよいのか？」を簡単に説明しましょう。

本書の演習用ファイルから作成した［sales］テーブルには［revenue］というカラムがあり、消費税抜きの販売金額のデータが格納されています。このデータを元に消費税のみの金額を計算したい場合、SELECTのあとに計算式を記述すれば、結果テーブルでその金額を取り出すことができます。以下の［2-7-4］は具体的なSQL文と、その結果テーブルです。

2-7-4 カラムの別名を定義しない例

```
1  SELECT revenue * 0.1 FROM sample.sales
```

結果テーブル

行	f0_
1	180.0
2	180.0
3	180.0
4	180.0
5	180.0

取得できた値が何を意味しているのか分かりにくい

正しく実行されましたが、結果テーブルのカラム名が「f0_」となり、何を表すカラムなのかが分かりにくくなっています。そこで出番となるのが、SQLの基本的な命令の1つである「AS（アズ）」です。

ASは取り出したカラムに別名を定義する命令で、後ろに半角スペースを空けて任意の別名を入力します。次ページのSQL文［2-7-5］では、結果テーブルのカラム名に「tax」という別名が定義されます。ASは［2-7-6］のように省略しても問題はありませんが、本書では分かりやすさを優先し、別名を利用するときには必ずASを記述しています。

2-7-5 カラムの別名を定義した例

```
1  SELECT revenue * 0.1 AS tax FROM sample.sales
```

2-7-6 カラムの別名を定義した例（ASを省略）

```
1  SELECT revenue * 0.1 tax FROM sample.sales
```

結果テーブル

行	tax
1	180.0
2	180.0
3	180.0
4	180.0
5	180.0

［tax］と別名が付与され、値の意味が分かりやすくなった

消費税のみの金額が取り出されたことが分かりやすくなりました。このようにASで別名を定義すると、結果テーブルのデータが意味する内容を分かりやすくできるメリットがあります。

本題の記述ルールとしては、カラムやテーブルの別名には半角英数字が利用でき、大文字・小文字を混在させることが可能です。ただし、数字から始まる別名は付けられないので、その点のみ注意してください。本書ではカラム名、テーブル名、カラムやテーブルの別名は小文字で統一しました。

クエリエディタに記述するお作法が分かってきました。

次章から、基本的なSQLの構文を学んでいきますが、その準備が整いましたね。

定数や固定文字列の記述ルール

『数量が「100」より大きい』『ユーザーIDが「12345」である』『都道府県が「東京」である』など、定数や固定文字列をSQL文の条件式に組み込みたいことがあります。これらは次の記述ルールに従います。

- 数値は半角で記述する
- 固定文字列は「'」(シングルクォーテーション)または「"」(ダブルクォーテーション)で囲んで記述する

具体的には以下のような記述となります。

```
quantity > 100

user_id = 12345

prefecture = "東京"
```

コメントの記述ルール

「SQL文のこの部分では何をしているのか？」を、SQL文自体に記述しておきたいことがあります。そのようなときに利用するのがコメントです。SQL文の実行上は完全に無視され、影響を与えません。

本書で学ぶSQL文は長くても20〜30行程度で、みなさんが実務上で記述するSQL文もおおよそ同程度になると思います。一方、エンジニアが記述するSQLは、何百〜何千行にもなる場合があります。より複雑なSQL文であるほど、コメントを残すことの重要性が高まるでしょう。

コメントの記述には、コメントであることを示す記号を使うのがルールです。1行の場合は「--」(2つのハイフン)または「#」(1つのシャープ)に続けて記述し、複数行の場合は「/」(スラッシュ)と「*」(アスタリスク)を組み合わせた「/*」と「*/」で囲みます。コメントの内容には、漢字やひらがななどの日本語も利用できます。具体例は次ページの［2-7-7］の通りです。

2-7-7 コメントの例

1行のコメントは「--」または「#」に続けて記述する

```
1  # コメントの例
2  SELECT order_id
3  FROM sample.sales
4  WHERE quantity > 3 -- 数量が3より大きい条件での絞り込み
5  /*revenueでの絞り込みは今はしていない
6  AND revenue > 10000
7  */
```

複数行のコメントは「/*」と「*/」で囲む

覚えることがたくさんありましたね……！ でも、実際にやっていくうちに自然と身につきそうです。

はい、SQLはとにかく「自ら実践する」ことが上達の近道です。本書を目で追うだけなく、BigQueryのクエリエディタにどんどん入力して試してみてください。

STEP UP

SQL文の末尾に「;」は必要？

一般にSQL文の末尾には、終わりを意味する「;」(セミコロン）を記述するのがルールとなっています。ただし、単一のSQL文では不要です。SQL文の末尾に「;」が必要となるのは、1つのクエリに2つ以上のSQL文を記述して、実行するケースです。

本書では1つのクエリに1つのSQL文しか記述していませんので、一貫して不要としていますが、もし「;」が記述されたSQL文を見かけたときは、そのように理解してください。

2-7-8 末尾に「;」が必要になる例

1つのクエリに2つ以上のSQL文を記述する場合、末尾に「;」が必要になる

```
1  SELECT quantity FROM sample.sales;
2  SELECT revenue FROM sample.sales
```

CHAPTER 3

SQLの基本構文

3-1　列の取得と並べ替え
3-2　列の作成・演算と別名の付与
3-3　行の絞り込み
3-4　確認ドリル

本章では、SQLの基本的な命令である「SELECT」「FROM」「AS」「LIMIT」「WHERE」「ORDER BY」などについて学びます。SQLを初めて使うみなさんにとって、データをすばやく取り出せる喜びを感じてもらえるはずです。SQLによるデータ分析の土台になる知識でもあるので、しっかりマスターしてください。

SECTION

3-1 列の取得と並べ替え

さぁ、ここからが面白くなってきますよ。実践的なSQLの基本構文について学んでいきましょう。

自分でSQL文を実行して、ちゃんと結果が返ってきたので自信がつきました。続きが気になります！

指定したフィールドの全行を取得する

まずはBigQuery上のテーブルに対し、フィールドを1つだけ指定して、値を全行取得してみましょう。次ページのSQL文［3-1-1］を見てください。これはSECTION 2-6で、はじめてのSQL文として紹介したものと本質的には同じですが、基本中の基本ということで、おさらいです。

SELECTで取得するフィールド名を指定し、FROMで対象となるテーブル名を指定します。FROMの後ろは「データセット名.テーブル名」とするか、「`プロジェクト名.データセット名.テーブル名`」とするのは、SECTION 2-7で学んだ通りです。この結果テーブルは、［sales］テーブルの［order_id］フィールドの値を全行取得したデータとなります。

「フィールド」という言葉がでてきましたが、「カラム」のことですか？

はい。前述の通り、本書ではテーブルの構造としての列はカラム、値を取り出す対象としての列はフィールドと使い分けますが、ほぼ同じ意味と考えて大丈夫です。

3-1-1 単一のフィールドの値を全行取得する

```
1  SELECT order_id FROM sample.sales
```

このSQL文を少し発展させ、**複数のフィールドの値を全行取得**してみましょう。SELECTで指定するフィールド名を「,」(カンマ) で区切って記述します。

例えば、[sales] テーブルから [order_id] [user_id] [quantity] という3つのフィールドを取得するには、以下の［3-1-2］のように記述します。みなさんも実行し、結果テーブルを確認してください。SELECT句で指定した通りの順序でフィールドが取得できています。

3-1-2 複数のフィールドの値を全行取得する

```
1  SELECT order_id, user_id, quantity FROM sample.sales
```

結果テーブル

行	order_id	user_id	quantity
1	168655	12132	1
2	132188	19442	1
3	196737	16801	1

[sales] テーブルから [order_id] [user_id] [quantity] を取得できた

SELECTがフィールドを取得する命令ということは分かりましたが、「SELECT句」とは何でしょうか？

SELECTという命令語と、それに続く [order_id] などの取得したいフィールド名の記述をあわせて「SELECT句」と呼びます。このあとに学ぶ「ORDER BY」も単体では命令語で、「句」が付けばその後ろに来る条件を含むと考えてください。

取得したフィールドを並べ替える

SQL文の結果テーブルにおいて、行が並ぶ順序には規則性がありません。前掲のSQL文［3-1-2］の結果テーブルで［order_id］フィールドに注目すると、1行目から順に「168655」「132188」「196737」と、まるで規則性がない状態で並んでいることが分かります。

そこで、［order_id］を基準とし、その値が小さい順に行を並べ替えたいとします。**行の並べ替えに使うSQLの命令は「ORDER BY（オーダー バイ）」**です。FROMの後に記述し、ORDER BYに続けて並べ替えの基準とするフィールド名を記述します。具体的には、以下のSQL文［3-1-3］となります。

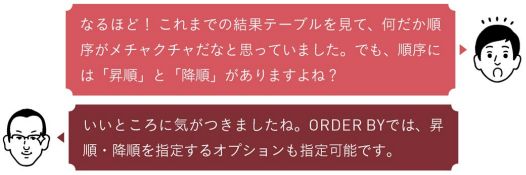
3-1-3 並べ替えの基準となるフィールドを指定する

```
1  SELECT order_id, user_id, quantity
2  FROM sample.sales
3  ORDER BY order_id
```

結果テーブル

行	order_id	user_id	quantity
1	100393	13576	1
2	100393	13576	1
3	100407	15686	1
4	100532	16707	1
5	100749	17469	1
6	100805	10448	1

［order_id］の昇順で並べ替えられた



ORDER BYで並べ替えの基準とするフィールド名を指定した後、「ASC（アスク）」「DESC（デスク）」で**順序のオプションを指定**できます。ASCは「ascending」の略で昇順、DESCは「descending」の略で降順を意味します。このうち、ASCは省略が可能です。

SQL文［3-1-3］を実行すると、結果テーブルは［order_id］を基準とした昇順で行が並びますが、これはASCが省略されているためです。ORDER BY句でASCを明示的に指定すると、以下の［3-1-4］となります。降順で並べたいときは、このASCをDESCに書き換え、降順とすることを明示的に記述する必要があります。

3-1-4 並べ替えの順序を指定する（昇順を明示）

```
1  SELECT order_id, user_id, quantity
2  FROM sample.sales
3  ORDER BY order_id ASC
```

さらに、複数のフィールドを基準とし、異なる順序で並べ替えたいケースを考えてみましょう。ORDER BYに続けて、フィールド名とオプションのペアを「,」で区切って記述します。

次ページのSQL文［3-1-5］では、［user_id］フィールドを基準とした昇順、かつ［order_id］フィールドを基準とした降順で行を並べ替えます。結果テーブルで確認できる通り、［user_id］が同一の場合、次に［order_id］を基準に並べ替えを実行できます。この場合も、昇順を指定するオプションであるASCは省略可能です。

3-1-5 並べ替えの基準・順序を複数指定する

```
1  SELECT user_id, order_id, quantity
2  FROM sample.sales
3  ORDER BY user_id ASC, order_id DESC
```

結果テーブル

行	user_id	order_id	quantity
1	10059	101345	3
2	10059	101345	1
3	10059	101345	1
4	10060	173619	1
5	10060	173619	1
6	10070	184065	1
7	10161	175149	1
8	10161	175149	1
9	10182	114257	1
10	10187	100840	1
11	10206	178515	1
12	10214	117454	1
13	10233	163441	1

[user_id] を昇順、[order_id] を降順に並べ替えられた

なお、ORDERBYで基準として指定するフィールドは、フィールド名ではなく、左端の列を「1」、左から2列目を「2」のように、テーブルの列番号で指定することも可能です。以下のSQL文は前掲の［3-1-5］を書き換えたもので、同時にASCも省略しています。同じ結果が表示されることを確認しましょう。

3-1-6 並べ替えの基準を列番号で指定する

```
1  SELECT user_id, order_id, quantity
2  FROM sample.sales
3  ORDER BY 1, 2 DESC
```

すべてのフィールドを取得する

フィールドの取得の特殊例として、元のテーブルにあるすべてのフィールドの値を取得する構文があります。各フィールドにどのような値が格納されているのかを、ざっと確認したいときに便利です。

具体的には、SELECTで指定するフィールド名の代わりに「*」(アスタリスク)を記述します。例えば、以下のSQL文［3-1-7］では［prodocts］テーブルのすべてのフィールドの値を全行取得します。もちろん、ORDER BYを利用した並べ替えも可能です。

ただし、特定のフィールドを指定する場合に比べて、**SQL文の実行によって処理されるデータ量が増える**ことには注意が必要です。本書の演習用テーブルでは問題になりませんが、レコード数が非常に多いテーブルに対して不用意に実行すると、BigQueryの料金として予想外の金額が請求される可能性があります。実務上は、対象となるテーブルのレコード数がそれほど多くないと分かっている場合のみ、利用してください。

3-1-7 すべてのフィールドの値を全行取得する

```
1  SELECT *
2  FROM sample.products
```

結果テーブル

行	product_id	product_name	product_category	cost
1	1	レアチーズケーキ	ケーキ	400
2	2	ベイクドチーズケーキ	ケーキ	500
3	3	ショートケーキ	ケーキ	600
4	4	モンブラン	ケーキ	500
5	5	ガトーショコラ	ケーキ	800
6	6	ケーキ詰め合わせ	ケーキ	1000
7	7	オレンジゼリー	ゼリー	300
8	8	グレープゼリー	ゼリー	300
9	9	洋ナシゼリー	ゼリー	300
10	10	ゼリー詰め合わせ	ゼリー	1000

すべてのフィールドの値を全行取得できた

不要なフィールドを除いて取得する

すべてのフィールドの取得には「*」を指定しましたが、すべてのフィールドから特定のフィールドだけを除いて取得したいこともあります。そのような場合は「EXCEPT（エクセプト）」という命令を使いましょう。

以下のSQL文［3-1-8］は、［products］テーブルを対象にすべてのフィールドの値を取得するが、［cost］フィールドのみ除外する、という内容です。「*」に続けてEXCEPTと、取得したくないフィールド名を「()」（半角カッコ）で囲んで記述します。

さらに、単一のフィールドではなく複数のフィールドを除外したい場合は、「()」内のフィールド名を「,」でつないで記述します。［3-1-9］のSQL文は、［product_id］と［cost］の2つのフィールドを除外する、という意味になります。

3-1-8 単一の不要なフィールドを除外する

```
1  SELECT * EXCEPT (cost)
2  FROM sample.products
```

3-1-9 複数の不要なフィールドを除外する

```
1  SELECT * EXCEPT (product_id, cost)
2  FROM sample.products
```

SECTION

3-2 列の作成・演算と別名の付与

SQLには、元のテーブルにはない列を結果テーブル上に作成したり、四則計算を行ったりする機能もあります。新しく作成した列に名前を付けることも可能です。

どのようなケースで使うのでしょうか？

例えば、元のテーブルには税抜きの販売金額しかないが、税込みの金額を求めるようなケースです。本節で詳しく見ていきましょう。

新しいフィールドを作成して定数を格納する

SQLでは、元のテーブルに存在しない新しいフィールドを作成することもできます。利用する機会は少ないかもしれませんが、SELECTが持つ機能の1つとして覚えておきましょう。ここでは定数、固定文字列、固定日付の値が格納されたフィールドを作成する例を紹介します。

次ページのSQL文［3-2-1］をクエリエディタに記述し、実行してみてください。1行目ではSELECTに続いて数値の「100」、2行目では文字列の「東京」、3行目では日付の「2020-01-01」を記述しています。SECTION 2-7でも述べたように、文字列や日付の値は「'」（シングルクォーテーション）または「"」（ダブルクォーテーション）で囲みます。

いずれも対象の［sales］テーブルには存在しないフィールドなので、一見、不思議に思えるかもしれませんが、このSQL文は成立します。次ページの結果テーブルのように、指定した数値、文字列、日付が新しいフィールドに格納されたデータが返されることを確認できるはずです。

3-2-1 新しいフィールドに数値・文字列・日付を格納する

```
1  SELECT 100
2  , "東京"
3  , "2020-01-01"
4  FROM sample.sales
```

結果テーブル

行	f0_	f1_	f2_
1	100	東京	2020-01-01
2	100	東京	2020-01-01
3	100	東京	2020-01-01
4	100	東京	2020-01-01
5	100	東京	2020-01-01
6	100	東京	2020-01-01

新しいフィールドが作成され、すべてのレコードに指定した値が格納された

既存の値と定数の演算結果を新しいフィールドに格納する

続いて、取得したフィールドに対して**四則計算などの演算を行い、その結果を新しいフィールドに格納する**方法を解説しましょう。これには大きく分けて、次の2つのケースが考えられます。

1. 既存のフィールドの値と定数の演算結果を求める
2. 既存のフィールド同士の演算結果を求める

まずは上記1のケースです。演習用の［sales］テーブルには、販売金額が格納された［revenue］フィールドがあります。その取得と同時に、消費税込みの販売金額を結果テーブルに表示したいとします。

次ページのSQL文［3-2-2］を見てください。SELECTに続き、まずは注文IDとして［order_id］フィールドを取得します。その後に「revenue * 1.1」と記述することで、［revenue］フィールドを1.1倍した値、つまり消費税込みの販売金額を格納する新しいフィールドを作成します。結果テーブルはSQL文の下に記載した通りとなります。

なお、[revenue] フィールドを1.1倍した値を見ると、小数点以下第13位に「2」の誤差が含まれています。これは演算結果のフィールドが浮動小数点型であることが原因です（SECTION 1-2を参照）。

誤差を解消するにはデータ型を指定するか、切り捨てなどで数値を丸める処理を行う必要がありますが、ここではひとまず「フィールドの値は演算できる」ということを理解してください。

3-2-2 フィールドの値と定数の演算結果を格納する

```
1  SELECT order_id, revenue * 1.1
2  FROM sample.sales
```

結果テーブル

行	order_id	f0_
1	168655	1980.0000000000002
2	132188	1980.0000000000002
3	196737	1980.0000000000002
4	116176	1980.0000000000002
5	101389	1980.0000000000002
6	155449	1980.0000000000002
7	196927	1980.0000000000002
8	142363	1980.0000000000002
9	105131	1980.0000000000002
10	136221	1980.0000000000002

[revenue] を 1.1 倍した値が新しいフィールドに格納された

小数点以下第 13 位の「2」は誤差を表している

なるほど！ たったこれだけのSQL文で、目的の計算結果が入った表をすばやく取得できるわけですね。

はい。SQLでの四則計算はExcelと同じ演算子で実行できるので、すぐに理解できるはずです。

SQLにおけるフィールドの四則計算は、以下の表［3-2-3］に示した演算子を使って記述します。また、計算の優先順位の指定も可能です。

その下のSQL文［3-2-4］では、「(quantity + 2) * 3」という計算式を記述しており、「()」で囲んだ計算が優先されます。結果、［quantity］フィールドに2を足した値を3倍した値が、新しいフィールドに格納されます。

3-2-3 BigQueryで利用できる演算子と記述例

種類	演算子	記述例
和	+	quantity + 1
差	-	quantity - 1
積	*	revenue * 1.1
除	/	revenue / 2

3-2-4 演算の優先順位を指定する

```
1  SELECT (quantity + 2) * 3
2  FROM sample.sales
```

結果テーブル

行	f0_
1	9
2	9
3	9
4	9
5	9
6	9
7	9
8	9
9	9
10	9

［quantity］に 2 を足した値を 3 倍した値が取得できた

既存の値同士の演算結果を新しいフィールドに格納する

今度は、前掲の2のケースです。既存のフィールドに格納されている値同士を演算する方法も覚えておきましょう。同じく［sales］テーブルを対象とします。

例えば、［revenue］（販売金額）フィールドと［quantity］（販売個数）フィールドから、商品単価を求めたいとします。この場合の計算式は［revenue］÷［quantity］となり、以下のSQL文［3-2-5］の4行目にある「revenue / quantity」が該当する記述です。

結果テーブルは記載の通りとなり、演算結果、つまり商品単価が新しいフィールドとして表示されました。レコードの順序は5行目のORDER BYで指定している通り、［quantity］フィールドの降順で並んでいます。

3-2-5 フィールドの値同士の演算結果を格納する

```
1  SELECT order_id
2  , revenue
3  , quantity
4  , revenue / quantity
5  FROM sample.sales ORDER BY quantity DESC
```

結果テーブル

行	order_id	revenue	quantity	f0_
1	184864	8000	4	2000.0
2	115355	8000	4	2000.0
3	125185	8000	4	2000.0
4	107827	8000	4	2000.0
5	129303	8000	4	2000.0
6	188562	8800	4	2200.0
7	192249	9600	4	2400.0
8	110849	6400	4	1600.0
9	121169	12000	4	3000.0
10	118856	5400	3	1800.0

［revenue］÷［quantity］の計算結果が新しいフィールドに格納された

フィールドに別名を付ける

前掲の［3-2-5］の結果テーブルでは、[revenue] ÷ [quantity] の値を格納したフィールドの名前が「f0_」となっています。このままでは、どのような値が格納されているか分かりにくい状態です。そこで利用するのが、フィールドに別名を付与する命令であるASです。

[revenue] ÷ [quantity] が意味する値は商品単価なので、「unit_price」という別名を付けてみましょう。SQL文は以下の［3-2-6］となり、4行目にあるAS句で別名を指定しています。結果テーブルにそれが反映され、新しいフィールドが何を意味しているかが分かりやすくなりました。

3-2-6 新しいフィールドに別名を付ける

```
1  SELECT order_id
2  , revenue
3  , quantity
4  , revenue / quantity AS unit_price
5  FROM sample.sales ORDER BY quantity DESC
```

結果テーブル

ff	order_id	revenue	quantity	unit_price
1	184864	8000	4	2000.0
2	115355	8000	4	2000.0
3	125185	8000	4	2000.0
4	107827	8000	4	2000.0
5	129303	8000	4	2000.0
6	188562	8800	4	2200.0
7	192249	9600	4	2400.0
8	110849	6400	4	1600.0

［revenue］÷［quantity］の計算結果を格納するフィールドに「unit_price」と別名を付けた

このASは、既存のフィールドに別名を付けたいときにも利用できます。次ページのSQL文［3-2-7］では、[sales] テーブルから [order_id] と [revenue] のフィールドを取得し、[revenue] に「uriage」（売上）という別名を付けています。

3-2-7 既存のフィールドに別名を付ける

```
1  SELECT order_id, revenue AS uriage
2  FROM sample.sales
```

なお、SECTION 2-7でも少し触れましたが、ASの記述を省略し、フィールド名の後ろに半角スペースを1つ挿入して別名を指定することもできます。以下のSQL文［3-2-8］のようになりますが、慣れないうちはASを省略するとSQL文が分かりにくくなるため、省略せずにASを記述することをおすすめします。

3-2-8 ASを省略して別名を指定する

```
1  SELECT order_id, revenue uriage
2  FROM sample.sales
```

本節までで、SQLの基本構文についての前半を学習したことになります。お疲れさまでした。実際に記述してみて、SQLが「データベースからデータを取得する言語」であることを実感できたのではないでしょうか。

SECTION

3-3 行の絞り込み

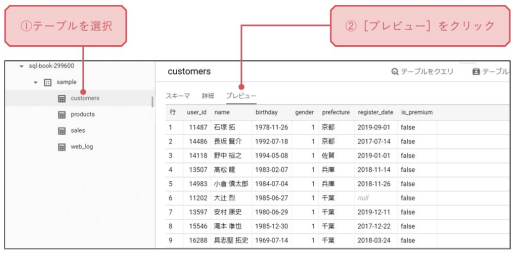
続いて学ぶ基本構文は、結果テーブルの行数を制限する「LIMIT」と、条件を付与する「WHERE」です。条件を指定する記号や演算子も含めて見ていきます。

分析のために必要な行＝レコードだけに、実行結果のデータを絞り込むわけですね。

結果テーブルの表示行数を制限する

BigQuery上にあるテーブルに格納されているデータを確認したいとき、わざわざSQL文を記述する必要はありません。以下の［3-3-1］のようにプレビュー機能を利用すれば、すぐにテーブルの内容を表示できます。

3-3-1 テーブルのプレビューを表示する



しかし、プレビュー機能はテーブルのありのままの姿を確認するために存在するため、次に挙げるような場合には利用できません。

- 演算結果を新しいフィールドに格納した場合
  - ⇒例：[revenue] を1.1倍した値を新しいフィールドに格納したが、正しい値が取得できているかを確認したい
- 条件を指定してデータを絞り込んだ場合
  - ⇒例：[gender] フィールドが男性に一致するレコードを取得するSQL文を記述したが、結果が適切かどうかを確認したい
- 2つ以上のテーブルを参照する場合
  - ⇒例：男女別の販売金額を比較するために［sales］と［customers］の2つのテーブルを結合したが、結果が適切かどうかを確認したい

上記のような場合で、結果テーブルの一部だけを確認したいときは、**SQL文の実行結果を特定のレコード数に制限する**命令である「LIMIT（リミット）」を利用します。以下の［3-3-2］は、[sales] テーブルを対象にレコード数を「5」に制限して表示するSQL文です。

3-3-2 5レコードのみを取得する

```
1  SELECT *
2  FROM sample.sales
3  LIMIT 5
```

結果テーブル

行	order_id	user_id	product_id	date_time	quantity	revenue	is_proper
1	168655	12132	1	2018-01-31T11:25:30	1	1800	false
2	132188	19442	1	2017-12-16T12:48:04	1	1800	false
3	196737	16801	1	2018-01-27T18:30:11	1	1800	false
4	116176	16252	1	2018-04-23T13:56:31	1	1800	false
5	101389	13911	1	2019-08-31T18:28:24	1	1800	false

レコードを5件に絞り込んだ

ただし、単に「LIMIT 5」と記述しただけでは、どの5件を表示するのかがBigQuery任せとなり、表示されるレコードを特定できません。SECTION 3-1で学んだORDER BYも併用し、フィールドの順序を指定したうえでレコードを絞り込みましょう。

例えば、[order_id]と[date_time]の2つのフィールドを、注文が入った日時(date_time)の新しい順(DESC)の上位5レコードだけ取得するSQL文は、以下の[3-3-3]のようになります。

3-3-3 順序を指定して5レコードを取得する

```
1  SELECT order_id, date_time
2  FROM sample.sales
3  ORDER BY date_time DESC
4  LIMIT 5
```

結果テーブル

行	order_id	date_time
1	101469	2019-12-31T13:26:25
2	163388	2019-12-30T23:30:54
3	127978	2019-12-28T12:57:45
4	127978	2019-12-28T12:57:45
5	158635	2019-12-28T11:14:07

[date_time]が新しい順の上位5レコードに絞り込んだ

なお、利用頻度は高くありませんが、LIMITには「OFFSET(オフセット)」というオプションも利用できます。OFFSETは「ずらす」という意味で、指定したレコード数分だけずらして結果を取得することが可能です。

例えば、[3-3-3]の4行目の「LIMIT 5」の後ろに「OFFSET 100」と追記して次ページのSQL文[3-3-4]にすると、日付が新しい上位1〜5番目のレコードではなく、上位101〜105番目のレコードを取得できます。

3-3-4 101〜105番目のレコードを取得する

```
1  SELECT order_id, date_time
2  FROM sample.sales
3  ORDER BY date_time DESC
4  LIMIT 5 OFFSET 100
```

結果テーブル

行	order_id	date_time
1	125401	2019-09-29T23:51:56
2	133789	2019-09-29T15:25:22
3	133962	2019-09-27T14:34:40
4	138552	2019-09-26T13:25:40
5	129768	2019-09-25T21:06:53

OFFSETで指定したレコード数分ずらして5レコードが表示された

条件に合致するレコードのみを取得する

LIMITが「結果テーブルの表示行数の絞り込み」だったのに対し、**条件を明示的に指定して合致するレコードだけを取得**する命令が「WHERE（ホエア）」です。WHEREはFROMの直後に記述します。

例えば、[customers]テーブルから女性客のレコードを取得するには、以下の［3-3-5］のように記述します。演習用テーブルでは［gender］フィールドの値が「2」＝女性客なので、WHEREで指定する条件式は「gender = 2」となります。

3-3-5 女性客のレコードのみを取得する

```
1  SELECT *
2  FROM sample.customers
3  WHERE gender = 2
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	14012	王 美和	1994-04-01	2	三重	2018-12-28	false
2	12465	土井 佳実子	1978-10-17	2	京都	2018-09-27	false
3	12485	有村 智恵	1995-09-24	2	京都	2018-08-31	false
4	14328	野田 智絵	1978-01-23	2	京都	2018-12-05	false
5	14918	芳賀 千穂	1976-03-21	2	京都	2019-03-14	false
6	16616	土屋 智代	1977-05-16	2	京都	2018-08-15	false
7	10462	宮川 愛	1996-05-17	2	佐賀	*null*	false
8	12660	竹田 のどか	1993-01-16	2	佐賀	2019-09-07	false
9	14044	太田 美由紀	1999-01-22	2	佐賀	2018-11-04	false

女性客（gender = 2）のレコードのみに絞り込んだ

WHEREはLIMITやORDER BYと組み合わせることもできますが、記述する順序に注意が必要です。この3つの命令を同時に記述する場合、SELECTとFROMに続いて**WHERE、ORDER BY、LIMITの順に記述**します。

以下の［3-3-6］は、［customers］テーブルから女性客のみを、登録日（register_date）が新しい順に5レコードをだけ表示するSQL文です。上記の順序で記述されていることが分かると思います。

3-3-6 女性客の登録日が新しい順に5レコードを取得する

```
1  SELECT *
2  FROM sample.customers
3  WHERE gender = 2
4  ORDER BY register_date DESC
5  LIMIT 5
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	13672	服部 早苗	1995-10-28	2	滋賀	2019-12-23	false
2	13434	馬場 茉莉恵	1997-05-22	2	大阪	2019-12-20	false
3	13420	柏木 有花	1994-05-12	2	長野	2019-12-18	false
4	10187	徳弘 美憂	1991-06-15	2	福岡	2019-12-18	false
5	15681	福永 静香	1991-01-24	2	沖縄	2019-12-13	false

女性客（gender = 2）に絞り込んだうえで登録日が新しい順の5件が表示された

条件式に利用できる記号

WHEREに続けて記述する条件式に利用できる「記号」は、以下の表[3-3-7]にまとめた通りです。また、SQL分に記述する際は、表の下に記載した点に注意してください。

3-3-7 条件式における記号の意味と記述例

記号	意味	記述例
=	等しい	gender = 2
<>	等しくない	prefecture <> "東京"
!=		quantity !=1
>	より大きい	quantity > 2
<	より小さい	revenue < 10000
>=	以上	cost >= 1000
<=	以下	birthday <= "2000-12-31"

- 日付型、日時型、文字列型のデータは「'」または「"」で囲む
- 整数型、数値型、浮動小数点型のデータは半角で記述し、「"」で囲まない
- 日付型、日時型のデータを不等号で指定する場合、古い日付・日時ほど小さい、新しい日付・日時ほど大きい扱いとなる
- 以上（<=）と以下（>=）は不等号の次に等号を記述する

データを分析するときには、「○年だけの売上」や「商品○だけの利益率」「○歳以上の顧客」など、データを絞り込んで確認する状況が多くあり、実務上、WHEREは非常によく使われます。

複数の条件を組み合わせる

WHEREの条件式では、性別と誕生日、販売個数と販売金額など、複数の条件を組み合わせて指定することも可能です。「かつ」を表す「AND（アンド）」と「または」を表す「OR（オア）」を利用して条件式を連結します。

それぞれの記述方法と意味は、以下の表［3-3-8］の通りです。ANDとORの前後には、半角スペースを挿入してください。

3-3-8 ANDとORの記述方法と意味

演算子	記述方法	意味
AND	条件1 AND 条件2	条件1かつ条件2に合致する
OR	条件1 OR 条件2	条件1または条件2に合致する

ANDとORの具体的な記述方法を見ていきましょう。例えば「性別が女性（2）」かつ「誕生日が1999年1月1日以降」という条件は、ANDを使って以下のSQL文［3-3-9］の2行目のように記述します。

一方、「販売個数が4以上」または「販売金額が5000以上」という条件は、ORを使って次ページの［3-3-10］の2行目のように記述します。

3-3-9 ANDの記述例

```
1  SELECT * FROM sample.customers
2  WHERE gender = 2 AND birthday >= "1999-01-01"
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	14044	太田 美由紀	1999-01-22	2	佐賀	2018-11-04	false
2	15015	松江 摩理子	2000-03-19	2	東京	null	false
3	13247	塚原 雪菜	2000-07-26	2	長崎	2017-07-19	false
4	14182	宮本 かおり	2000-09-07	2	長崎	2019-07-01	false

「性別が女性（2）」かつ「誕生日が1999年1月1日以降」という条件で絞り込んだ

3-3-10 ORの記述例

```
1  SELECT * FROM sample.sales
2  WHERE quantity >= 4 OR revenue >= 5000
```

結果テーブル

行	order_id	user_id	product_id	date_time	quantity	revenue	is_proper
1	118856	12772	1	2018-09-06T04:48:59	3	5400	false
2	184864	19940	1	2018-02-01T14:59:17	4	8000	true
3	115355	16550	1	2019-09-06T09:10:38	4	8000	true
4	125185	19375	1	2018-07-23T20:59:21	4	8000	true
5	107827	18415	1	2018-10-21T09:37:37	4	8000	true
6	129303	14328	1	2018-12-05T21:14:26	4	8000	true
7	167725	15446	1	2017-07-01T10:49:08	3	6000	true
8	197572	17809	1	2019-07-23T12:39:55	3	6000	true
9	127928	14298	1	2019-12-28T12:57:45	3	6000	true

「販売個数が4以上」または「販売金額が5000以上」という条件で絞り込んだ

また、ANDやORは2つ以上をつなげて指定することもできます。ANDのみ、ORのみで条件を指定する場合は、記述順を気にする必要はありません。例えば、以下の2つの条件式の意味は同じになります。

条件1 AND **条件2** AND **条件3**
条件3 AND **条件2** AND **条件1**

一方、ANDとORを組み合わせて条件を指定する場合は、記述順によって意味が異なってくるので注意が必要です。意図しないデータを取得してしまうことのないよう、**優先する条件を半角カッコで囲み、条件式を精査する**ようにしてください。

例えば、次ページのSQL文［3-3-11］の2〜3行目にある条件式は、「性別が男性（1）」または「都道府県が東京」に合致し、かつ「誕生日が2000年1月1日以降」である、という意味になります。

3-3-11 ANDとORを組み合わせた記述例①

```
1  SELECT * FROM sample.customers
2  WHERE (gender = 1 OR prefecture = "東京")
3  AND birthday >= "2000-01-01"
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	14761	田島 茂	2000-09-27	1	宮崎	2019-05-25	false
2	12177	佐藤 直哉	2001-12-03	1	北海道	2017-05-18	false
3	15015	松江 摩理子	2000-03-19	2	東京	*null*	false

「性別が男性（1）」または「都道府県が東京」に合致し、かつ「誕生日が 2000 年 1 月 1 日以降」という条件で絞り込んだ

しかし、以下のSQL文［3-3-12］のように半角カッコで囲む箇所を変更すると、「性別が男性（1）」である、または「都道府県が東京」かつ「誕生日が2000年1月1日以降」に合致する、という意味になり、結果が大きく変わります。実際にクエリエディタで実行し、［3-3-11］の結果テーブルとの違いを確認してみてください。

3-3-12 ANDとORを組み合わせた記述例②

```
1  SELECT * FROM sample.customers
2  WHERE gender = 1 OR (prefecture = "東京"
3  AND birthday >= "2000-01-01")
```

条件式に利用できる演算子

WHEREの条件式には、前述した記号とAND、ORのほかにも、以下の表［3-3-13］に挙げた「演算子」が利用できます。これらの演算子はフィールド名に続けて記述します。

それぞれの具体的な記述例について、以降で1つずつ見ていきましょう。

3-3-13 条件式における演算子の記述方法と意味

演算子	記述方法	意味
IN	IN（値1 , 値2, 値3）	いずれかの値に当てはまる
NOT IN	NOT IN（値1 , 値2, 値3）	いずれの値にも当てはまらない
LIKE	LIKE（パターン）	パターンに当てはまる
NOT LIKE	NOT LIKE（パターン）	パターンに当てはまらない
BETWEEN	BETWEEN 値1 AND 値2	値1から値2の間に当てはまる
NOT BETWEEN	NOT BETWEEN 値1 AND 値2	値1から値2の間に当てはまらない

▶IN

「IN（イン）」は、半角カッコ内で指定した「いずれかの値に当てはまる」という意味になります。次ページのSQL文［3-3-14］では、[quantity] フィールドの値が「2」「4」「6」のいずれかである、と指定しています。

その下のSQL文［3-3-15］では、[prefecture] フィールドの値が「青森」「岐阜」「鹿児島」のいずれかである、と指定しています。文字列型のデータを条件に指定する場合は、このように「'」または「"」で値を囲んでください。

3-3-14 INの記述例①

```
1  SELECT * FROM sample.sales
2  WHERE quantity IN (2,4,6)
```

結果テーブル

行	order_id	user_id	product_id	date_time	quantity	revenue	is_proper
1	185395	16653	1	2018-06-28T15:32:22	2	3600	false
2	105331	14791	1	2019-01-06T10:55:32	2	3600	false
3	184864	19940	1	2018-02-01T14:59:17	4	8000	true
4	115355	16550	1	2019-09-06T09:10:38	4	8000	true
5	125185	19375	1	2018-07-23T20:59:21	4	8000	true
6	107827	18415	1	2018-10-21T09:37:37	4	8000	true
7	129303	14328	1	2018-12-05T21:14:26	4	8000	true
8	144415	16668	1	2019-05-09T17:04:52	2	3200	false
9	121930	18945	1	2018-06-11T12:34:43	2	3200	false

［quantity］フィールドの値が「2」「4」「6」のいずれかであるレコードに絞り込んだ

3-3-15 INの記述例②

```
1  SELECT * FROM sample.customers
2  WHERE prefecture IN ("青森", "岐阜", "鹿児島")
```

▶NOT IN

INの前にNOTを付けた「NOT IN（ノット イン）」は、「いずれの値にも当てはまらない」という意味になります。つまり、半角カッコ内で指定した値以外のレコードを取得します。

次ページのSQL文［3-3-16］では、［prefecture］フィールドの値が「東京」「大阪」「愛知」以外である、という条件を指定しています。

3-3-16 NOT INの記述例

```
1  SELECT * FROM sample.customers
2  WHERE prefecture NOT IN ("東京", "大阪", "愛知")
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	11487	石塚 拓	1978-11-26	1	京都	2019-09-01	false
2	14486	長坂 賢介	1992-07-18	1	京都	2017-07-14	false
3	14118	野中 裕之	1994-05-08	1	佐賀	2019-01-01	false
4	13507	高松 麗	1983-02-07	1	兵庫	2018-11-14	false
5	14983	小倉 慎太郎	1984-07-04	1	兵庫	2018-11-26	false
6	11202	大辻 烈	1985-06-27	1	千葉	null	false

[prefecture]フィールドの値が「東京」「大阪」「愛知」以外であるレコードに絞り込んだ

▶LIKE

「LIKE（ライク）」は英語のlikeと同じく、「〜のような」という意味で用いる演算子です。LIKEに続く半角カッコ内で文字列型のデータを記述してパターンを指定し、それに合致するレコードのみを取得します。

まずは以下の表［3-3-17］に記載した、パターンの指定に利用する3つの記号を覚えてください。

3-3-17 LIKEと組み合わせる記号の意味

記号	読み	意味
%	パーセント	任意の数の任意の文字
_	アンダースコア	1つの任意の文字
\	バックスラッシュ	エスケープ処理をする

LIKEの具体的な記述例を見ていきましょう。例えば、「木」から始まる姓名には「木村 佳乃」「木梨 憲武」などさまざまにありますが、これをパターンとして表すと「木」＋「任意の数の任意の文字」となります。

これは前述の記号を使って表すと「木%」となり、SQL文に記述すると以下の［3-3-18］の2行目のようになります。パターンは文字列型のデータなので、「'」または「"」で囲むことを忘れないようにしてください。

3-3-18 LIKEの記述例①

```
1  SELECT * FROM sample.customers
2  WHERE name LIKE ("木%")
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	11779	木ノ下 圭恵	1979-04-09	2	東京	2018-06-05	false
2	12161	木内 麻美	1978-06-09	2	茨城	2018-11-26	false

氏名が「木」で始まるレコードを取得できる

今度は、姓名の名が「○○子」で終わる、というパターンを考えてみましょう。演習用の［customers］テーブルにある［name］フィールドには、姓と名の間に半角スペースがあることも考慮します。

姓は何でもかまわないので「任意の数の任意の文字」、つまり「%」で表せます。名が「○○子」で終わる、のパターンは「1つの任意の文字」＋「1つの任意の文字」＋「子」となり、記号で表すと「_ _子」です。結果、LIKEで指定するパターンは「% _ _子」となり、以下のSQL文［3-3-19］の2行目のようになります。

3-3-19 LIKEの記述例②

```
1  SELECT * FROM sample.customers
2  WHERE name LIKE ("% __子")
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	12465	土井 佳菜子	1978-10-17	2	京都	2018-09-27	false
2	14024	米田 由利子	1973-11-28	2	千葉	2018-07-16	false
3	14154	久保 江美子	1969-11-24	2	埼玉	2019-04-22	false
4	16668	三浦 亜沙子	1984-09-15	2	埼玉	2017-01-10	false
5	12066	上原 英美子	1997-12-25	2	大分	2019-05-30	false
6	15906	西山 由香子	1991-10-21	2	大阪	2017-11-29	false

名が「○○子」で終わるレコードを取得できる

▶NOT LIKE

LIKEの前にNOTを付けた「NOT LIKE（ノット ライク）」は、パターンに一致しないレコードを取得します。パターンの指定方法はLIKEと同様です。

▶BETWEEN

主に数値や日付など、指定した2つの値の範囲内にあるレコードを取得したいときに利用するのが「BETWEEN（ビトゥイーン）」です。

例えば、[birthday] フィールドの値が「1999年1月1日から1999年12月31日の間」にあるレコードは、「BETWEEN 値1 AND 値2」の構文として、以下のSQL文［3-3-20］の2行目のように表せます。条件に日付を指定するときは「'」または「"」で囲んでください。

3-3-20 BETWEENの記述例

```
1  SELECT * FROM sample.customers
2  WHERE birthday BETWEEN "1999-01-01" AND "1999-12-31"
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	14044	太田 美由紀	1999-01-22	2	佐賀	2018-11-04	false
2	14726	土門 茉梨奈	1999-08-19	2	北海道	2019-02-28	false
3	17236	北原 莉沙	1999-11-21	2	佐賀	2017-05-14	true
4	17880	園田 美由紀	1999-04-17	2	岩手	2017-07-07	true
5	17809	井筒 彩耶	1999-12-28	2	東京	2017-02-28	true
6	18923	三須 亜美香	1999-04-21	2	熊本	2018-01-13	true
7	19307	柳瀬 純乃	1999-01-06	2	青森	2018-03-03	true

誕生日が「1999年1月1日から1999年12月31日の間」にあるレコードを取得できる

なお、前述の「BETWEEN 値1 AND 値2」の「AND」は、BETWEENの構文中のもので、複数の条件を組み合わせるAND演算子とは意味合いが異なります。BETWEENで条件を指定し、さらに追加の条件をANDで組み合わせたいときには混同しやすいので、注意してください。

仮に、[birthday] フィールドの値が「1999年1月1日から1999年12月31日の間」にあるレコードをAND演算子で表現しようとすると、以下のSQL文［3-3-21］の2〜3行目のようになります。比べてみると、BETWEENのほうがスッキリと表現できていることが分かると思います。

3-3-21 BETWEENを使わない記述例

```
1  SELECT * FROM sample.customers
2  WHERE birthday >= "1999-01-01"
3  AND birthday <= "1999-12-31"
```

そのほか、BETWEENの使い方として2つ注意したいことがあります。1つは「**日本語の文字列を範囲として指定しない**」ことです。

BETWEENは文字列型のフィールドに対しても利用でき、格納されている文字列がアルファベットの場合、「a」がもっとも小さく、「z」がもっとも大きい値として扱われます。例えば、次ページのSQL文［3-3-22］では、「Adam」「Ben」「Charles」「Dennis」「Ed」が格納されたテーブルに対して、2行目のWHEREでBETWEENを使って「Adam」から「Charles」まで絞り込んでいます。結果テーブルでは「Adam」「Ben」「Charles」が返されています。

しかし、対象のフィールドに日本語の文字列が格納されている場合は、結果が定かでないため利用を避けるべきです。したがって、順序を持つフィールドである整数型、浮動小数点型、数値型、日付型、日時型、タイムスタンプ型のデータに対して利用するようにしましょう。

3-3-22 BETWEENで文字列の範囲を指定する

```
1  SELECT first_name FROM sample.letter
2  WHERE first_name BETWEEN "Adam" AND "Charles"
```

結果テーブル

行	first_name
1	Adam
2	Ben
3	Charles

BETWEENは文字列型のデータも対象にできるが、日本語の文字列への利用は避ける

もう1つは、**「BETWEEN値1 AND値2」の2つの値で「値1＜値2」の関係を保つ**ことです。「値1＞値2」と指定した場合は合致するレコードが存在しないため、結果は表示されません。

例えば、以下のSQL文［3-3-23］の2行目は誤った指定で、正しくは「BETWEEN 3000 AND 3400」と記述する必要があります。

3-3-23 BETWEENで値の範囲を指定する（誤用例）

```
1  SELECT * FROM sample.sales
2  WHERE revenue BETWEEN 3400 AND 3000
```

「値1＜値2」の関係になっておらず、結果は表示されない

▶NOT BETWEEN（ノット ビトウイーン）

「NOT BETWEEN」は、指定した2つの範囲にないレコードを取得します。範囲の指定方法はBETWEENと同様です。

「true」「falase」の判定

「true」または「falase」のデータが格納されるブール型のフィールドに対する絞り込みには、「IS TRUE（イズ トゥルー）」「IS FALSE（イズ フォールス）」の条件を指定します。

演習用の［sales］テーブルにある［is_proper］フィールドには、販売価格が定価（true）か、割引価格（false）かというブール型の値が格納されています。例えば、定価で販売されたレコードを取得するには、以下のSQL文［3-3-24］を記述します。

3-3-24 IS TRUEの記述例

```
1  SELECT * FROM sample.sales
2  WHERE is_proper IS TRUE
```

結果テーブル

行	order_id	user_id	product_id	date_time	quantity	revenue	is_proper
1	184864	19940	1	2018-02-01T14:59:17	4	8000	true
2	115355	16550	1	2019-09-06T09:10:38	4	8000	true
3	125185	19375	1	2018-07-23T20:59:21	4	8000	true
4	107827	18415	1	2018-10-21T09:37:37	4	8000	true
5	129303	14328	1	2018-12-05T21:14:26	4	8000	true
6	167725	15446	1	2017-07-01T10:49:08	3	6000	true
7	197572	17809	1	2019-07-23T12:39:55	3	6000	true
8	127978	14298	1	2019-12-28T12:57:45	3	6000	true

定価で販売（is_proper = true）されたレコードのみに絞り込んだ

逆に、割引価格で販売されたレコードを取得するには、上記のIS TRUEをIS FALSEに書き換えて実行します。もしくは、次ページのSQL文［3-3-25］のように「IS NOT TRUE（イズ ノット トゥルー）」と書き換えても同じ結果になります。

ただし、これは［is_proper］フィールドに「null」が存在しないためで、もし「null」のレコードが存在する場合は、IS FALSEはfalseのレコードだけ、IS NOT TRUEはfalseと「null」のレコードを結果テーブルとして返します。「null」の取り扱いは重要なので、以降でも別途解説します。

3-3-25 IS NOT TRUEの記述例

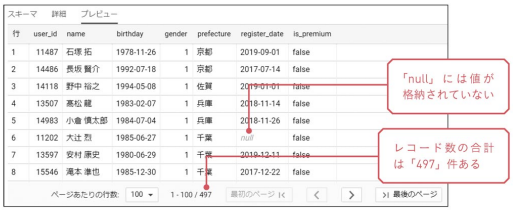
```
1  SELECT * FROM sample.sales
2  WHERE is_proper IS NOT TRUE
```

「null」の判定

「ユーザー登録時の誕生日は必須項目でなかった」「以前はECサイトに会員制度がなく、登録日は任意入力の項目だった」など、自社で扱っているデータベース（テーブル）の特定のフィールドに値が格納されていないことがあります。値が格納されていない状態を「null（ヌル）」といい、この「null」を判定するための条件として「IS NULL（イズ ヌル）」があります。

IS NULLの構文を紹介する前に、まずは「null」の重要性を理解してください。演習用の［customers］テーブルをプレビュー機能を使って表示すると、以下の図［3-3-26］に示すように、いくつかのフィールドの値として「null」が含まれています。また、全部で「497」件のレコードがあることを確認できるはずです。

3-3-26 「null」が含まれているフィールドの例



その［customers］テーブルに対して、以下のSQL文［3-3-27］を実行してみてください。「誕生日が1999年1月1日以降」のレコードが取得され、結果は「15」件となるはずです。

続いて、その下のSQL文［3-3-28］も実行しましょう。「誕生日が1998年12月31日以前」のレコードが取得され、結果は「444」件となります。

3-3-27 誕生日が1999年1月1日以降のレコードを取得する

```
SELECT * FROM sample.customers
WHERE birthday >= "1999-01-01"
```

3-3-28 誕生日が1998年12月31日以前のレコードを取得する

```
SELECT * FROM sample.customers
WHERE birthday <= "1998-12-31"
```

2つのSQL文の結果を合計すると、レコード数は15＋444＝「459」件となりますが、［customers］テーブルのレコード数の合計は「497」件です。その差は497−444＝「38」件で、この38レコードの［birthday］フィールドには値が格納されていない、すなわち「null」であることが分かります。

では、これをIS NULLを利用して検証してみましょう。次ページのSQL文［3-3-29］を実行すると、結果テーブルで示した通り、レコード数は「38」件となりました。

3-3-29 「null」のレコードを取得する

```
1  SELECT * FROM sample.customers
2  WHERE birthday IS NULL
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	13295	中山 剛	null	1	山形	null	false
2	12829	河野 大樹	null	1	岡山	null	false
3	16638	谷田 聡士	null	1	東京	null	false
4	15571	武藤 洋	null	1	福岡	null	false
5	10329	野村 正志	null	1	鳥取	null	false

ページあたりの行数: 100 1 - 38 38 最初

「null」のレコードが 38 件であることを確認できた

なお、nullではないレコードを取得するには、以下のSQL文［3-3-30］のように「IS NOT NULL（イズ ノット ヌル）」を利用します。

3-3-30 「null」ではないレコードを取得する

```
1  SELECT * FROM sample.customers
2  WHERE register_date IS NOT NULL
```

「null」の存在を考慮していなかったことが原因で、分析に利用したデータが実は正しくないものだった、というミスがよく起こります。適宜「WHERE フィールド名 IS NULL」の構文を利用し、「null」の有無を確認するようにしましょう。

STEP UP

SQLの書き方によって料金が変わる！？

SECTION 2-3で解説した、BigQueryでSQLを実行するときの料金（クエリの料金）を覚えているでしょうか？ サンドボックスを利用しているうちは料金がかかりませんが、本書での学習を終え、BigQueryを実務で利用するようになれば、料金が気になりはじめます。

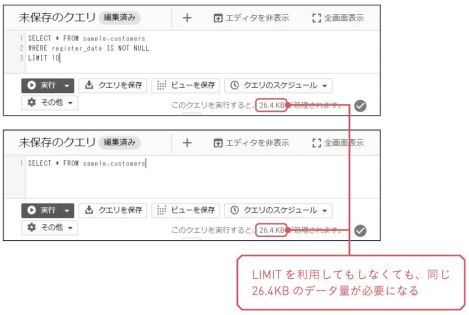
この料金はSQLを実行した量に応じて発生しますが、ついつい実行結果、つまり結果テーブルのデータ量に応じて発生しているように思い込んでしまう人がいます。このような人は「本章で学んできたLIMITやWHEREを使ってレコードを絞り込めば、料金を節約できるでは？」と考えるかもしれませんが、実際は節約になりません。

なぜなら、LIMITやWHEREを利用して取得した結果が数レコードでも数百レコードでも、**SQL文の実行対象が同じであれば、クエリエディタが処理するデータ量は変わらない**からです。処理するデータ量が変わらないのですから、クエリの料金も変わりません。

次ページの［3-3-31］を見てください。「SELECT *」と記述しているので、2つのSQL文はいずれも［customers］テーブルの全フィールドを取得しています。上の画面ではWHEREとLIMITを利用し、下の画面では利用していませんが、クエリエディタが処理するデータ量はいずれも「26.4KB」です。実行対象が同じである以上、WHEREやLIMITを利用しようがしまいが、処理するデータ量は変わらないことを証明しています。

サンドボックスを卒業し、クレジットカードを登録して本利用に進んだ場合、無料で提供されるクエリ処理量は1TB/月です。1TBは1,000GBなので、処理するクエリ量がMB単位のときには、料金について気にする必要はほとんどないと思ってよいでしょう。

3-3-31 実行対象が同じなら処理するデータ量も同じ



しかし、SELECTで取得するフィールドを指定すれば、実行対象となるテーブルが同じでも、フィールドを減らすことでデータ量を節約できます。以下の［3-3-32］では［user_id］フィールドのみを取得しており、データ量が「3.9KB」に削減できることを確認できます。**料金面を考えると安易に「SELECT *」を利用するべきではない**といえますね。

3-3-32 フィールドを指定すればデータ量を減らせる

STEP UP

FROMがなくてもSQL文は成立する

ここまで学び終えたみなさんは、SQL文において「SELECTとFROMは必須」と思われているのではないでしょうか。前掲のすべてのSQL文の中に、SELECTとFROMが記述されていたからです。

確かにSELECTは必須なのですが、実はFROMは必須ではありません。試しに、以下のSQL文［3-2-33］を実行してみてください。

3-2-33 FROMを省略した例①

```
1  SELECT 2 * 5
```

戻り値は「10」となります。SQLはテーブルに格納されたデータを処理する以外に、計算機能も備えているのですね。

ほかにも、以下の［3-2-34］のように記述すると、SQL文を実行した日の日付が返ってくることを確認できるはずです。

3-2-34 FROMを省略した例②

```
1  SELECT CURRENT_DATE() AS Today
```

この「実行した日を"今日"として取得できる」というSQLの機能を利用すると、例えば「最終購入日から今日までに○日以上経過した顧客のリスト」など、毎日変動する"今日"を基準にしたデータの取得が可能になります。SQLが持つ便利な機能の1つとして覚えておいてください。

SECTION

3-4 確認ドリル

本章から「確認ドリル」がはじまります。CHAPTER 2で解説したように、BigQueryの環境と、本書の演習用ファイルから作成したテーブルを準備して、ぜひチャレンジしてください。解答は問題の後に掲載しています。

問題 001

［sales］テーブルのすべてのフィールドから［revenue］だけを取得せず、代わりに［revenue］に「1.1」を掛けた値を［revenue_with_tax］に格納してください。ただし、SELECTには「*」を使用し、結果テーブルは［order_id］の小さい順に3レコードに絞り込んでください。

問題 002

［sales］テーブルを対象に、［order_id］［quantity］［revenue］フィールドに加えて、［quantity］を「1」増やした個数のフィールド（new_qty）と、［quantity］を「1」増やしたときの販売金額のフィールド（new_revenue）を取得してください。なお、結果テーブルは［new_revenue］の大きい順に3レコードに絞り込んでください。

問題 003

［customers］テーブルから、［birthday］フィールドの値が「null」でなく、［is_premium］が「true」のレコードの全フィールドを取得してください。結果テーブルは年齢の若い順に並べ替え、誕生日が同じ場合は［register_date］の古い順に並べ替えてください。また、3レコードに絞り込んでください。

問題 004

［customers］テーブルから『「プレミアム顧客」または「1970年代生まれの顧客」』で「名前の最後が"美"で終わる女性」の全フィールドを取得してください。結果テーブルは年齢が高い順の3レコードに絞り込みます。

問題 005

［customers］テーブルから、[prefecture] フィールドが「東京」「千葉」「埼玉」「神奈川」以外のプレミアム顧客の全フィールドを取得してください。結果テーブルは、年齢が若い順に並べ替えて、3レコードだけ表示します。

解答

解答 001

```
1  SELECT * EXCEPT (revenue)
2  , revenue * 1.1 AS revenue_with_tax
3  FROM sample.sales
4  ORDER BY 1
5  LIMIT 3
```

結果テーブル

行	order_id	user_id	product_id	date_time	quantity	is_proper	revenue_with_tax
1	100393	13576	11	2019-01-04T14:35:32	1	true	1320.0
2	100393	13576	4	2019-01-04T14:35:32	1	true	2640.0
3	100407	15686	2	2019-04-18T11:19:31	1	false	2178.0

「SELECT *」で全フィールドを取得できること、EXCEPTで特定のフィールドを除外できること、「*」に続けて「,」で区切って、別のフィールドを追加できることを学びました。この3つの知識で正解にたどり着けたと思います。

解答 002

```
SELECT order_id
, quantity
, quantity + 1 AS new_quantity
, revenue
, (revenue / quantity) * (quantity + 1) AS new_revenue
FROM sample.sales
ORDER BY 5 DESC
LIMIT 3
```

結果テーブル

行	order_id	quantity	new_quantity	revenue	new_revenue
1	121169	4	5	12000	15000.0
2	127500	3	4	10800	14400.0
3	146864	3	4	10800	14400.0

結果テーブルの5列目にある［new_revenue］フィールドの取得には、少し工夫が必要です。元の［sales］テーブルには単価を表すフィールドはないので、［revenue］を［quantity］で割って単価を計算し、「quantity + 1」を掛けることで「注文の個数が1個多いときの販売金額」が求められます。また、［new_revenue］を大きい順に並べ替えるORDER BYをDESCオプションとともに利用します。3レコードに絞り込むには、LIMITを利用します。

解答 003

```
1  SELECT * FROM sample.customers
2  WHERE birthday IS NOT NULL AND is_premium IS TRUE
3  ORDER BY birthday DESC, register_date
4  LIMIT 3
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	19690	平良 美友紀	2000-07-14	2	石川	2017-05-19	true
2	17809	井筒 彩耶	1999-12-28	2	東京	2017-02-28	true
3	17236	北原 莉沙	1999-11-21	2	佐賀	2017-05-14	true

レコードの絞り込みとしてIS (NOT) TRUE/FALSE、IS (NOT) NULLを学びました。この解答では、それらを組み合わせてWHERE句で利用しています。SQL文の2行目にある「is_premium IS TRUE」は、この例では「is_premium IS NOT FALSE」でも同じ結果になりますが、[is_premium] フィールドに「null」が存在した場合には結果が変わるので、注意してください。

ORDER BYの表現は、列番号を利用して「ORDER BY 3 DESC, 6」としても構いません。なお、誕生日が同じユーザーがいるかどうかは、CHAPTER 4で学ぶテクニックを使うと確認できます。

解答 004

```sql
SELECT *
FROM sample.customers
WHERE (is_premium IS TRUE OR
birthday BETWEEN "1970-01-01" AND "1979-12-31")
AND name LIKE ("%美")
AND gender = 2
ORDER BY birthday
LIMIT 3
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	17097	美馬 素美	1969-05-09	2	兵庫	2018-01-24	true
2	15008	遠山 鮎美	1970-06-15	2	埼玉	2017-01-22	false
3	15543	平沼 吉美	1970-10-08	2	山梨	2019-06-16	false

WHEREを構成する条件がポイントです。「プレミアム顧客」の条件は「is_premium IS TRUE」で表せます。「1970年代生まれ」の条件には、BETWEEN演算子を使います。

「プレミアム顧客」と「1970年代生まれ」はOR条件なので、それらを半角カッコでまとめ、「姓名の最後が"美"で終わる」と「女性」の条件は、それぞれANDでつなげます。

解答 005

```
SELECT * FROM sample.customers
WHERE prefecture NOT IN ("東京", "千葉", "神奈川","埼玉")
AND is_premium IS TRUE
ORDER BY birthday DESC
LIMIT 3
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	19690	平良 美友紀	2000-07-14	2	石川	2017-05-19	true
2	17236	北原 莉沙	1999-11-21	2	佐賀	2017-05-14	true
3	18923	三須 亜美香	1999-04-21	2	熊本	2018-01-13	true

［prefecture］フィールドが「東京」「千葉」「埼玉」「神奈川」以外なので、WHEREでは次ページのように東京、埼玉、神奈川、千葉以外の43道府県をORでつないでも同じ結果が得られます。

```
(prefecture = "北海道" OR
prefecture = "青森" OR
prefecture = "……" OR
prefecture = "沖縄")  AND is_premium IS TRUE
```

ただ、IN演算子を否定するNOT INを利用して、「東京、千葉、神奈川、埼玉に当てはまらない」という条件と、「is_premium IS TRUE」をANDでつなぐほうが断然スマートです。年齢が若い順は「誕生日の大きい順」と考えれば、［birthday］を降順に並べ替えればよい、と思いつくと思います。

CHAPTER 4

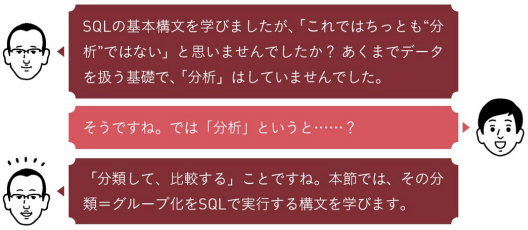
グループ化とデータの集計

4-1　グループ化
4-2　集計関数
4-3　集計結果の絞り込み
4-4　柔軟なグループ化
4-5　確認ドリル

本章では「GROUP BY」構文を利用したグループ化と、集計関数を利用したグループ別の集計値の取得、集計結果の絞り込みなどを学びます。データを任意のカラムの値でグループ化して集計することで、分析の基本であるグループ別の比較ができるようになり、「SQLは分析に使える」という実感を得られるはずです。

SECTION

4-1 グループ化



複数のレコードをまとめて1行にする

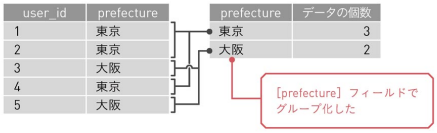
SQL文の記述方法を解説する前に、まずは「グループ化とはどのようなことか？」をおさらいしておきましょう。以降、グループ化や集計関数の挙動を分かりやすくするために、演習用テーブルとは別の、人間の目で理解しやすい小さなテーブルを例に解説します。

次ページの図［4-1-1］の左側にある表は、[user_id] [prefecture] という2カラムのテーブルです。[user_id] には整数型、[prefecture] には文字列型のデータが格納されています。

このテーブルを［prefecture］でグループ化すると、右側の表のようになります。元は5レコードあったものが、グループ化した後は2レコードになりました。つまり、**グループ化とは同じ値が入った複数のレコードをまとめて1行にすること**と表現できます。

グループ化した結果、例えば「東京」の行に個別の［user_id］の値を含めることはできなくなります。一方、何レコードあるか（顧客が何人いるか）を表す「データの個数」は含めることが可能です。

4-1-1 グループ化の基本①

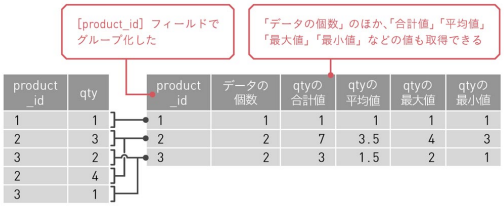


グループ化の例をもう1つ見てみましょう。以下の図［4-1-2］では、[product_id] と [qty]（quantity）の2カラムのテーブルを、[product_id] でグループ化しています。データ型はいずれも整数型です。

[product_id] フィールドには、「1」「2」「3」の3種類の値があります。1つの [product_id] の値が1つのレコードにまとまり、右側のグループ化したテーブルでは3レコードになりました。また、前掲の［4-1-1］ではデータの個数だけが集計されましたが、今度は「合計値」「平均値」「最大値」「最小値」などの集計値も存在しています。

つまり、グループ化した結果、[product_id] の各レコードに個別の [qty] の値を含めることはできなくなりますが、**量的なデータが格納されているフィールドであれば、合計や平均などの集計値を取り出すことが可能**です。ここまでが、もっとも基本的なグループ化の考え方となります。

4-1-2 グループ化の基本②



グループ化する基準によって結果は異なる

グループ化を実行するときは「どのフィールドを基準としてグループ化するか？」を考えることが重要です。同じテーブルでも、グループ化するフィールドによって結果が異なるためです。

以下の図［4-1-3］では、［4-1-2］に［is_proper］フィールドを加えた表を［product_id］でグループ化しています。［is_proper］にはブール型のデータが格納されていますが、［product_id］を基準としているため、グループ化の結果は変わりません。一方、その下の図［4-1-4］では、グループ化する基準を［is_proper］に変更しています。結果は「T」と「F」の2レコードにまとまり、［qty］の集計値も異なっています。

4-1-3 グループ化するフィールドによる結果の違い①

product_id	is_proper	qty
1	T	1
2	T	3
3	F	2
2	T	4
3	F	1

product_id	データの個数	qtyの合計値	qtyの平均値	qtyの最大値	qtyの最小値
1	1	1	1	1	1
2	2	7	3.5	4	3
3	2	3	1.5	2	1

［product_id］フィールドでグループ化した

4-1-4 グループ化するフィールドによる結果の違い②

product_id	is_proper	qty
1	T	1
2	T	3
3	F	2
2	T	4
3	F	1

is_proper	データの個数	qtyの合計値	qtyの平均値	qtyの最大値	qtyの最小値
T	3	8	2.7	4	1
F	2	3	1.5	2	1

［is_proper］フィールドでグループ化した

［product_id］でグループ化した場合とは、レコード数や集計値が異なっている

また、**複数のフィールドを基準としてグループ化する**こともできます。前掲の［4-1-3］［4-1-4］と同じ元のテーブルのレコード数を少し増やし、[product_id] と [is_proper] の2つのフィールドでグループ化すると、以下の［4-1-5］のようになります。

[product_id] には3つ、[is_proper] には2つの値が格納されているので、結果テーブルのレコード数は3×2＝6レコードです。それぞれのレコードは「1 - T」「1 - F」「2 - T」…のように2つのフィールドの値を掛け合わせたものになり、それぞれの集計値を取得できます。[is_proper] [product_id] の順でグループ化してもよく、その場合は結果テーブルの1列目と2列目が入れ替わるだけで、合計や平均などの集計値は同じです。

4-1-5 2つのフィールドによるグループ化

product_id	is_proper	qty
1	T	1
2	T	3
3	F	2
2	T	4
3	F	1
1	F	2
2	F	1
1	F	1
3	T	3
2	F	5

product_id	is_proper	データの個数	qtyの合計値	qtyの平均値	qtyの最大値	qtyの最小値
1	T	1	1	1	1	1
1	F	2	3	1.5	2	1
2	T	2	7	3.5	4	3
2	F	2	6	3	5	1
3	T	1	3	3	3	3
3	F	2	3	1.5	2	1

［product_id］［is_proper］の2つのフィールドでグループ化した

グループ化の基本と、どのフィールドでグループ化するかによって結果のレコード数や集計値が変化することがお分かりいただけたでしょうか。

はい！「顧客ごと」「商品ごと」にデータを分類するというのは、分析の基本でもありますね。

フィールドを指定してグループ化する

グループ化の概念が理解できたところで、グループ化をSQLで実現する命令である「GROUP BY（グループ バイ）」の記述方法を見ていきます。以下の［4-1-6］はGROUP BYの構文を示したものです。

4-1-6 GROUP BYの構文

```
1  SELECT フィールド名，集計関数
2  FROM テーブル名
3  GROUP BY フィールド名
```

フィールド名	グループ化に利用するフィールドを指定。SELECT句とGROUP BY句には、同じフィールドを指定する
集計関数	データの個数を数える「COUNT」や合計する「SUM」などの集計関数を指定（詳しくは次節で解説）
テーブル名	SQLの実行対象とあるテーブルを指定

この構文の通り、GROUP BYで指定するフィールド名は、SELECTでも同じものを指定するのが一般的です。ただし、指定しなくてもエラーにはなりません（本節末のSTEP UPを参照）。

ただし、次の2点には注意が必要です。

1. グループ化の対象にしないフィールドをSELECT句に指定してはいけない
2. 量的なデータ（販売金額や個数など）のフィールドは集計関数で集計しなくてはいけない

以下のSQL文［4-1-7］を見てください。本来は［product_id］フィールドのみを対象としてグループ化し、［qty］の合計値を取得する意図で記述したのですが、誤ってSELECT句に［is_proper］を含めてしまいました。これは前述の1に該当し、エラーとなります。

エラーメッセージを日本語に訳すと、「SELECT句のリスト表現は［is_proper］を参照していますが、［is_proper］はグループ化も集計もされていません」となります。

4-1-7 グループ化して合計値を取得する（誤用例①）

```
1  SELECT product_id, is_proper, SUM(qty) AS sum_qty
2  FROM sample.small_table
3  GROUP BY product_id
```

SELECT list expression references column is_proper which is neither grouped nor aggregated at [1:20]

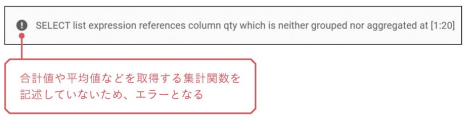
グループ化の対象にしないフィールドまで
SELECT 句で指定しているため、エラーとなる

また、次ページの［4-1-8］は、SELECT句に［qty］フィールドの合計値を取得するための集計関数が記述されておらず、前述の2に該当するエラーとなります。エラーメッセージの意味は「SELECT句のリスト表現は［qty］を参照していますが、［qty］はグループ化も集計もされていません」です。

本書の演習用テーブルはレコード数が多く、SQL文を実行しないと結果が分かりません。そのため、あくまで紙面上での例として、以降も折に触れて「小さなテーブル」が登場します。SQL文中の「small_」で始まるテーブル名は、そのようなテーブルを指しています。

4-1-8 グループ化して合計値を取得する（誤用例②）

```
1  SELECT product_id, qty
2  FROM sample.small_table
3  GROUP BY product_id
```



［product_id］フィールドのみを対象としてグループ化し、［qty］の合計値を［sum_qty］という名前を付けた新しいフィールドとして取得する正しいSQL文は、以下の［4-1-9］となります。グループ化と集計値を取得する基本的なSQL文の記述方法として、しっかり覚えてください。

4-1-9 グループ化して合計値を取得する

```
1  SELECT product_id, SUM(qty) AS sum_qty
2  FROM sample.small_table
3  GROUP BY product_id
```

結果テーブル

行	product_id	sum_qty
1	3	3
2	1	1
3	2	7

［product_id］フィールドでグループ化し、［qty］の合計値を［sum_qty］として取得できた

STEP UP

SELECT句のフィールド名を省略すると？

GROUP BYの構文における注意点として、グループ化の「対象にしない」フィールドをSELECT句に指定してはいけない、と述べました。では逆に、グループ化の「対象にする」フィールドをSELECT句に指定しなかったら、どうなるのでしょうか？

以下のSQL文［4-1-10］では、GROUP BY句で［product_id］フィールドを指定していますが、SELECT句には［product_id］がありません。しかし、エラーにはならず、下に記載した通りの結果テーブルが返ってきます。

ただし、グループ化したフィールド（ここでは［product_id］）が結果テーブルに表示されないため、［sum_qty］の「3」「1」「7」といった値が何を意味しているのかが分かりません。一見、役に立たないように思えますが、実はこのような記述方法にも利用価値があります。詳しくはCHAPTER 6で解説しますので、お楽しみに。

4-1-10 SELECT句のフィールド名を省略する

```
1  SELECT SUM(qty) AS sum_qty
2  FROM sample.small_table
3  GROUP BY product_id
```

結果テーブル

行	sum_qty
1	3
2	1
3	7

エラーにはならないが、グループ化の基準にした［product_id］フィールドが表示されない

SECTION

4-2 集計関数

グループ化の構文の中でも登場した「集計関数」について、本節で学んでいきましょう。合計値を取得するSUM関数以外にも多くの関数があり、マスターすることで分析の幅が広がります。

グループ化と組み合わせて分析を行うために必須

データの個数や値の合計、平均などの集計を行う集計関数は、前節で解説したグループ化と組み合わせて分析を行うために欠かせない機能です。

まずは以下の表［4-2-1］に記載した、BigQueryで利用できる基本的な集計関数をマスターしてください。いずれも関数名に続けて、引数を半角カッコ「()」で囲んで指定します。

4-2-1 基本的な集計関数

構文	機能
COUNT(*)	グループ内のレコード数を数える
COUNT(フィールド名)	グループ内の値の個数を数える
COUNT(DISTINCT フィールド名)	グループ内の固有な値の個数を数える
SUM(フィールド名)	該当フィールドのグループ内の値を合計する
AVG(フィールド名)	該当フィールドのグループ内の値の平均を返す
MAX(フィールド名)	該当フィールドのグループ内の最大値を取得する
MIN(フィールド名)	該当フィールドのグループ内の最小値を取得する

また、本節での集計関数の動作を確認するための小さなテーブルとして、以下の［4-2-2］をSQL文の実行対象として想定します。この［small_users］テーブルには次のような特徴がありますが、以降で実際にグループ化と集計を行うSQL文と実行結果を見ていきましょう。

- ［user_id］フィールドには「1」「2」「3」の3つの値がある。つまり、3人のユーザーがいる
- 1人のユーザーに対して3レコードずつ、全部で9レコードがある
- 1人のユーザーが同一の商品、例えば「クッキー」を複数回購入することがある
- ［product］および［qty］フィールドには「null」となっているレコードがある

4-2-2 集計関数の対象とするテーブル

user_id	product	qty
1	クッキー	1
1	クッキー	2
1	ショートケーキ	6
2	ショートケーキ	null
2	null	2
2	ショートケーキ	6
3	null	0
3	ショートケーキ	2
3	ゼリー	4

演習用テーブル以外の「小さなテーブル」を利用するのは、人間が目で見て関数の挙動を理解するためです。いきなり大きなテーブルに関数を適用すると、挙動や正しさが検証できず、学習には向いていません。

データの個数を数えるCOUNT関数

COUNT関数は文字通り「数える」ための関数で、構文は「COUNT()」です。半角カッコ内に記述する引数は、前掲の表［4-2-1］で示したとおり「*」「フィールド名」「DISTINCT フィールド名」の3つがあります。

1つ目の「COUNT(*)」を使ったSQL文の記述例は、以下の［4-2-3］となります。[user_id] フィールドを基準にグループ化して、それぞれのレコード数を［orders］として取得する、という意味です。

結果テーブルは下に記載の通り、3人のユーザーとも［orders］の値が「3」となり、それぞれ3レコードあることが分かります。[products] や [qty] に「null」があってもお構いなしに、レコード数を返します。

4-2-3 グループ化してレコード数を取得する

```
1  SELECT user_id, COUNT(*) AS orders
2  FROM sample.small_users
3  GROUP BY user_id
```

結果テーブル

行	user_id	orders
1	1	3
2	2	3
3	3	3

レコード数を［orders］として取得できた

2つ目の「COUNT(フィールド名)」を使ったSQL文は、次ページの［4-2-4］となります。COUNT()関数の引数として［product］を指定したことで、[user_id] フィールドごとに何個の［product］を購入したかを、[orders] と名前を付けたフィールドで調べられます。

結果テーブルからは、次に挙げることが分かります。

1. [user_id] が「1」のユーザーが購入したのは「クッキー」「クッキー」「ショートケーキ」で [orders] の値は「3」。つまり、同一の [product] があってもカウントする
2. [user_id] が「2」のユーザーが購入したのは「ショートケーキ」「ショートケーキ」「null」で [orders] の値は「2」。つまり、同一の [product] はカウントしても「null」はカウントしない
3. [user_id] が「3」のユーザーの結果からも上記1、2を確認できる

4-2-4 グループ化して値の個数を取得する

```
1  SELECT user_id, COUNT(product) AS orders
2  FROM sample.small_users
3  GROUP BY user_id
```

結果テーブル

行	user_id	orders
1	1	3
2	2	2
3	3	2

値の個数を［orders］として取得できた

ここで学んでいるCOUNT関数は、非常によく使われます。「レコード数」「データの個数」「ユニークな値の個数」のどの値を取得するのかで結果も意味も変わるので、気をつけながら使いましょう。

3つ目の「COUNT(DISTINCT フィールド名)」ですが、「DISTINCT」は英語で「別個の」という意味があります。この構文を利用したSQL文は以下の［4-2-5］となり、［user_id］フィールドごとに、何種類の固有な［product］を購入したかを、固有の値のみを数えて［orders］として取得します。

結果テーブルからは、次に挙げることが分かります。

1. ［user_id］が「1」のユーザーが購入したのは「クッキー」「クッキー」「ショートケーキ」で［orders］の値は「2」。つまり、同一の［product_id］はカウントに加わらない
2. ［user_id］が「2」のユーザーが購入したのは「ショートケーキ」「ショートケーキ」「null」で［orders］の値は「1」。つまり、同一の［product］はカウントに加わらず、「null」もカウントしない
3. ［user_id］が「3」のユーザーの結果からも上記**1**、**2**を確認できる

4-2-5 グループ化して固有の値の個数を取得する

```
1  SELECT user_id, COUNT(DISTINCT product) AS orders
2  FROM sample.small_users
3  GROUP BY user_id
```

結果テーブル

行	user_id	orders
1	1	2
2	2	1
3	3	2

固有の値の個数を［orders］として取得できた

STEP UP

Web解析でのCOUNT関数の使い分け

「COUNT(フィールド名)」と「COUNT(DISTINCT フィールド名)」の使い分けについて、Web解析における利用例を紹介しましょう。例えば、ユーザーを区別する［user_id］と、そのユーザーが閲覧したWebページのURLが格納されている［page］という、2つのフィールドからなるテーブルがあるとします。

ユーザー別のページビュー数であれば、［user_id］でグループ化したうえで「COUNT(page)」を利用するのが適切です。その場合、ページビュー数はどのようなページを見ても構わないので、トップページを2回見たケースでは「2」とカウントされます。

一方、［user_id］でグループ化したうえで「COUNT(DISTINCT page)」を利用した場合、ユーザーごとの閲覧したページの種類数を表すことになります。トップページを2回見たケースでは「1」とカウントされます。

一般的にユーザーは同一のページを複数回見ることがあるので、「COUNT(page)」と「COUNT(DISTINCT page)」の値は相当に異なります。どちらの値を入手したいのか、意識して使い分けることが必要です。

従来のGoogleアナリティクスでは、基本的にWebログが提供されていないので、SQLで分析する機会はほぼなかったと思います。一方、GA4ではBigQueryにWebログがエクスポートされるので、GA4が一般的になればなるほど、SQLが活躍する機会は増えていくはずです。今のうちに演習用の［web_log］テーブルで、Webログに対するSQLを学ぶことには意義があります。

合計値、平均値、最大値、最小値を取得する関数

今度は、前掲の［4-2-2］で示したテーブルを元に、ユーザーごとの販売個数の合計値、平均値、最大値、最小値をまとめて取得してみましょう。[user_id] フィールドでグループ化し、[qty] をSUM関数、AVG関数、MAX関数、MIN関数で集計したうえで、それぞれの結果に別名を付けます。

実行するSQL文は、以下の［4-2-6］の通りです。

4-2-6 合計値、平均値、最大値、最小値を取得する

```
1  SELECT user_id
2  , SUM(qty) AS sum_qty
3  , AVG(qty) AS avg_qty
4  , MAX(qty) AS max_qty
5  , MIN(qty) AS min_qty
6  FROM sample.small_users
7  GROUP BY user_id
```

結果テーブル

行	user_id	sum_qty	avg_qty	max_qty	min_qty
1	1	9	3.0	6	1
2	2	8	4.0	6	2
3	3	6	2.0	4	0

合計値、平均値、最大値、最小値を別名を付けたフィールドで取得できた

上記の結果テーブルからは、次に挙げることが分かります。

▶SUM関数

1. [user_id] が「1」のユーザーが購入した個数は「1」「2」「6」で、[sum_qty] の値は「9」
2. [user_id]が「2」のユーザーが購入した個数は「6」「null」「2」だが、[sum_qty] の値は「8」。つまり「null」は合計に入らない

▶AVG関数

1. [user_id] が「1」のユーザーが購入した個数は「1」「2」「6」で、[avg_qty] の値は9÷3=「3.0」
2. [user_id] が「2」のユーザーが購入した個数は「6」「null」「2」だが、[avg_qty] の値は8÷2=「4.0」。つまり「null」は分母に入らない
3. [user_id] が「3」のユーザーが購入した個数は「0」「2」「4」で、[avg_qty] の値は6÷3で「2.0」。つまり「0」は分母に入る

▶MAX関数

1. [user_id] が「1」のユーザーが購入した個数は「1」「2」「6」で、[max_qty] の値は「6」
2. [user_id] が「2」のユーザーが購入した個数は「6」「null」「2」で、[max_qty] の値は「6」。つまり「null」は対象にならない
3. [user_id] が「3」のユーザーが購入した個数は「0」「2」「4」で、[max_qty] の値は「4」

▶MIN関数

1. [user_id] が「1」のユーザーが購入した個数は「1」「2」「6」で、[min_qty] の値は「1」
2. [user_id] が「2」のユーザーが購入した個数は「6」「null」「2」で、[min_qty] の値は「2」。つまり「null」は対象にならない
3. [user_id] が「3」のユーザーが購入した個数は「0」「2」「4」で、[min_qty] の値は「0」

グループ化せずに集計関数を利用する

GROUP BYを利用しない、つまり特定のフィールドでグループ化せずに、集計関数を使って各種集計値を取得することもできます。その場合、テーブル全体が対象となります。

例えば、次ページのSQL文 [4-2-7] を見てください。テーブル全体におけるレコード数、[product] フィールドの値の個数、[product] の固有の値の

個数、[qty] の合計値、平均値、最大値、最小値のそれぞれを、別名を付けたフィールドで取得します。

このテクニックは、SQL文の実行対象となるテーブルに、データが適切に格納されているかのアタリを付けたいときに役立ちます。結果テーブルを見ると、レコード数や合計値・平均値などから、テーブルのおおまかな全体像を捉えられるのではないでしょうか。

4-2-7 グループ化せずに各種集計値を取得する

```
1  SELECT COUNT(*) AS count_row
2  , COUNT(product) AS count_product
3  , COUNT(DISTINCT product) AS count_unique_product
4  , SUM(qty) AS sum_qty
5  , AVG(qty) AS avg_qty
6  , MAX(qty) AS max_qty
7  , MIN(qty) AS min_qty
8  FROM sample.small_users
```

結果テーブル

行	count_row	count_product	count_unique_product	sum_qty	avg_qty	max_qty	min_qty
1	9	7	3	23	2.875	6	0

テーブル全体を対象とした各種集計値を取得できた

もう1つ、例を挙げましょう。[order_date] フィールドに、2019年の年間データが格納されているはずの [small_date] テーブルがあるとします。

このテーブルに2019年以外のデータが紛れ込んでいないかをすばやく確認するには、次ページの [4-2-8] のようなSQL文が有効です。もっとも古い日付（最小値）ともっとも新しい日付（最大値）を探し出し、意図通りのデータになっているかが分かります。

4-2-8 最大値と最小値をすばやく取得する

```
SELECT MIN(order_date), MAX(order_date)
FROM sample.small_date
```

グループ化と集計を絞り込みや並べ替えと同時に使う

GROUP BYによるグループ化と集計を、これまで学んだSQLの句と組み合わせて、どのようなことができるかを見ていきましょう。

以下のSQL文［4-2-9］では、演習用の［customers］テーブルを1990年代以前に生まれたユーザーに絞り込んだうえで、都道府県でグループ化し、それぞれの都道府県別でユーザー数をカウントしてから、ユーザー数の多い順に並べ替えて取得しています。

4-2-9 グループ化と集計関数をほかの句と組み合わせる①

```
SELECT prefecture, COUNT(DISTINCT user_id) AS users
FROM sample.customers
WHERE birthday <= "1999-12-31"
GROUP BY prefecture
ORDER BY 2 DESC
```

結果テーブル

行	prefecture	users
1	東京	103
2	神奈川	24
3	大阪	21
4	埼玉	20
5	千葉	19

WHERE句で絞り込んだうえで、[prefecture]でグループ化した［user_id］のユニークな数（ユーザー数）を取得できた

意図通り、[prefecture] フィールドの値ごとにグループ化され、[user_id] の固有の値の個数を集計できました。みなさんもBigQueryで実行してもらうと、同じ結果テーブルが表示されると思います。

このとき、**WHERE句、GROUP BY句、ORDER BY句の順序を間違えるとエラー**になります。最初のうちはGROUP BY句を最後に記述してしまうこともあると思いますが、たくさんのSQL文を記述するうちに身についていきます。章末の確認ドリルを解くなどして慣れていってください。

もう1つ、組み合わせの例を見てみましょう。以下のSQL文 [4-2-10] では、[sales] テーブルを対象に、定価で販売された商品ID「1」「2」だけに絞り込んだうえで、[user_id] と [product_id] の2つのフィールドでグループ化し、さらに [revenue] の平均値が多い順に並べ替えたうえで、上位5レコードだけを取得しています。

4-2-10 グループ化と集計関数をほかの句と組み合わせる②

```
SELECT user_id, product_id, AVG(revenue) AS avg_revenue
FROM sample.sales
WHERE is_proper IS TRUE AND product_id IN (1,2)
GROUP BY user_id, product_id
ORDER BY avg_revenue DESC
LIMIT 5
```

結果テーブル

行	user_id	product_id	avg_revenue
1	19156	2	8800.0
2	14328	1	8000.0
3	17033	2	6600.0
4	17264	2	6600.0
5	13012	2	6600.0

WHERE句で絞り込んだレコードを対象に、2つのフィールドでグループ化した [revenue] の平均値を取得できた

複数のフィールドでグループ化したいときは、基本的にそれらのフィールドをSELECT句とGROUP BY句の両方に記述することがポイントです。

なお、結果テーブルはSELECT句に記述するフィールドの順序に従って表示されますが、GROUP BY句に記述するフィールドの順序は結果に影響しません。以下の記述方法は、どちらも同じ結果になります。

```
GROUP BY user_id, priduct_id
GROUP BY product_id, user_id
```

STEP UP

高度な集計を可能にする「統計集計関数」

本節で紹介した集計関数よりも高度な集計関数として、CHAPTER 7でも登場する「統計集計関数」に分類される関数群があります。代表的なものは、データの散らばりの度合いを表す指標の1つである「標準偏差」を求める関数で、以下の表［4-2-11］に記載しました。

SQL文における利用方法としては、SUM関数やAVG関数と同じく関数名として「STDDEV_POP」や「STDDEV_SAMP」を記述して、半角カッコ内にフィールド名を指定します。

4-2-11 標準偏差を求める関数

構文	機能
STDDEV_POP(フィールド名)	データの全数である母集団から標準偏差を求める
STDDEV_SAMP(フィールド名)	母集団から一部を取得したサンプルから標準偏差を求める

ここでは演習用の［sales］テーブルを対象に、STDDEV_POP関数を利用して標準偏差を取得してみます。例えば、［product_id］フィールドの値が「1」と「2」の商品では、どちらも1回の注文で1個だけ売れたり、2個売れたり、4個売れたりと、購入された個数に散らばりがあります。このとき、どちらの商品の個数のほうが散らばりの度合いが大きいのかを、標準偏差で求められます。

SQL文と結果テーブルは以下の［4-2-12］の通りです。

4-2-12 母集団から標準偏差を求める

```
1  SELECT product_id
2  , STDDEV_POP(quantity) AS stddev_qty
3  FROM sample.sales
4  WHERE product_id IN (1, 2)
5  GROUP BY product_id
```

結果テーブル

行	product_id	stddev_qty
1	1	0.5759250626978513
2	2	0.5509388450144583

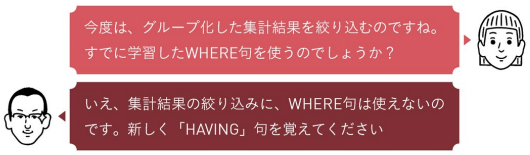
［product_id］ が「1」「2」の商品について、販売個数の標準偏差が求められた

「1」の商品のほうが、販売個数のバラつきがやや大きいということが分かります。もし、個数の分布が正規分布に近ければ、「平均±（2×標準偏差）」に約95%のデータが含まれるため、極端に高額な注文を見つけたいときなどに利用できます。

例えば、平均が1.5個で、標準偏差が0.5個だった場合、1.5－2×0.5＝0.5個、1.5＋2×0.5＝2.5個となり、データの95%が0.5個から2.5個の間に分布します。すると、3個購入された注文があった場合、極めて珍しいケースだという判断につながります。

SECTION

4-3 集計結果の絞り込み



集計結果の絞り込みにWHERE句は使えない

元のテーブルを特定のフィールドでグループ化し、合計や個数などを集計した結果を「値が○以上」などの条件で絞り込みたいことがあります。　結論から言うと、グループ化したフィールドの絞り込みにWHEREは使えません。ここで学ぶ「HAVING（ハビング）」という命令を利用します。

具体例で説明しましょう。以下の［4-3-1］は、演習用の［customer］テーブルを［prefecture］フィールドでグループ化し、［user_id］に格納されている固有の値の個数を集計するSQL文と結果テーブルです。

4-3-1 グループ化して固有の値の個数を取得する

```
1  SELECT prefecture, COUNT(DISTINCT user_id) AS users
2  FROM sample.customers
3  GROUP BY prefecture
4  ORDER BY 2 DESC
```

結果テーブル

行	prefecture	users
1	東京	113
2	神奈川	27
3	大阪	23
4	千葉	22
5	埼玉	21
6	北海道	19

[prefecture] フィールドでグループ化し、COUNT 関数で固有の値の個数を集計した値を取得できた

集計値は別名を付けた [users] フィールドに格納し、降順で並べ替えている

この結果テーブルに対して、[users] フィールドの値が「20」以上の都道府県だけに絞り込むことを考えます。CHAPTER 3で学んだWHEREを使って、以下の［4-3-2］、または［4-3-3］のSQL文を記述すればよいと思うかもしれません。しかし、実際にはいずれもエラーとなります。

4-3-2 集計結果を絞り込む（誤用例①）

```
SELECT prefecture, COUNT(DISTINCT user_id) AS users
FROM sample.customers
GROUP BY prefecture
WHERE users >= 20
ORDER BY 2 DESC
```

4-3-3 集計結果を絞り込む（誤用例②）

```
SELECT prefecture, COUNT(DISTINCT user_id) AS users
FROM sample.customers
WHERE users >= 20
GROUP BY prefecture
ORDER BY 2 DESC
```

SQL文［4-3-2］では4行目にWHERE句がありますが、「WHERE句を書く場所はここではありません」という文法エラー（Syntax error）が表示されます。WHERE句とGROUP BY句を同時に使うことは可能ですが、WHERE句を先に記述する、という決まりがあるからです。

しかし、SQL文［4-3-3］のようにWHERE句をGROUP BY句よりも先に記述しても、今度は［users］フィールドを認識できない（Unrecognized name）というエラーが表示されます。その理由は、SQLの内部的な処理の順序として、**WHERE句で指定した命令が、GROUP BY句によるグループ化よりも先に行われる**からです。

WHERE句が実行された時点では、まだグループ化が行われていないため［users］フィールドが存在せず、エラーとなってしまいます。

HAVING句で条件を指定して集計結果を絞り込む

そこで利用するのがHAVINGです。［4-3-1］の結果テーブルを、［users］フィールドの値が「20」以上の都道府県だけに絞り込む正しいSQL文は、以下の［4-3-4］となります。

4行目のHAVING句で「users >= 20」と、絞り込むための条件を指定しています。実際にクエリエディタで実行すると、同じ結果テーブルが取得できるはずです。

4-3-4 集計結果を絞り込む

```
1  SELECT prefecture, COUNT(DISTINCT user_id) AS users
2  FROM sample.customers
3  GROUP BY prefecture
4  HAVING users >= 20
5  ORDER BY 2 DESC
```

結果テーブル

行	prefecture	users
1	東京	113
2	神奈川	27
3	大阪	23
4	千葉	22
5	埼玉	21

［users］フィールドの値が「20」以上の［prefecture］に絞り込めた

とはいえ、WHERE句の代わりとしてHAVING句を使うわけではなく、同時に使うこともできます。例えば、**WHERE句で絞り込んだ後にグループ化し、その集計結果をHAVING句で絞り込む**、といった処理は成立します。以下のSQL文［4-3-5］を見てください。

3行目にWHERE句があり、[gender］フィールドの値が「2」(女性）に合致するレコードだけに絞り込んでいます。その次の4行目でグループ化を実行し、5行目のHAVING句で［user］フィールドの値が「20」以上という条件を指定しました。「都道府県別の女性顧客数を取得し、20人以上に絞り込んで表示する」という、意図通りの結果テーブルが取り出せています。

4-3-5 WHERE句とHAVING句を併用する

```
1  SELECT prefecture, COUNT(DISTINCT user_id) AS users
2  FROM sample.customers
3  WHERE gender = 2
4  GROUP BY prefecture
5  HAVING users >= 20
6  ORDER BY 2 DESC
```

結果テーブル

行	prefecture	users
1	東京	94
2	神奈川	22

［gender］フィールドで絞り込んだ後にグループ化し、［users］フィールドの値が「20」以上の［prefecture］に絞り込めた

STEP UP

SQLの各句が実行される順序

グループ化した集計結果をWHERE句で絞り込めない理由として、「SQLの内部的な処理の順序」を挙げました。これまでに学んだSELECT、FROM、ORDER BY、LIMIT、WHERE、GROUP BY、HAVINGの各句が実行される順序をまとめると、以下の図［4-3-6］のようになります。

これを丸暗記する必要はありませんが、「なぜ、このSQL文がエラーになるのか？」と疑問に思ったときに参照すると、その理由が分かることも多いものです。参考にしてください。

4-3-6 各句の実行順序

FROM	SQLを実行するテーブルを特定する
↓	
WHERE	条件に照らし、どのレコードを対象とするかを決める
↓	
GROUP BY	グループ化が指定されていれば、グループ化する
↓	
HAVING	条件に照らし、どのグループを取得対象とするかを決める
↓	
SELECT	取得する列を決める
↓	
ORDER BY	表示する順序を決める
↓	
LIMIT	表示するレコード数を決める

SECTION

4-4 柔軟なグループ化

データ分析の実務では、これまでの例よりも、もっと柔軟に分類したいことがあります。例えば、都道府県別のデータを都道府県そのものではなく、「一都三県」といった括りで分類したいケースです。

30歳から39歳の顧客を「30代顧客」として、販売金額の平均を取得したいといったケースもありそうです。どのようにするのですか？

「IF」や「CASE」などの条件式に合致するかどうかで、元の値を別の値に「変形」し、変形した値を格納するフィールドでグループ化します。

値そのものではなく、任意の分類でグループ化する

本章ではここまで、SQLによる分析、つまり「分類して、比較する」ことの「分類」がGROUP BY句で行えること、「比較」は関数を利用した合計や平均などの集計値で行えることを説明してきました。そして、「分類」は基準としたフィールドの値によって行いました。

しかし、実務上の分析においては、これまでに学んだグループ化よりも、もっと柔軟に（自由に）グループ化したい場面があります。

例えば、次ページの［4-4-1］では、[prefecture] フィールドの値を「東京」「大阪」「千葉」「神奈川」といった値そのもので分類しています。そうではなく、［4-4-2］の「一都三県」と「一都三県以外」のように、分析意図にあわせたグループで比較したいケースもあるはずです。

ほかにも、[birthday] フィールドの値に着目し、顧客を「平成以降生まれ」と「昭和以前生まれ」といった分類で比較したいケースもあるでしょう。このように、テーブルの各フィールドに格納されている値そのものでグループ化するのではなく、一定の条件に基づき、任意の分類でグループ化することを、本節では「柔軟なグループ化」と表現しています。

4-4-1 値そのものでのグループ化

user_id	prefecture
1	東京
2	千葉
3	大阪
4	東京
5	神奈川
6	大阪
7	東京

→

prefecture	users
東京	3
大阪	2
千葉	1
神奈川	1

[prefecture] フィールドでグループ化した

[prefecture] の値そのもの（都道府県）で分類される

4-4-2 任意の分類でのグループ化

user_id	prefecture
1	東京
2	千葉
3	大阪
4	東京
5	神奈川
6	大阪
7	東京

→

prefecture	users
一都三県	5
一都三県以外	2

任意の分類でグループ化した

「東京」「千葉」「神奈川」は「一都三県」に分類するなど、柔軟なグループ化が可能になる

2つの分類でグループ化する

こうした自由なグループ化のために利用するのが、本節で学ぶ「IF（イフ）」と「CASE（ケース）」の構文です。まず、2つの分類でグループ化するIFの構文は、次ページの［4-4-3］のようになります。ExcelのIF関数と似ているので、スムーズに理解できるのではないでしょうか。

4-4-3 IFの構文

```
1  IF（条件式，TRUEの場合の値，FALSEの場合の値）
```

このIF文とGROUP BY句、集計関数を組み合わせることで、任意のグループで集計した値を結果テーブルとして取得し、比較できるようになります。以下のSQL文［4-4-4］では、［customers］テーブルの［birthday］フィールドを利用して「平成以降生まれ」と「昭和以前生まれ」にグループ化し、それぞれの固有の値の個数を集計することで顧客数を比較しています。

2〜3行目にあるIF文に注目してください。「birthday >= “1989-01-08”」を条件式とし、その結果が真（TRUE）の場合は「平成以降生まれ」、偽（FALSE）の場合は「昭和以前生まれ」に分類するよう記述しています。

この結果はAS句により、「era」という別名を付けたフィールドに格納されます。さらに、6行目にあるGROUP BY句で「era」を指定し、グループ化が実行されるという仕組みです。

4-4-4 任意の2グループで集計する

```
1  SELECT
2  IF (birthday >= "1989-01-08"
3  , "平成以降生まれ", "昭和以前生まれ") AS era
4  , COUNT(DISTINCT user_id) AS users
5  FROM sample.customers
6  GROUP BY era
```

結果テーブル

行	era	users
1	昭和以前生まれ	307
2	平成以降生まれ	190

［era］フィールドの2グループに分類し、集計値を取得できた

IF文の条件式には、SECTION 3-3で学んだ演算子であるAND（かつ）とOR（または）も利用できます。

以下のSQL文［4-4-5］では、[prefecture] フィールドを対象とした条件式をORでつなげることにより、「東京」「神奈川」「千葉」「埼玉」のいずれかに合致するなら「一都三県」、それ以外に合致するなら「一都三県以外」にグループ化するように指定しています。

4-4-5 任意の2グループで集計する（ORを利用）

```
1  SELECT
2  IF (prefecture = "東京"
3  OR prefecture = "神奈川"
4  OR prefecture = "埼玉"
5  OR prefecture = "千葉", "一都三県", "一都三県以外")
6  AS pref_group
7  , COUNT(DISTINCT user_id) AS users
8  FROM sample.customers
9  GROUP BY pref_group
```

結果テーブル

行	pref_group	users
1	一都三県以外	314
2	一都三県	183

［pref_group］フィールドの２グループに分類し、集計値を取得できた

上記ではあえてORを利用し、複数条件が指定できることを示しましたが、たくさんの条件をORでつなぐのは大変です。これを回避するため、SECTION 3-3で学んだIN演算子を使って「IF((prefecture IN ("東京", "神奈川", "埼玉", "千葉")) IS TRUE, "一都三県", "一都三県以外")」としてもよいでしょう。

さらに、同じくSECTION 3-3で学んだIS NULL演算子を、IF文の条件式に利用することもできます。以下のSQL文［4-4-6］では、誕生日が「null」かどうかをIS NULL演算子で判定し、「未登録」「登録済」に分類します。

4-4-6 任意の2グループで集計する（IS NULLで判定）

```
SELECT
IF (birthday IS NULL, "未登録", "登録済")
AS birthday_regist
, COUNT(DISTINCT user_id) AS users
FROM sample.customers
GROUP BY birthday_regist
```

結果テーブル

行	birthday_regist	users
1	登録済	459
2	未登録	38

［birthday_regist］フィールドの2グループに分類し、集計値を取得できた

3つ以上の分類でグループ化する

続いて、CASEの構文を見ていきましょう。IF文は「条件に当てはまればA、当てはまらなければB」という2グループでの分類でしたが、CASE文は「条件1に当てはまればA、条件2に当てはまればB、条件3に当てはまればC、どれにも当てはまらなければD」というふうに、3グループ以上での分類に利用します。

CASEの構文には、次ページに示した2通りがあることを覚えてください。［4-4-7］では条件を判定したいフィールド名をCASEに続けて記述していますが、［4-4-8］ではフィールド名を使った条件式をWHENの後に記述している、という違いがあります。

4-4-7 CASEの構文①

```
CASE グループ化するフィールド名
WHEN 条件を判定したい値 THEN 該当した場合の値
WHEN 条件を判定したい値 THEN 該当した場合の値
WHEN 条件を判定したい値 THEN 該当した場合の値
ELSE いずれにも該当しない場合の値
END
```

4-4-8 CASEの構文②

```
CASE
WHEN 判定したい条件式 THEN 該当した場合の値
WHEN 判定したい条件式 THEN 該当した場合の値
WHEN 判定したい条件式 THEN 該当した場合の値
ELSE いずれにも該当しない場合の値
END
```

CASE文の利用例として、[prefecture] フィールドの値が「東京」「神奈川」であれば「関東主要都県」、「大阪」「京都」「兵庫」であれば「関西主要府県」、それ以外の都道府県であれば「それ以外」と、合計3つのグループに分類するケースを考えてみます。

次ページの［4-4-9］は構文①、［4-4-10］は構文②で記述しています。いずれの構文で記述しても、結果テーブルは［4-4-9］のSQL文の下に示した結果テーブルと同じです。[prefecture_group] と別名を付けたフィールドに3つのグループが行として存在し、[user_id] の固有の値の個数を集計した [users] フィールドの降順で並んでいることが分かります。

4-4-9 都道府県を任意の3グループで集計する（構文①）

```
SELECT
CASE prefecture
WHEN "東京" THEN "関東主要都県"
WHEN "神奈川" THEN "関東主要都県"
WHEN "大阪" THEN "関西主要都県"
WHEN "京都" THEN "関西主要都県"
WHEN "兵庫" THEN "関西主要都県"
ELSE "その他"
END AS prefecture_group
, COUNT(DISTINCT user_id) AS users
FROM sample.customers
GROUP BY prefecture_group
ORDER BY 2 DESC
```

結果テーブル

行	prefecture_group	users
1	その他	320
2	関東主要都県	140
3	関西主要都県	37

[prefecture_group] フィールドの3グループに分類し、集計値を取得できた

4-4-10 都道府県を任意の3グループで集計する（構文②）

```
SELECT
CASE
WHEN prefecture = "東京" THEN "関東主要都県"
WHEN prefecture = "神奈川" THEN "関東主要都県"
WHEN prefecture = "大阪" THEN "関西主要都県"
WHEN prefecture = "京都" THEN "関西主要都県"
```

```
    WHEN prefecture = "兵庫" THEN "関西主要都県"
    ELSE "その他"
    END AS prefecture_group
    , COUNT(DISTINCT user_id) AS users
    FROM sample.customers
    GROUP BY prefecture_group
    ORDER BY 2 DESC
```

このような都道府県のグループ化では、CASE文は構文①、構文②のどちらでも成立します。よりシンプルに記述できる構文①のほうが望ましく、構文②の必要性は感じられないかもしれません。

ここで、別の例を挙げて②の構文の必要性を説明しましょう。例えば、[birthday] フィールドの値が2000年1月1日以降生まれなら「2000年代生まれ」、1990年1月1日以降生まれなら「1990年代生まれ」、1980年1月1日以降生まれなら「1980年代生まれ」、それより昔なら「1970年代以前生まれ」と分類するケースを考えてみましょう。

構文①で記述すると以下の［4-4-11］となりますが、これは3〜6行目にわたって文法エラーとなります。一方、構文②で記述すると［4-4-12］となり、意図した通りの結果テーブルが返ってきます。

4-4-11 誕生日を任意の3グループで集計する（誤用例）

```
    SELECT
    CASE birthday
    WHEN >= "2000-01-01" THEN "2000年代生まれ"
    WHEN >= "1990-01-01" THEN "1990年代生まれ"
    WHEN >= "1980-01-01" THEN "1980年代生まれ"
    ELSE "1970年代以前生まれ"
    END AS era_group
```

```
8   , COUNT(DISTINCT user_id) AS users
9   FROM sample.customers
10  GROUP BY era_group
11  ORDER BY 2 DESC
```

4-4-12 誕生日を任意の3グループで集計する

```
1   SELECT
2   CASE
3   WHEN birthday >= "2000-01-01" THEN "2000年代生まれ"
4   WHEN birthday >= "1990-01-01" THEN "1990年代生まれ"
5   WHEN birthday >= "1980-01-01" THEN "1980年代生まれ"
6   ELSE "1970年代以前生まれ"
7   END AS era_group
8   , COUNT(DISTINCT user_id) AS users
9   FROM sample.customers
10  GROUP BY era_group
11  ORDER BY 2 DESC
```

結果テーブル

行	era_group	users
1	1970年代以前生まれ	222
2	1990年代生まれ	172
3	1980年代生まれ	95
4	2000年代生まれ	8

[era_group] フィールドの4グループに分類し、集計値を取得できた

CASEの構文①と構文②の使い分けは「どのような条件式にするか」によって決まります。端的に言うと、**単純な「等しい」以外の条件では構文②となる**のですが、次のように考えるといいでしょう。

●特定の値と等しい
　⇒構文①を利用する

●特定の値より大きい・小さい
●複数の特定の値に合致する（IN演算子で記述）
●特定の2つの値の間にある（BETWEEN演算子で記述）
　⇒構文②を利用する

ご覧の通り、構文②のほうが柔軟に分類できるため、実務上は構文②を利用することが多いです。

STEP UP

CASE文の条件式は順序も重要

CASE文の例として、誕生日を基準に「2000年代生まれ」などのグループに分類する方法を紹介しました。このCASE文は構文②に従い、WHENに続く条件式を「birthday >= "2000-01-01"」のように記述しますが、その順序にも注意する必要があります。

以下の［4-4-13］はCASE文だけを抜粋したものです。2〜5行目の条件式が、正しい順序で記述されています。

4-4-13 正しい順序で条件式を記述したCASE文

```
1  CASE
2  WHEN birthday >= "2000-01-01" THEN "2000年代生まれ"
3  WHEN birthday >= "1990-01-01" THEN "1990年代生まれ"
4  WHEN birthday >= "1980-01-01" THEN "1980年代生まれ"
5  ELSE "1970年代以前生まれ"
6  END
```

CASE文の条件式は上から順に判定され、合致しない場合、次の条件式で判定されます。[4-4-13]では、2行目の条件式に合致したら「2000年代生まれ」、合致しなかったら3行目の条件式が判定され、合致すれば「1990年代生まれ」、合致しなければ4行目の条件式が判定され……となります。

そのため、年代別にグループ化したいとき、条件式も年代の順序で記述しておかないと、正しい結果が得られなくなります。例えば、以下の[4-4-14]は、条件式が誤った順序で指定されています。

4-4-14 誤った順序で条件式を記述したCASE文

```
CASE
WHEN birthday >= "2000-01-01" THEN "2000年代生まれ"
WHEN birthday >= "1980-01-01" THEN "1980年代生まれ"
WHEN birthday >= "1990-01-01" THEN "1990年代生まれ"
ELSE "1970年代以前生まれ"
END
```

3行目と4行目が逆になっていることに気づいたでしょうか。

1990年1月1日生まれのユーザーは、正しくは「1990年代生まれ」としてグループ化されるべきです。しかし、[4-4-14]のCASE文では、4行目の条件式を判定する前に、3行目の条件式に合致してしまい、「1980年代生まれ」としてグループ化されてしまいます。

このようなミスを避けるために、**WHENに続く条件式は「古い順、または新しい順で記述する」**と覚えておいてください。誕生日だけでなく、販売金額などの連続値を判定するときも同様です。

SECTION

4-5 確認ドリル

問題 006

[customers] テーブルに誕生日が同じユーザーが何人いるか調べてください。結果テーブルは誕生日（birthday）とその誕生日の顧客数（users）の2カラムとし、1つの誕生日に2人以上の顧客がいるレコードに絞り込んで表示してください。

問題 007

[web_log] テーブルから、ユーザーID（user_id）別のページビュー数（pageviews）の合計値を取得してください。ただし、そのページビューはemail経由で発生したものだけとします。結果テーブルは［user_id］と［pageviews］の2カラムとし、[pageviews] が「10」以上のユーザーのみを含めます。[user_id] が「null」のレコードは除外してください。[web_log] テーブルは、1レコードが1つの［pageview］で構成されています。

問題 008

[customers] テーブルから、プレミアム顧客がもっとも多い都道府県とプレミアム顧客数を求めてください。結果テーブルは、都道府県名（prefecture）とプレミアム顧客数（users）の2カラムとします。

問題 009

[web_log] テーブルからユーザー別の訪問回数（number_of_visits）を取得してください。結果テーブルは［cid］と［number_of_visits］の2カラムとし、訪問回数が多い順に3ユーザーに絞り込んで表示します。

問題 010

［products］テーブルから、仕入金額（cost）ごとの商品アイテム数を取得してください。仕入金額は「0円以上300円未満」「300円以上600円未満」……と、300円刻みで「900円以上1,199円未満」までを分類して［cost_range］とし、アイテム数は［items］とします。並べ替えは［cost_range］の小さい順としてください。

解答

解答 006

```
1  SELECT birthday, COUNT(*) AS users
2  FROM sample.customers
3  WHERE birthday IS NOT NULL
4  GROUP BY birthday
5  HAVING users > 1
```

結果テーブル

行	birthday	users
1	1986-05-03	2
2	1974-02-10	2

［birthday］でグループ化して「COUNT(*)」でレコード数を調べます。もちろん「COUNT(DISTINCT user_id)」でも正解ですが、1レコードが1ユーザーに対応しているテーブルであれば、「COUNT(*)」でも結果は同じです。

解答 007

```
SELECT user_id, count(*) AS pageviews
FROM sample.web_log
WHERE media = "email" AND user_id IS NOT NULL
GROUP BY user_id
HAVING pageviews >= 10
```

結果テーブル

行	user_id	pageviews
1	16816	10
2	11026	13

［media］を「email」に絞り込むには、3行目のようにWHERE句を使います。表示するレコードを「pageviews >= 10」を利用して絞り込むには、5行目のようにHAVING句を使います。

別解として、以下の通り［media］と［pageview］の両方をHAVING句で絞り込む方法があります。最初にメディアで絞り込んでいるので、余計な集計をしないぶん、最初のSQL文のほうが優れています。

```
SELECT user_id, media, count(*) AS pageviews
FROM sample.web_log
GROUP BY user_id, media
HAVING media = "email"
AND pageviews >= 10
AND user_id IS NOT NULL
ORDER BY 3 DESC
```

結果テーブル

行	user_id	media	pageviews
1	11026	email	13
2	16816	email	10

解答 008

```sql
SELECT prefecture, COUNT(DISTINCT user_id) AS users
FROM sample.customers
WHERE is_premium IS TRUE
GROUP BY prefecture
ORDER BY 2 DESC
LIMIT 1
```

結果テーブル

行	prefecture	users
1	東京	19

［prefecture］フィールドでグループ化する必要があるので、SELECT句とGROUP BY句に［prefecture］を指定します。また、ユニークな顧客数（user_id）は「COUNT(DISTINCT user_id)」で調べます。絞り込み条件の「プレミアム顧客に一致」は、3行目のWHERE句としてFROM句に続けて記述します。

最終行で「LIMIT 1」と記述していますが、まずは並べ替えが正しいかを確認するためにLIMIT句なしで実行し、その後に付与すると安全です。

解答 009

```
SELECT cid
, MAX(session_count) - MIN(session_count) + 1
AS number_of_visits
FROM sample.web_log
GROUP BY cid
ORDER BY number_of_visits DESC
LIMIT 3
```

結果テーブル

行	cid	number_of_visits
1	1401531153.1531099940	19
2	1306791679.1535448230	17
3	1454236587.1542348510	15

［cid］でグループ化する必要があることは、容易に理解できると思います。訪問回数は「MAX(session_count)」と「MIN(session_count)」の集計値を引き、それに「1」を足して求めました。

なぜそれが解答となるのか説明が必要だと思いますので、以下のSQL文を実行してみてください。［session_count］の最大値（max）・最小値（min）と、訪問回数（visits）の関係が結果テーブルから分かります。

```
SELECT cid
, MIN(session_count) AS min
, MAX(session_count) AS max
, MAX(session_count) - MIN(session_count) + 1 AS visits
FROM sample.web_log
GROUP BY cid
ORDER BY 4 DESC
LIMIT 3
```

結果テーブル

行	cid	min	max	visits
1	1401531153.1531099940	3	21	19
2	1306791679.1535448230	1	17	17
3	1454236587.1542348510	1	15	15

訪問回数が「19」回ともっとも多いユーザーは、[web_log] テーブルに記録された [session_count] の最小値が「3」です。これは、ログにこのユーザーの「3回目の訪問」が記録されているということです。同様に最大値として「21回目の訪問」が記録されています。ということは、ログに記録されている期間中には19回訪問した、ということになります。

訪問回数が「17」回と2番目に多いユーザーは、初回訪問である「1」が最小値、「17」が最大値ですので、初回訪問から17回目の訪問まで、合計17回訪問した、ということになります。

解答 010

```
SELECT
CASE
WHEN cost >= 0 AND cost < 300 THEN "1.価格帯 0-299"
WHEN cost >= 300 AND cost < 600 THEN "2.価格帯 300-599"
WHEN cost >= 600 AND cost < 900 THEN "3.価格帯 600-899"
WHEN cost >= 900 AND cost < 1200 THEN "4.価格帯 900-1199"
END AS cost_range
, COUNT(DISTINCT product_id) AS items
FROM sample.products
GROUP BY cost_range
ORDER BY 1
```

別解として、CASE文の条件に「BETWEEN 値1 AND 値2」を利用する方法もあります。こちらのほうがスマートかもしれません。

```
SELECT
CASE
WHEN cost BETWEEN 0 AND 299 THEN "1.価格帯 0-299"
WHEN cost BETWEEN 300 AND 599 THEN "2.価格帯 300-599"
WHEN cost BETWEEN 600 AND 899 THEN "3.価格帯 600-899"
WHEN cost BETWEEN 900 AND 1199 THEN "4.価格帯 900-1199"
END AS cost_range
, COUNT(DISTINCT product_id) AS items
FROM sample.products
GROUP BY cost_range
ORDER BY 1
```

結果テーブル

行	cost_range	items
1	1.価格帯 0-299	2
2	2.価格帯 300-599	8
3	3.価格帯 600-899	3
4	4.価格帯 900-1199	2

これらのCASE文では、3～6行目のWHEN句で指定した条件のどこかに、すべての［product_id］が当てはまります。しかし、もし1,200円以上の原価（cost）の［product_id］がテーブルに存在した場合、どの条件にも当はまらないことになります。その場合、［cost_range］の「null」にその［product_id］が分類されます。

この問題では［cost］の最大値が1,000円となるため割愛しましたが、「null」が現れるのは一般的に好ましいことではないため、通常はELSE句を含めるのが作法です。上記では、6行目と7行目の間に「ELSE "5.その他の価格帯"」を挿入します。その場合でも、結果テーブルに「5.その他の価格帯」が出現することはなく、同一の結果となります。

なお、この問題では、仕入金額を［cost_range］という4つのカテゴリに分類し、そのカテゴリに該当する［product］などの数（商品アイテム数）を取得しました。こうした表を「度数分布表」と呼び、統計学ではカテゴリのことを「階級」、該当する数を「度数」といいます。

階級が金額や個数などの数値で表せる大小関係がある「量的変数」に基づく場合、その中央値を「階級値」と呼びます。例えば、「0円以上300円未満」の階級の階級値は150円です。

度数分布表の階級が量的変数に基づく場合は、通常、階級値の小さい順に並べるのが作法です。階級をCASE文で分類するとき、「1. 価格帯 0-299」のように先頭に数字を入れることで、ORDER BY句で並べ替えができるように工夫しています。

また、度数分布表をBIツールに読み込み、度数を面積に対応させ、棒状のグラフを棒と棒の隙間なく視覚化したグラフを「ヒストグラム」と呼びます。この問題・解答をもって、SQLで度数分布表の作成ができること、そのデータをBIツールに読み込めば容易にヒストグラムを作成できることを理解してもらえるかと思います。

本書の例ではテーブルが小さいのでメリットを感じにくいですが、BigQueryは結果を返すスピードが速いので、重い処理はBigQuery上のSQLで、訴求力の高いチャートはTableauなどのBIツールで、という使い分けができると、分析者としての引き出しが増えるでしょう。

CHAPTER 5

テーブルの結合と集合演算

5-1　複数のテーブルが存在する理由
5-2　テーブルの結合
5-3　複数条件や3テーブル以上の結合
5-4　同一テーブルの結合とnull値の置換
5-5　テーブルの集合演算
5-6　仮想テーブルの作成
5-7　確認ドリル

ここまでで学んだSQLは、1つのテーブルを対象としたものでした。しかし、実務では顧客・商品・取引といったデータの性質や生成方法に応じた複数のテーブルが存在するのが一般的です。本章では、複数のテーブルを「つなげる」ための構文や、「仮想テーブル」と通常のテーブルと組み合わせて利用する方法を学びます。

SECTION

5-1 複数のテーブルが存在する理由

データベースは通常、複数のテーブルで構成されています。時には非常にたくさんのテーブルが存在するのですが、その理由は分かりますか？

いえ……あらためて考えると分かりません。

もっとも大きな理由は「メンテナンスが容易になるから」です。本節では、さらに2つに分けて、複数のテーブルが存在する理由を見ていきましょう。

異なる性質のデータを複数のテーブルで管理

SECTION 1-2において、リレーショナルデータベースについて触れたことを覚えているでしょうか。SQLの実行対象となるデータベースは、複数のテーブルが関係性を持って存在しているのが普通です。

それでは、なぜ複数のテーブルが存在しているのかというと、**単一のテーブルで扱うことと比較してメリットがある**からです。大きくは前述の通り、メンテナンスが容易になるからですが、筆者としては、さらに次の2つの理由に分解できると考えています。

1. 異なる性質のデータを扱いやすい
2. 一定期間ごとに集計されたデータを扱いやすい

理由1について掘り下げましょう。例えば、ECサイトの販売データベースでは、格納すべきデータは次のような性質に分かれているはずです。

●誰が買ったのか？　⇒顧客データ（氏名や住所など）
●何を買ったのか？　⇒商品データ（商品名やカテゴリなど）
●いくらで買ったのか？　⇒取引データ（販売金額や割引など）

これらのデータを「大きな単一のテーブル」としてまとめると、以下の［5-1-1］のようになります。顧客・商品・取引に関するすべてのデータが、1つのテーブルにまとまっていることが分かるでしょう。このような大きな単一のテーブルも、存在し得ないわけではありません。

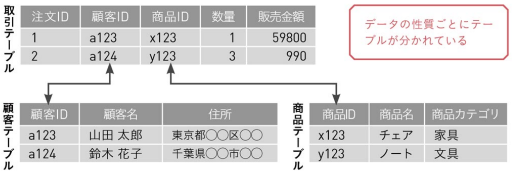
一方、顧客・商品・取引というデータの性質ごとにテーブルが分かれていると、その下の［5-1-2］のようなイメージになります。

5-1-1 単一の大きなテーブル

すべてのデータが1つのテーブルにまとまっている

注文ID	顧客ID	顧客名	住所	商品ID	商品名	商品カテゴリ	数量	販売金額
1	a123	山田 太郎	東京都〇〇区〇〇	x123	チェア	家具	1	59800
2	a124	鈴木 花子	千葉県〇〇市〇〇	y123	ノート	文具	3	990

5-1-2 性質が異なる複数のテーブル



ここで留意したいのが、**テーブルが複数に分かれていても、共通するフィールドを利用して、1つのテーブルにつなげられる**ということです。

［5-1-2］の矢印に注目してください。顧客ごとの取引に関するデータを参照したい場合、［顧客ID］という共通のフィールドを利用すれば、取引テーブルと顧客テーブルをつなげることが可能になります。さらに［商品ID］を利用して商品テーブルもつなげれば、顧客・商品・取引に関係するすべてのデータがまとまった、［5-1-1］と同じテーブルが完成します。

つまり、データの関係性が保たれていれば、複数のテーブルを大きな単一のテーブルに作り替えることは難しくありません。そもそも、性質が異なるデータは異なるシステムで管理されていることが多いため、性質ごとに異なるテーブルで管理するほうが合理的、ということもできます。

逆に、大きな単一のテーブルのみで管理することには、明確なデメリットがあります。例えば、山田太郎さんが東京から大阪に引っ越したり、鈴木花子さんが結婚して姓が変わったりしたケースを想定してみてください。

それらの変更を単一の大きなテーブルに反映するには、山田さんと鈴木さんが登場するすべての行を修正する必要が出てきます。しかし、複数のテーブルで管理していれば、顧客テーブルの山田さんと鈴木さんの行だけを修正すればよいことになります。取引テーブルや商品テーブルにまで修正が及ぶことはないため、メンテナンスが容易になる、というわけです。

一定期間ごとのデータを複数のテーブルで管理

複数のテーブルが存在する理由**2**としては、「一定期間ごとに集計されたデータを扱いやすい」ことを挙げました。

みなさんの中には、Google広告に代表されるWeb広告のデータを扱っている人もいると思います。そうしたデータからBigQueryのテーブルを作成するときには、Google広告などの管理画面から、月別にCSVファイルをダウンロードすることが多いはずです。すると、次ページの［5-1-3］のように、月別に複数のテーブルが存在することになります。

5-1-3 月別に集計された複数のテーブル

月別のデータでテーブルが分かれている

1月テーブル

campaign_name	group_name	impression	click	cost	conversion
AAA	XX	15000	120	30000	6
AAA	YY	12000	350	150000	8
AAA	ZZ	3000	60	20000	3

2月テーブル

campaign_name	group_name	impression	click	cost	conversion
AAA	XX	12000	100	32000	7
AAA	YY	18000	320	120000	12
AAA	ZZ	6500	80	26000	5

また、システムの仕様として、テーブルが分かれて生成されることもあります。例えば、Googleアナリティクスの新バージョン「GA4」のデータはBigQueryと連携できますが、自動的に日別のテーブルが作成されます。つまり、10日間のデータであれば、BigQuery上に10個のテーブルが生成されることになります。

Google広告やGoogleアナリティクスのデータは、24時間365日、絶えず発生しています。よって、どこかでテーブルを区切らないと、巨大で扱いにくい1つのテーブルが生成されることになってしまいます。月別・日別でテーブルを分けることは、管理上、合理的なことです。

また、同じシステムから一定期間ごとに生成された複数のテーブルは、**同じスキーマで構成されているため、簡単につなげることが可能**です。日別のテーブルをつなげて月別、月別のテーブルをつなげて四半期別といったように、必要とする範囲のデータを柔軟に抽出できるようになります。

テーブルを横につなぐ、縦につなぐ

データベースに複数のテーブルが存在することは合理的であり、それらを「つなげる」ことも可能である、と解説しました。本章では、このテーブルをつなげることについて学んでいきますが、理由**1**と**2**で述べたつなぎ方に違いがあることにお気づきでしょうか？

［5-1-2］で示したつなぎ方は、横方向にフィールドが増える、いわば「横につなぐ」方法です。データベースとSQLの世界において、この横につなぐ方法をテーブルの結合、または「**JOIN**（ジョイン）」と呼びます。

対して、［5-1-3］では縦方向にレコードが増える、いわば「縦につなぐ」方法です。縦につなぐ方法は「**UNION**（ユニオン）」と呼びます。UNIONはテーブル同士の集合演算の一種で、詳しくはSECTION 5-5で学習します。

データ分析の実務においては、JOINとUNIONの両方を使うシーンが多々あります。例えば「カタログを発送した顧客IDリストが、カタログの冊数分だけのテーブルに存在するため、それらをUNIONし、カタログ発送履歴の有無別に販売金額を分析するために、UNIONした結果を販売テーブルとJOINする」といった形です。現時点ではイメージしにくいかもしれませんが、次節以降でしっかり学習してください。

「横につなぐ」「縦につなぐ」という考え方は、確かに合理的ですし、面白いですね！

複数のテーブルが存在する理由、納得できましたか？これが腑に落ちると、リレーショナルデータベースの理解が一気に進むのではないかと思います。

SECTION

5-2 テーブルの結合

テーブル同士を横につなぐJOINには、処理内容が異なる4つのタイプがあります。簡単なテーブルを例に説明するので、それぞれの処理を覚えてください。

テーブルを結合する4つのタイプ

複数のテーブルが共通して持っているフィールドを利用して、テーブルを横に結合することをJOINと呼びます。JOINには、その「共通して持つフィールド」に格納されている値の重なり、または重なりのなさをどう処理するかによって、以下の表［5-2-1］の4つ（実質的には3つ）のタイプに分かれます。順に見ていきましょう。

5-2-1 テーブル結合のタイプと処理内容

タイプ	処理内容
INNER JOIN （内部結合）	2つのテーブルが共通して持つフィールドの値が重なっているレコードのみを残す
LEFT OUTER JOIN （左外部結合）	先に指定したテーブルを「左側」、後に指定したテーブルを「右側」とし、左側は全レコード、右側は左側と値が重なっているレコードのみを残す
RIGHT OUTER JOIN （右外部結合）	先に指定したテーブルを「左側」、後に指定したテーブルを「右側」とし、右側は全レコード、左側は右側と値が重なっているレコードのみを残す
FULL OUTER JOIN （完全外部結合）	2つのテーブルにある全レコードを残す

2つのテーブルの値が重なっているレコードを残す

「INNER JOIN（インナー ジョイン）」（内部結合）は、もっとも基本的なタイプの結合です。まずは以下の［5-2-2］を見てください。JOINする前のテーブルとして「販売テーブル」と「商品マスタテーブル」があります。以降、しばらくはこの2つの小さなテーブルを例に、結合の動作を解説していきます。

この販売テーブルと商品マスタテーブルをINNER JOINで結合すると、矢印で指した下のテーブルになります。2つのテーブルには、共通して持つフィールドとして［product_id］があります。また、［product_id］の値である「B」「C」は、2つのテーブルの両方に存在します。よって、INNER JOINの結果は「B」「C」のレコードのみが残ります。

逆に、［product_id］の値の中でも、販売テーブルの「D」、商品マスタテーブルの「A」は2つのテーブルで重なっていないため、INNER JOINの結果には含まれません。このINNER JOINを実行する意図としては、「商品マスタテーブルに登録されており、かつ、実際に売れた商品のみを取得したい」と表現できるでしょう。

5-2-2 INNER JOINの処理内容

販売テーブル

product_id	qty
D	3
B	4
C	2

商品マスタテーブル

product_id	product_name
A	アジ
B	サバ
C	タコ

↓

product_id	qty	product_name
B	4	サバ
C	2	タコ

2つのテーブルをINNER JOINで結合した

INNER JOINの処理内容を理解できたところで、次は構文を覚えましょう。INNER JOINの構文は、次ページの［5-2-3］のようになります。

FROM句で指定する最初に読み込むテーブルの名前を「テーブル名①」、INNER JOIN句で読み込むテーブルの名前を「テーブル名②」としています。重要なのは、その次の4行目です。

4行目の「ON」は結合条件を表し、テーブル名①とテーブル名②が共通して持つフィールドの名前を指定します。構文の通り、テーブル名とフィールド名を「.」で区切り、それぞれを「=」でつないでください。

5-2-3 INNER JOINの構文

```
SELECT フィールド名
FROM テーブル名①
INNER JOIN テーブル名②
ON テーブル名①.テーブル名①の結合に使うフィールド名
= テーブル名②.テーブル名②の結合に使うフィールド名
```

今度は、実際のSQL文を見てみましょう。[5-2-2]の販売テーブルの名前を「small_sales_1」、商品マスタテーブルの名前を「small_master」とし、「sample」データセット内にあると仮定します。INNER JOINで結合するSQL文は、以下の[5-2-4]となります。

5-2-4 INNER JOINで結合する（ON句を利用）

```
SELECT hanbai.product_id, product_name, qty
FROM sample.small_sales_1 AS hanbai
INNER JOIN sample.small_master AS master
ON hanbai.product_id = master.product_id
```

INNER JOINを実行した、[5-2-4]のSQL文で留意すべき点を次にまとめます。記述のルールや省略可能な点などを確認してください。

- 2行目のFROM句でテーブル名に続いて記述した「AS hanbai」は、[small_sales_1]テーブルに別名として「hanbai」を付ける命令です。半角英数字を使った任意の名前（ただし、数字で始めるのは不可）を指定できます。
- 3行目のINNER JOIN句でテーブル名に続いて記述した「AS master」も同様で、[small_master]テーブルに別名として「mastar」を付ける命令です。
- 3行目のINNER JOINの「INNER」を省略して「JOIN」のみ記述することも可能です。
- AS句による別名はテーブルを短い名前で表現できるため便利で、とてもよく使われますが、省略することもできます。その場合、「データセット名.テーブル名」と「テーブルの別名」の間には半角スペースが入ります。
- 4行目のON句で記述した「hanbai.product_id = master.product_id」は、別名［hanbai］テーブルの［product_id］フィールドと、別名［master］テーブルの［product_id］フィールドの値が一致するレコードを結合させる命令です。
- 1行目のSELECT句の「hanbai.product_id」の「hanbai.」は、[small_sales_1]テーブルの［product_id］フィールドを取得することを明示するために必ず指定します。なお、「hanbai.product_id」を「master.product_id」に変えても結果は同じになります。
- 1行目のSELECT句で「hanbai.product_id」（または「master.product_id」）と指定すべきところを、別名を付け忘れて「product_id」としてしまうと、『フィールド名「product_id」は曖昧です』という主旨のエラーになります。これは、結合したテーブルには［product_id］フィールドが2つ存在するため、どちらのテーブルのものを取得するのかが曖昧になるからです。

INNER JOINの説明の最後として、別の記述方法1つ紹介しましょう。2つのテーブルが共通して持つフィールドの名前が同じである場合、ON句の代わりに「**USING**（ユージング）」という命令を使うことができます。

USING句を利用して前掲の［5-2-4］を書き換えると、次ページのSQL文

[5-2-5] となります。USINGに続いて結合するフィールド名を半角カッコで囲んで記述するだけと、ON句よりもシンプルにできるので、結合するフィールド名が2つのテーブルで同一の場合はおすすめです。

ON句を利用した場合と、USINGを利用した場合のJOIN結果の違いをイメージすると、[5-2-6] のように図示できます。ON句ではテーブルを指定しないとエラーになる理由がより理解できると思います。

5-2-5 INNER JOINで結合する（USING句を利用）

```
1  SELECT product_id, product_name, qty
2  FROM sample.small_sales_1
3  INNER JOIN sample.small_master
4  USING (product_id)
```

5-2-6 ON句とUSING句でのJOIN結果の違い

ON 句でフィールドを指定した場合

（販売テーブルの）product_id	（商品マスタテーブルの）product_id	qty	product_name
B	B	4	サバ
C	C	2	タコ

USING 句でフィールドを指定した場合

（テーブルを問わない）product_id	qty	product_name
B	4	サバ
C	2	タコ

先に指定したテーブルの値が重なっているレコードを残す

「LEFT OUTER JOIN（レフト アウター ジョイン）」（左外部結合）は、FROM句で指定する先に読み込むテーブルを「左側」、LEFT OUTER JOIN句で指定する後に読み込むテーブル

を「右側」とし、左側は全レコード、右側は左側と共通して持つフィールドの値が重なっているレコードのみを残す結合です。

結合するフィールドにおいて、左側テーブルに存在し、右側にない値のレコードはすべて「null」になります。例を挙げて説明しましょう。

以下の［5-2-7］のうち、上にある2つのテーブルは、INNER JOINの説明で使った［5-2-2］の販売テーブル（small_sales_1）、および商品マスタテーブル（small_master）と同じものです。販売テーブルを左側、商品マスタテーブルを右側としてLEFT OUTER JOINで結合すると、矢印で指した下のテーブルになります。

左側（販売テーブル）の［product_id］フィールドには「D」「B」「C」の3レコードが存在し、これらのレコードはすべて結果テーブルに残ります。ただし、「D」は右側（商品マスタテーブル）に存在しないため、右側のフィールドである［product_name］の値は「null」となります。

ただし、右側テーブルの「A」は左側に存在しないため、結果には残りません。このLEFT OUTER JOINを実行する意図は、次のように表現できるでしょう。

- 商品マスタテーブルに未登録の商品でも、販売個数はすべて取得したい。
- 商品マスタテーブルに未登録の商品が販売されていた場合、商品名が空欄（null）となっても構わない。

5-2-7 LEFT OUTER JOINの処理内容

販売テーブル

product_id	qty
D	3
B	4
C	2

商品マスタテーブル

product_id	product_name
A	アジ
B	サバ
C	タコ

product_id	qty	product_name
D	3	
B	4	サバ
C	2	タコ

2つのテーブルを LEFT OUTER JOIN で結合した

LEFT OUTER JOINの構文は以下の［5-2-8］のようになり、INNER JOINの構文で「INNER JOIN」と記述していたところを「LEFT OUTER JOIN」と書き換えるだけです。なお、LEFT OUTER JOINの「OUTER」は省略することもできます。

5-2-8 LEFT OUTER JOINの構文

```
SELECT フィールド名
FROM テーブル名①
LEFT OUTER JOIN テーブル名②
ON テーブル名①.テーブル名①の結合に使うフィールド名
= テーブル名②.テーブル名②の結合に使うフィールド名
```

実際に記述するSQL文は、以下の［5-2-9］の通りです。2つのテーブルで結合するフィールド名が同じ場合、INNER JOINのときと同様、ON句の代わりにUSING句を使うこともできます。

5-2-9 LEFT OUTER JOINで結合する

```
SELECT product_id, product_name, qty
FROM sample.small_sales_1
LEFT OUTER JOIN sample.small_master
USING (product_id)
```

後に指定したテーブルの値が重なっているレコードを残す

「RIGHT OUTER JOIN（ライト アウター ジョイン）」（右外部結合）は、FROM句で指定する先に読み込むテーブルを「左側」、RIGHT OUTER JOIN句で指定する後に読み込むテーブルを「右側」とし、右側は全レコード、左側は右側と共通して持つフィールドの値が重なっているレコードのみを残す結合です。

LEFT OUTER JOINと挙動は似ていて、左側と右側のテーブルが逆になっています。前掲の［5-2-7］と同様に、販売テーブル（small_sales_1）を左側、商品マスタテーブル（small_master）を右側とした場合、RIGHT OUTER JOINを実行する意図は次のように表現できるでしょう。

- 商品マスタテーブルに登録済みの商品だけ、販売個数を取得したい。
- 商品マスタテーブルに未登録の商品が販売されていた場合、その個数はカウントしなくてよい。

構文や実際のSQL文も、LEFT OUTER JOINの記述例に準じます。「OUTER」は省略可能で、「RIGHT JOIN」と記述することも可能です。

2つのテーブルにある全レコードを残す

「FULL OUTER JOIN（フル アウター ジョイン）」（完全外部結合）は、FROM句で指定する先に読み込むテーブルを「左側」、FULL OUTER JOIN句で指定する後に読み込むテーブルを「右側」とし、左側も右側も全レコードを残す結合です。相手方のテーブルにないフィールドの値は「null」になります。

実際の記述例は次ページの［5-2-10］の通りです。LEFT OUTER JOINおよびRIGHT OUTER JOINと同じく「OUTER」は省略可能で、「FULL JOIN」と記述してもSQL文は成立します。

5-2-10 FULL OUTER JOINで結合する

```
1  SELECT * FROM sample.small_sales_1
2  FULL OUTER JOIN sample.small_master
3  USING (product_id)
```

結果テーブル

行	product_id	qty	product_name
1	A	*null*	アジ
2	B	4	サバ
3	C	2	タコ
4	D	3	*null*

FULL OUTER JOIN で 2 つのテーブルを結合できた

相手のテーブルに一致する値がないレコードも結果テーブルに含まれる

上記の結果テーブルは、販売テーブル（small_sales_1）を左側、商品マスタテーブル（small_master）を右側として、FULL OUTER JOINで結合しています。前掲の［5-2-7］とは、結合後の結果が異なることが分かるでしょうか。

結果テーブルの［product_id］が「B」「C」のレコードは、左側と右側の両方に値があるため、すべてのフィールドに値が格納されます。「A」は右側に値がありますが、左側には値がないため、左側のみのフィールドである［qty］は「null」となります。「D」は逆に、左側にあって右側にはないため［product_name］が「null」となる、といった具合です。

このFULL OUTER JOINを実行する意図は、『「null」が含まれても構わないから、とにかく2つのテーブルに、つまり「販売されたりマスター登録されていたりする商品」のレコードをすべて結果テーブルに含めたい』と表現できるでしょう。

SECTION

5-3 複数条件や3テーブル以上の結合

2つのテーブルを横につなぐ、JOINの基本構文を見てきました。本節ではJOINの応用的な構文として、複数条件での結合や、3つ以上のテーブルの結合について学びます。

新しい命令が登場するのでしょうか？

命令自体はこれまでに学んだものが使えますが、SQL文の記述方法が複雑になっていきます。引き続き、人間の目で見て理解できる小さなテーブルを例に解説します。

複数のフィールドを使って結合する

多くの企業では、自社が扱う商品・サービスについて販売金額の目標が設定されており、それに対する実績を対比しながら、達成状況を把握しているはずです。そして、販売実績を記録したデータと、販売目標を記録したデータは、別になっていることが多いのではないでしょうか。

これらのデータをBigQuery上のテーブルとして管理し、実績と目標を対比したい場合、SQLを利用して2つのテーブルをJOINする必要性が出てきます。簡単な例として、次ページの［5-3-1］を挙げました。

上にある2つのテーブルに、販売実績と販売目標のデータが格納されています。これらを結合し、下にあるような販売実績・目標テーブルを作成したいとしましょう。[year_month]（年月）と［product_category]（商品カテゴリ）ごとに、[sales]（実績）と［target]（目標）が対比できるようになっています。

5-3-1 販売実績と販売目標のテーブル

販売実績テーブル

year_month	product_category	sales
2019-01-01	Men's	134
2019-01-01	Lady's	122
2019-02-01	Men's	155
2019-02-01	Lady's	116
2019-03-01	Men's	152
2019-03-01	Lady's	139

販売目標テーブル

month	category	target
2019-01-01	Men's	130
2019-01-01	Lady's	120
2019-02-01	Men's	160
2019-02-01	Lady's	120
2019-03-01	Men's	160
2019-03-01	Lady's	130

販売実績・目標テーブル

year_month	product_category	sales	target
2019-01-01	Men's	134	130
2019-01-01	Lady's	122	120
2019-02-01	Men's	155	160
2019-02-01	Lady's	116	120
2019-03-01	Men's	152	160
2019-03-01	Lady's	139	130

2つのテーブルを結合し、実績と目標を対比したい

販売実績は［small_sales_2］、販売目標は［small_target］というテーブル名にし、結合するためのSQL文を次ページの［5-3-2］のように記述しました。2つのテーブルには「null」がないため、内部結合と外部結合のどちらでも同じ結果になりますが、ここでは内部結合を利用しました。INNER JOINの「INNER」は省略して記述しています。

また、［small_sales_2］テーブルには［jisseki］、［small_target］テーブルには［mokuhyo］と別名を付けました。年月を表すフィールドが実績では［year_month］、目標では［month］と名前が異なっているため、USING句ではなくON句を使っています。

実行結果はSQL文の下の通りですが、期待していたテーブルとは異なっています。しかも、レコード数が6行から12行に増えてしまいました。何がいけなかったのでしょうか？

5-3-2 単一の条件で結合する

```
1  SELECT * FROM sample.small_sales_2 AS jisseki
2  JOIN sample.small_target AS mokuhyo
3  ON jisseki.year_month = mokuhyo.month
```

結果テーブル

行	year_month	product_category	sales	month	category	target
1	2019-01-01	Men's	134	2019-01-01	Men's	130
2	2019-01-01	Men's	134	2019-01-01	Lady's	120
3	2019-02-01	Men's	155	2019-02-01	Men's	160
4	2019-02-01	Men's	155	2019-02-01	Lady's	120
5	2019-03-01	Men's	152	2019-03-01	Men's	160
6	2019-03-01	Men's	152	2019-03-01	Lady's	130
7	2019-01-01	Lady's	122	2019-01-01	Men's	130
8	2019-01-01	Lady's	122	2019-01-01	Lady's	120
9	2019-02-01	Lady's	116	2019-02-01	Men's	160
10	2019-02-01	Lady's	116	2019-02-01	Lady's	120
11	2019-03-01	Lady's	139	2019-03-01	Men's	160
12	2019-03-01	Lady's	139	2019-03-01	Lady's	130

テーブルを結合できたが、意図とは異なる結果になっている

［5-3-2］はSQL文として成立していますが、今回の意図においては正しくありません。なぜなら、ON句において「[year_month］と［month］は同一である」という1つの条件しか指定していないからです。

実績テーブルと目標テーブルでは、年月を表す［year_month］と［month］に加えて、商品カテゴリを表す［product_category］と［category］も共通したフィールドになっています。年月だけを条件に結合すると、商品カテゴリで重なっている値（ここでは「Men's」「Lady's」）はそのまま残り、実績テーブルの1レコードに対して、目標テーブルの2レコードが結合します。結果、レコード数が倍増してしまったわけです。

これを回避するには、**結合する条件を複数指定**します。ON句において［year_month］と［month］は同一である」に加えて、「[product_category］と［category］は同一である」という条件も、AND演算子を使って指定しましょう。SQL文を修正すると、次ページの［5-3-3］になります。

5-3-3 複数の条件で結合する

```
1  SELECT * FROM sample.small_sales_2 AS jisseki
2  JOIN sample.small_target AS mokuhyo
3  ON jisseki.year_month = mokuhyo.month AND
4  jisseki.product_category = mokuhyo.category
```

結果テーブル

行	year_month	product_category	sales	month	category	target
1	2019-01-01	Men's	134	2019-01-01	Men's	130
2	2019-02-01	Men's	155	2019-02-01	Men's	160
3	2019-03-01	Men's	152	2019-03-01	Men's	160
4	2019-01-01	Lady's	122	2019-01-01	Lady's	120
5	2019-02-01	Lady's	116	2019-02-01	Lady's	120
6	2019-03-01	Lady's	139	2019-03-01	Lady's	130

2つの条件でテーブルを結合できた

実績テーブルと目標テーブルを対比できる形で、適切に結合できました。しかし、同じ内容のフィールドである［year_month］と［month］、［product_category］と［category］が結果テーブルに両方とも残っており、冗長な印象です。さらに整えていきましょう。

同時に、目標に対する実績の達成率をSQLによる演算で求め、フィールドの並べ替えも行うことにします。以下のSQL文［5-3-4］と結果テーブルを見てください。

5-3-4 演算・並べ替えと同時に複数の条件で結合する

```
1  SELECT jisseki.year_month
2  , jisseki.product_category
3  , jisseki.sales
4  , mokuhyo.target
5  , jisseki.sales / mokuhyo.target AS achivement_rate
```

```
6   FROM sample.small_sales_2 AS jisseki
7   JOIN sample.small_target AS mokuhyo
8   ON jisseki.year_month = mokuhyo.month AND
9   jisseki.product_category = mokuhyo.category
10  ORDER BY year_month, product_category
```

結果テーブル

行	year_month	product_category	sales	target	achivement_rate
1	2019-01-01	Lady's	122	120	1.0166666666666666
2	2019-01-01	Men's	134	130	1.0307692307692307
3	2019-02-01	Lady's	116	120	0.9666666666666667
4	2019-02-01	Men's	155	160	0.96875
5	2019-03-01	Lady's	139	130	1.0692307692307692
6	2019-03-01	Men's	152	160	0.95

2つの条件でテーブルを結合し、達成率を取得できた

[year_month]と[month]、[product_category]と[category]は、1～2行目においてそれぞれの前者だけを残すように指定しました。また、3～4行目で指定した[sales]と[target]を分子・分母とし、5行目で達成率の計算と[achivement_rate]という別名の付与を行っています。

さらに、10行目のORDER BY句で[year_month][product_category]の昇順で並べ替えを実行しました。結果、[5-3-1]で示した販売実績・目標テーブルに近いデータを取り出せたと思います。

「JOINしたら想定よりもレコード数が増えてしまった」というミスは、SQLの初心者にありがちです。[5-3-2]の例では商品カテゴリの値が2つであったため、6から12に倍増しただけで済みました。しかし、仮に50種類のカテゴリが存在したとすると、1に対して50のレコードが結合され、6レコードの50倍となる300レコードの結果が返ってきます。

これを逆手に取り、「JOINが間違っているとレコード数が増えている可能性がある」と考えることによって、緩やかにではありますが、テーブルの結合が意図通りに行われているかを検算できます。実務上は特定のフィールドにいくつのユニークな値があるか、分からないことも多いですが、**JOINの実行後は結果テーブルのレコード数を確認する習慣をつける**ことが望ましいといえます。

3つ以上のテーブルを結合する

続いて、JOINの応用的な構文として、3つ以上のテーブルの結合について見ていきます。これまで2つのテーブルの結合を学んできましたが、構文として大きな違いはありません。

以下の［5-3-5］と次ページの［5-3-6］は、3つのテーブルを結合する構文になります。いずれもINNER JOIN（「INNER」は省略）の構文になっていますが、LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOINに置き換えて使うこともできます。

［5-3-5］はON句を使っており、結合に使うフィールド名がテーブル間で異なる場合に利用します。結合に使うフィールド名が同一の場合でも問題ありません。［5-3-6］はUSING句を使った構文で、結合に使うフィールド名が同一の場合のみ利用できます。

いずれの構文も、2つのテーブルの結合と比較して、JOIN句とON句、JOIN句とUSING句のセットが1つ増えていることが分かると思います。

5-3-5 3つのテーブルを結合する構文（ON句を利用）

```
SELECT フィールド名
FROM テーブル名①
JOIN テーブル名②
ON テーブル名①.テーブル名①の結合に使うフィールド名 =
テーブル名②.テーブル名②の結合に使うフィールド名
JOIN テーブル名③
ON テーブル名①.テーブル名の①結合に使うフィールド名 =
テーブル名③.テーブル名の③結合に使うフィールド名
```

5-3-6 3つのテーブルを結合する構文（USING句を利用）

```
1  SELECT フィールド名
2  FROM テーブル名①
3  JOIN テーブル名② USING（①と②の結合に使うフィールド名）
4  JOIN テーブル名③ USING（①と③または
                          ②と③の結合に使うフィールド名）
```

3つのテーブルの結合について、具体例で実際の動作を確認していきましょう。検証用の小さなテーブルとして、以下の［5-3-7］に示した3つのテーブルを想定します。それぞれのテーブルは、次の名前でBigQuery上に存在すると仮定してください。

- ●販売テーブル　⇒［small_sales_3］
- ●ショップマスタ　⇒［small_s_master］
- ●商品マスタ　⇒［small_p_master］

5-3-7 販売・店舗・商品のテーブル

販売テーブル

product_id	shop_id	sales
1	1	134
1	2	122
1	3	155
2	1	116
2	2	152
2	3	139

ショップマスタ

shop_id	area_id	shop_name
1	1	築地
2	1	銀座
3	2	豊洲

商品マスタ

product_id	product_name
1	アジ
2	タコ

販売テーブルには［product_id］［shop_id］［sales］の3つのフィールドがあります。この［shop_id］をショップマスタとの結合に、［product_id］を商品マスタとの結合に使います。それぞれのテーブルで同じフィールド名が使

われているので、シンプルにUSING句を利用して記述したのが以下のSQL文[5-3-8]です。あえてON句を使い、テーブル名とフィールド名を指定したSQL文は[5-3-9]となります。

5-3-8 3つのテーブルを結合する①（USING句を利用）

```
SELECT *
FROM sample.small_sales_3
JOIN sample.small_s_master USING (shop_id)
JOIN sample.small_p_master USING (product_id)
```

結果テーブル

行	shop_id	product_id	sales	product_name	shop_area_id	shop_name
1	1	1	134	アジ	1	築地
2	2	1	122	アジ	1	銀座
3	3	1	155	アジ	2	豊洲
4	1	2	116	タコ	1	築地
5	2	2	152	タコ	1	銀座
6	3	2	139	タコ	2	豊洲

3つのテーブルを結合できた

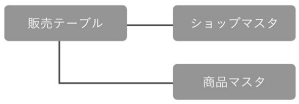
5-3-9 3つのテーブルを結合する①（ON句を利用）

```
SELECT p_master.product_name
, s_master.shop_name
, orders.sales
FROM sample.small_sales_3 AS orders
JOIN sample.small_s_master AS s_master
ON orders.shop_id = s_master.shop_id
JOIN sample.small_p_master AS p_master
ON orders.product_id = p_master.product_id
```

意外とあっさり、3つのテーブルのJOINが完成しました。

前ページの［5-3-8］と［5-3-9］では、販売テーブルにショップマスタ、商品マスタが紐づく形となっています。いわば販売テーブルを「幹」、ショップマスタと商品マスタを「枝」とした「幹と枝」の関係で結合しており、以下の図［5-3-10］のように表せます。

5-3-10 3つのテーブルの結合（幹と枝）



3つのテーブルの結合について、もう1つ例を見てみましょう。以下の［5-3-11］に示した3つのテーブルのうち、販売テーブルとショップマスタは、先ほどの例の［5-3-7］で示したものと同じになっています。

ただし、3つ目のテーブルは、新しく地域マスタを想定します。それぞれのテーブルは、次の名前でBigQuery上に存在すると仮定してください。

- 販売テーブル ⇒［small_sales_3］
- ショップマスタ ⇒［small_s_master］
- 地域マスタ ⇒［small_a_master］

5-3-11 販売・店舗・地域のテーブル

販売テーブル

product_id	shop_id	sales
1	1	134
1	2	122
1	3	155
2	1	116
2	2	152
2	3	139

ショップマスタ

shop_id	shop_id	shop_name
1	1	築地
2	1	銀座
3	2	豊洲

地域マスタ

shop_area_id	area_name
1	中央区
2	江東区

販売テーブルにある［shop_id］フィールドをショップマスタとの結合に使うのは先ほどと同じですが、販売テーブルと地域マスタには、共通して持つフィールドがありません。よって、ショップマスタのフィールドである［shop_area_id］を、地域マスタとの結合に使います。

さらに、取得するフィールドを［product_id］［area_name］［shop_name］［sales］の4つに指定すると、3つのテーブルを結合するSQL文は以下の［5-3-12］となります。

5-3-12 3つのテーブルを結合する②

```
SELECT orders.product_id
, areas.area_name
, shops.shop_name
, orders.sales
FROM sample.small_sales_3 AS orders
JOIN sample.small_s_master AS shops
ON orders.shop_id = shops.shop_id
JOIN sample.small_a_master AS areas
ON shops.shop_area_id = areas.shop_area_id
```

結果テーブル

行	product_id	area_name	shop_name	sales
1	1	中央区	築地	134
2	1	中央区	銀座	122
3	1	江東区	豊洲	155
4	2	中央区	築地	116
5	2	中央区	銀座	152
6	2	江東区	豊洲	139

取得するフィールドを指定して3つのテーブルを結合できた

前ページのSQL文と結果テーブルでは、販売テーブルにショップマスタ、ショップマスタに地域マスタが紐づくという、以下の図［5-3-13］で示したような、3つのテーブルが「数珠つなぎ」の格好になっています。

3つのテーブルを結合する場合、結合に使うフィールドをどう指定するかによって、「幹と枝」または「数珠つなぎ」の関係になることをイメージしてください。

5-3-13 3つのテーブルの結合（数珠つなぎ）

本節の最後に、4つのテーブルを結合する例も紹介しましょう。前掲の［5-3-7］と［5-3-11］で示した4つのテーブルが、次の名前でBigQuery上に存在すると仮定します。

- ●販売テーブル ⇒［small_sales_3］
- ●ショップマスタ ⇒［small_s_master］
- ●商品マスタ ⇒［small_p_master］
- ●地域マスタ ⇒［small_a_master］

上記4つのテーブルを結合し、取得するフィールドを［product_name］［area_name］［shop_name］［sales］の4つに指定したSQL文は、次ページの［5-3-14］となります。

このとき、4つのテーブルは図［5-3-15］のような形で結合されています。販売テーブル、ショップマスタ、地域マスタは「数珠つなぎ」、販売テーブルと商品マスタは「幹と枝」の関係になっていることを、SQL文と結果テーブルから読み取ってください。

5-3-14 4つのテーブルを結合する

```
SELECT p_master.product_name
, s_master.shop_name
, a_master.area_name
, orders.sales
FROM sample.small_sales_3 AS orders
JOIN sample.small_s_master AS s_master
ON orders.shop_id = s_master.shop_id
JOIN sample.small_a_master AS a_master
ON s_master.shop_area_id = a_master.shop_area_id
JOIN sample.small_p_master AS p_master
ON orders.product_id = p_master.product_id
```

結果テーブル

行	product_name	area_name	shop_name	sales
1	アジ	中央区	築地	134
2	アジ	中央区	銀座	122
3	アジ	江東区	豊洲	155
4	タコ	中央区	築地	116
5	タコ	中央区	銀座	152
6	タコ	江東区	豊洲	139

4つのテーブルを結合できた

5-3-15 4つのテーブルの結合

SECTION

5-4 同一テーブルの結合とnull値の置換

JOINについての発展的な知識を学びましょう。本節では「SELF JOIN」や「IFNULL関数」が登場します。

テーブルの結合について、だんだん分かってきました。用途が分かると楽しくなってきますね。

はい。どのような分析がしたいのかを、よくイメージしながらSQL文を記述するのが大切です。

同じテーブルをずらして結合する

これまでは、販売テーブルと商品マスタテーブルなど、異なるテーブルを結合してきました。このような結合が一般的ではありますが、SQLでは同じテーブルを結合することもできます。自分自身（SELF）を結合する（JOIN）ので、「SELF JOIN（セルフ ジョイン）」（自己結合）とも呼ばれます。

例として、ある商品の年間の販売個数について、前年比成長率を取得したいとしましょう。自己結合では同一のテーブルを結合するので、利用するテーブルは以下の［5-4-1］に示した1つのみです。

5-4-1 年間の販売個数のテーブル

販売テーブル

year	qty
2017	272
2018	309
2019	310

［5-4-1］のテーブルから、どのようにして前年比成長率を取得するのでしょうか？ SQLでは同一のレコード内にある異なるフィールド同士での演算が可能なので、2017年の［qty］と2018年の［qty］が同一のレコードにあれば、「2018年の販売個数÷2017年の販売個数」の計算式で［qty］の成長率を取り出せそうです。

そこで、以下の［5-4-2］のように［year］フィールドを1レコード分ずらして、販売テーブル同士を結合します。すると、❶のレコードには2017年と2018年、❷のレコードには2018年と2019年の販売個数が同一のレコードに格納されることになります。

5-4-2 1レコードずらした同一テーブルの結合

販売テーブル

year	qty
2017	272
2018	309
2019	310

year	qty	year	qty
2017	272	2018	309
2018	309	2019	310
2019	310		

❶
❷

この販売テーブルの名前を［small_year］とすると、［5-4-2］のように1レコードずらした自己結合は、次ページのSQL文［5-4-3］で実現できます。6〜7行目に記述したように、左側テーブルには［self1］、右側テーブルには［self2］と別名を付けました。

自己結合のためのSELF JOINという句があるわけはなく、7行目の通りにINNER JOINを利用します。成長率の計算式は5行目にあり、［growth_rate］フィールドに格納しています。

ポイントは8行目にあるON句です。左側テーブル［self1］の［year］を、右側テーブル［self2］の［year］から「1」を引いた値と結合するように記述しています。その結果、左側の［year］が2017年のレコードを、右側の2018年のレコードと結合できるわけです。

5-4-3 1レコードずらして自己結合する

```
1  SELECT self1.year AS base_year
2  , self2.year AS next_year
3  , self1.qty AS base_qty
4  , self2.qty AS next_qty
5  , self2.qty / self1.qty AS growth_rate
6  FROM sample.small_year AS self1
7  INNER JOIN sample.small_year AS self2
8  ON self1.year = self2.year - 1
```

結果テーブル

行	base_year	next_year	base_qty	next_qty	growth_rate
1	2017	2018	272	309	1.1360294117647058
2	2018	2019	309	310	1.0032362459546926

1レコードずらして販売テーブル同士を結合できた

同じテーブルを総当たりで結合する

続いて覚えてほしいのが「**CROSS JOIN**（クロス ジョイン）」（交差結合）です。これは左側テーブルのすべてのレコードの1行1行に、右側テーブルの全レコードを結合する「総当たり」での結合方法です。利用シーンはあまり多くないのですが、知っておくと意外なときに役立ちます。

前掲の［5-4-1］と同じ販売テーブルを例に説明しましょう。先ほどは前年比成長率を取得しましたが、今度は2017年の［qty］を「100」としたときの各年の成長率をSQLの演算によって求めます。同一のレコードにあるフィールド同士の値は計算できるので、次ページの［5-4-4］のようなテーブルが必要、というところまでは想像できると思います。

そこで、次ページのSQL文［5-4-5］を実行します。5〜6行目に注目すると、FROM句とCROSS JOIN句で同じ［small_year］テーブルを指定しています。つまり、自己結合しつつ交差結合する命令になっています。

CROSS JOINはINNER JOINやLEFT OUTER JOINと異なり、総当たりで結合するため、ON句やUSING句が果たしている結合の条件を示す必要がなく、非常にシンプルに記述できます。結果テーブルは、もともと3レコードだったテーブルが3×3＝9レコードになりました。

5-4-4 同一テーブルの総当たりでの結合

base_year	qty	year	qty
2017	272	2017	272
2017	272	2018	309
2017	272	2019	310

5-4-5 同一テーブルを交差結合する

```
1  SELECT self1.year AS base_year
2  , self2.year AS compare_year
3  , self1.qty AS base_qty
4  , self2.qty AS compare_qty
5  FROM sample.small_year AS self1
6  CROSS JOIN sample.small_year AS self2
```

結果テーブル

行	base_year	compare_year	base_qty	compare_qty
1	2019	2018	310	309
2	2019	2019	310	310
3	2019	2017	310	272
4	2017	2018	272	309
5	2017	2019	272	310
6	2017	2017	272	272
7	2018	2018	309	309
8	2018	2019	309	310
9	2018	2017	309	272

販売テーブル同士を総当たりで結合できた

［5-4-5］の結果テーブルの［base_year］フィールドが「2017」となっているレコードに注目してください。［compare_year］に「2018」「2017」「2019」の値があり、求めている組み合わせがそろっています。

そこで、［5-4-5］のSQL文を修正し、以下の［5-4-6］を記述します。8行目のWHERE句で、別名を［base_year］としている［self1.year］フィールドを「2017」で絞り込むように指定しました。

そして、2017年の販売数量を「100」とした、2018年、2019年の成長率を取得するため、5行目に計算式を記述して［growth_rate］として格納しました。これで最終的に取得したいテーブルとなりました。

5-4-6 交差結合した結果を絞り込む

```
1  SELECT self1.year AS base_year
2  , self2.year AS compare_year
3  , self1.qty AS base_qty
4  , self2.qty AS compare_qty
5  , self2.qty / self1.qty * 100 AS growth_rate
6  FROM sample.small_year AS self1
7  CROSS JOIN sample.small_year AS self2
8  WHERE self1.year = 2017
9  ORDER BY compare_year
```

結果テーブル

行	base_year	compare_year	base_qty	compare_qty	growth_rate
1	2017	2017	272	272	100.0
2	2017	2018	272	309	113.60294117647058
3	2017	2019	272	310	113.97058823529412

［base_year］フィールドの値が「2017」のレコードのみに絞り込んだ

［growth_rate］として2017年を「100」とした成長率を取得できた

テーブル内の「null」を別の値に置換する

テーブル結合の4つのタイプを学んだSECTION 5-2では、テーブル内に「null」があるかどうかで結合の挙動が変わることを説明しました。**テーブルの結合を行うときは「null」の扱いに慎重になるべき**であり、「null」をどのように扱うかを考えて結合のタイプを選択する必要があります。

そこでマスターしたいのが、**IFNULL関数**と**COALESCE関数**です。以下の［5-4-7］に示したLTVテーブルと顧客テーブルを例に解説します。

5-4-7 LTV・顧客テーブル

LTVテーブル

user_id	ltv
1	3400
2	8200
3	1500
4	1600
5	5100
6	3900

顧客テーブル

user_id	registration_year	first_purchase_year
1		2017
2	2017	2017
3		2018
4	2017	2018
5	2018	2019
6	2019	2019

LTVテーブルは、［user_id］ごとのライフタイムバリュー（これまでに顧客が購入した金額の総額）を記録したテーブルです。顧客テーブルには、それぞれの［user_id］が会員登録した年である［registration_year］と、初回購入を行った年である［first_purchase_year］が記録されています。

分析の意図は、会員登録年ごとのLTVの合計値を取得することで、例えば「2017年に会員登録した顧客は合計でいくら購入してくれたのか」を知ることです。LTVはLTVテーブルに、会員登録年は顧客テーブルにあるので、2つのテーブルを結合させる必要があることは容易に想像できます。

しかし、顧客テーブルの［registration_year］は、6レコードのうち2レコードが「null」となっており、会員登録年に「null」が出現することは不可避なように思えます。そこで『当該サービスを開始したのは2017年で、会員登録年が「null」の場合は2017年として処理する』こととしましょう。こうした場合に利用できるのがIFNULL関数です。

IFNULL関数を使えば、**値が「null」だったときに適用する値を任意に指定**できます。会員登録年の「null」をすべて「2017」に置換すれば、「null」の問題を解決することができるわけです。

LTVテーブルを［small_ltv］、顧客テーブルを［small_customers］とし、IFNULL関数を記述したうえでJOINするSQL文は、以下の[5-4-8]となります。IFNULL関数は2行目にあり、［registration_year］フィールドを対象に、値が「null」であれば「2017」に置換する働きをします。

また、IFNULL関数を実行した［registration_year］に［reg_year］と別名を付けてグループ化し、［ltv］の合計値を集計関数で求めている点にも注目してください。

結果テーブルでは、会員登録年が「null」だった［user_id］が「1」「3」の顧客も2017年の登録として計算でき、［user_id］が「1」～「4」の顧客のLTVの合計値である14,700円が2017年に集計されています。

5-4-8 「null」を任意の値で置換する

```
1  SELECT
2  IFNULL(registration_year, 2017) AS reg_year
3  , SUM(ltv) AS sum_ltv
4  FROM sample.small_ltv
5  INNER JOIN sample.small_customers USING (user_id)
6  GROUP BY reg_year
```

結果テーブル

行	reg_year	sum_ltv
1	2017	14700
2	2018	5100
3	2019	3900

「null」を「2017」に置換したうえでテーブルを結合できた

同じ例で、COALESCE関数についても解説しましょう。COALESCE関数では、IFNULL関数よりも高度な「null」の置換が可能になります。

先ほどのINFULL関数では、「null」を固定値の「2017」で置換しました。しかし、初回購入と同時に会員登録する顧客が多いと仮定すると、『会員登録年が「null」だった場合は初回購入年で代用する』ことが現実に即しているかもしれません。

このような場合、COALESCE関数を利用して『[registration_year]が「null」であれば、初回購入年を表す［first_purchase_year］の値で置換する』という条件を指定します。以下のSQL文［5-4-9］では、2行目のCOALESCE関数により［registration_year］フィールドに「null」があった場合、[first_purchase_year］の値に置換されます。

［user_id］が「3」の顧客は会員登録年が「null」だったため、初回購入年である「2018」が代用されます。[user_id］が「5」の顧客のLTVと合算され、2018年のLTVの集計値が6,600円として取得されました。

5-4-9 「null」を別フィールドの値で置換する

```
1  SELECT
2  COALESCE(registration_year, first_purchase_year)
3  AS reg_year
4  , SUM(ltv) AS sum_ltv
5  FROM sample.small_ltv
6  INNER JOIN sample.small_customers USING (user_id)
7  GROUP BY reg_year
```

結果テーブル

行	reg_year	sum_ltv
1	2017	13200
2	2018	6600
3	2019	3900

「null」を［first_purchase_year］の値に置換したうえでテーブルを結合できた

STEP UP

非等値結合を利用したバスケット分析

ここまでで学んだJOINは、いずれもON句やUSING句で、左右のテーブルの指定したフィールドの値が「等しい」レコード同士を結合していました。しかし、いつでも必ず「等しい値」で結合するかというと、そうでもありません。場合によっては「異なる値」で結合することもあり、そのような結合を「非等値結合」と呼びます。

非等値結合によって可能になる代表的な分析手法に「マーケットバスケット分析」があります。マーケットバスケット分析とは、同一の注文で併せ買いされる商品の組み合わせを可視化する手法です。

以下の［5-4-10］にある注文テーブルを見てください。[order_id]フィールドが「1」のレコードに注目すると、「アジとサバ」「アジとタコ」「サバとタコ」が併せ買いされていることが分かります。また、[order_id]フィールドが「3」のレコードでも、「アジとタコ」が併せ買いされています。このお店では「アジとタコ」の組み合わせが人気なのかもしれません。

そこで、1回の注文における商品の組み合わせを、SQLによって可視化してみることにします。注文テーブルの名前を［small_order］とし、自己結合しつつ、同じ値ではなく異なる値で非等値結合しているのが次ページのSQL文［5-4-11］です。

5-4-10 注文テーブル

order_id	product_name
1	アジ
1	サバ
1	タコ
2	キス
2	タコ
3	アジ
3	タコ

5-4-11 非等値で自己結合する

```
1  SELECT self1.order_id
2  , self1.product_name
3  , self2.product_name AS product_name2
4  FROM sample.small_order AS self1
5  INNER JOIN sample.small_order AS self2
6  ON
7  self1.order_id = self2.order_id AND
8  self1.product_name < self2.product_name
```

結果テーブル

行	order_id	product_name	product_name2
1	1	アジ	サバ
2	1	アジ	タコ
3	1	サバ	タコ
4	2	キス	タコ
5	3	アジ	タコ

非等値結合により、1回の注文における商品の組み合わせを取得できた

自己結合のフィールドを指定している6～8行目のON句に注目してください。[order_id] は等しい値で結合していますが、[product_name] については「左側テーブルの [product_name] の値が、右側テーブルの [product_name] の値より大きい」という非等値結合の条件を指定しています。これにより「アジ - アジ」といった等値ではなく、「アジ - サバ」「アジ - タコ」といった異なる商品の組み合わせが取得できます。

なお、8行目の「<」を「<>」に変更すると、「アジ - サバ」に加えて「サバ - アジ」の組み合わせも出力され、結果テーブルは倍の10レコードになります。今回は順序を問うていないため「<」が適切です。

SECTION

5-5 テーブルの集合演算

データベースで扱うテーブルは一般的に複数存在し、それらを「横につなぐ」JOINについて、ひと通り学びました。本節では、テーブルを「縦につなぐ」UNIONなどについて見ていきます。

「結合」ではなく「集合演算」と呼ぶのですね。

はい。UNIONはテーブルという「集合」同士の足し算＝「演算」をする働きをします。ほかにも、掛け算として複数テーブルの重複するデータを取得したり、引き算として差分だけを取得したりする方法があります。

「和集合」「積集合」「差集合」のイメージを理解する

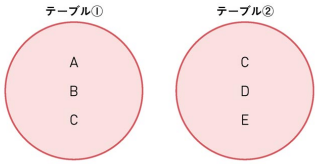
SECTION 5-1では、複数のテーブル同士をつなぐ方法には2種類があることを説明し、日別や月別など、同じスキーマで構成されたテーブルをつなぐことを「縦につなぐ」と表現しました。

SQLにおいて、テーブル同士を足し合わせて縦につなぐには「UNION」を利用します。また、重複しているレコードだけを取得する「INTERSECT（インターセクト）」、差分のレコードだけを取得する「EXCEPT（エクセプト）」についても、本節でまとめて学びます。これらは集合演算子とも呼ばれます。

具体的なSQLを学ぶ前に、テーブルの集合演算を概念的に理解してください。テーブルを数学の「集合」に例えて、それらの足し算、掛け算、引き算、つまり「演算」をすると考えると分かりやすいでしょう。

以下の図［5-5-1］は、2つのテーブルを集合としてイメージしたものです。テーブル①には「A」「B」「C」、テーブル②には「C」「D」「E」と、3つの値が格納されています。これらを例に、UNION、INTERSECT、EXCEPTからどのような結果を取得できるかを見ていきます。

5-5-1 集合演算に利用するテーブルのイメージ

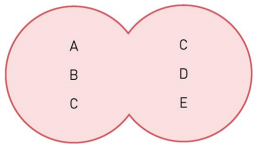


▶UNION

テーブル同士を縦につなぐUNIONは、テーブル①（集合①）とテーブル②（集合②）を単純に足し合わせます。その結果は以下の図［5-5-2］のようになり、結果テーブルには「A」「B」「C」「C」「D」「E」という値が残ります。こうした集合は「和集合」と呼びます。

ただし、これはUNIONを「値の重複を許可する」というルールで実行した場合で、別途「値の重複を許可しない」こともできます、重複を許可しない場合、結果に残る値は「A」「B」「C」「D」「E」になります。

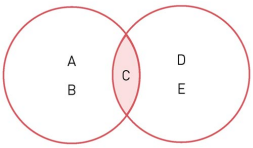
5-5-2 UNION（和集合）のイメージ



▶INTERSECT

テーブル①とテーブル②のうち、共通（重複）している部分だけを取り出すのがINTERSECTです。結果は以下の図［5-5-3］のイメージとなり、値には両方のテーブルに含まれている「C」だけが残ります。こうした集合は「積集合」と呼びます。

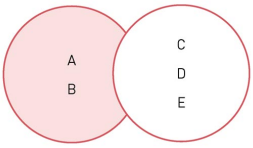
5-5-3 INTERSECT（積集合）のイメージ



▶EXCEPT

テーブル①からテーブル②を差し引き、その結果を取得するのがEXCEPTです。以下の図［5-5-4］のように、結果に残る値は「A」「B」となります。「C」はテーブル②にも含まれているので、結果には残りません。このようにしてできた集合は「差集合」と呼びます。

5-5-4 EXCEPT（差集合）のイメージ



この3つの集合演算のうち、EXCEPT（差集合）のみ、計算式の記述順序によって結果が異なるので注意してください。

前掲の［5-5-4］では「テーブル①ーテーブル②」という演算をしていますが、これを「テーブル②ーテーブル①」と逆にすると、結果に残る値は「D」「E」となります。一方、UNION（和集合）とINTERSECT（積集合）は、順序が入れ替わっても結果は変わりません。

UNION、INTERSECT、EXCEPTの、データ分析における利用シーンを想定すると、以下の表［5-5-5］のようになります。これまでの例に照らすと、テーブル①＝1月の販売テーブル、テーブル②＝2月の販売テーブルという形です。1月、2月の販売テーブルをその下の［5-5-6］のような内容と仮定し、集合演算を行う具体的なSQL文を見ていきましょう。

5-5-5 集合演算子と想定する利用シーン

集合演算子	想定する利用シーン
UNION	1月と2月のいずれかに販売された商品を知りたい
INTERSECT	1月と2月の両方で販売された商品を知りたい
EXCEPT	1月には販売されたが、2月には販売されていない商品を知りたい

5-5-6 1月・2月の販売テーブル

1月販売テーブル

date	product_name	qty
2020-01-10	アジ	3
2020-01-11	タコ	1
2020-01-15	サバ	3
2020-01-18	キス	2
2020-01-20	タイ	1

2月販売テーブル

date	product_name	qty
2020-02-09	アジ	3
2020-02-12	ブリ	3
2020-02-15	キス	3
2020-02-19	タイ	2
2020-02-28	イカ	1

2つのテーブルから和集合を作成する

まず、和集合を作るUNIONの構文は次ページの［5-5-7］となります。留意点としては、最初と2番目のSELECT句で、**指定するフィールドの種類と数、順序を完全に一致させる必要がある**ことが挙げられます。

5-5-7 UNIONの構文

```
SELECT フィールド名 FROM テーブル名①
UNION DISTINCT（または UNION ALL）
SELECT フィールド名 FROM テーブル名②
```

例えば、以下のSQL文［5-5-8］は成立しますが、その下の［5-5-9］は成立しません。後者のSQL文は、2つのSELECT句でフィールドの数が一致していないためです。

5-5-8 UNIONの記述例

```
SELECT date, product_name, qty FROM sample.small_jan
UNION DISTINCT
SELECT date, product_name, qty FROM sample.small_feb
```

5-5-9 UNIONの誤用例

```
SELECT date, product_name, qty FROM sample.small_jan
UNION DISTINCT
SELECT product_name, qty FROM sample.small_feb
```

最初のSELECT句にある［date］フィールドが指定されていない

構文［5-5-7］の2行目で指定している「UNION DISTINCT」と「UNION ALL」は、前者が「重複を許さず、結果テーブルから除く」のに対し、後者は「重複を許し、結果テーブルに残す」という点で違いがあります。

例えば、ユーザーや商品のユニーク数（固有値の種類数）を取得したい場合はUNION DISTINCT、のべ数を取得したい場合はUNION ALLを使います。2つのテーブルに重複がない場合は、どちらの結果も同じになります。

ただ、勘違いしやすい点もあるので、実際にSQLを実行した結果で違いを見ていきましょう。例として示した［5-5-6］の2つの販売テーブルのうち、1月を［small_jan］、2月を［small_feb］とします。

以下の［5-5-10］は、UNION DISTINCTの記述例です。UNIONした結果に並べ替えを適用するため、4行目で［date］フィールドの昇順を指定しています。次ページの［5-5-11］では、2行目をUNION ALLに書き換えて実行しました。それぞれの結果テーブルを見比べてみてください。

5-5-10 UNION DISTINCTの記述例①

```
1  SELECT date, product_name, qty FROM sample.small_jan
2  UNION DISTINCT
3  SELECT date, product_name, qty FROM sample.small_feb
4  ORDER BY 1
```

結果テーブル

行	date	product_name	qty
1	2020-01-10	アジ	3
2	2020-01-11	タコ	1
3	2020-01-15	サバ	3
4	2020-01-18	キス	2
5	2020-01-20	タイ	1
6	2020-02-09	アジ	3
7	2020-02-12	ブリ	3
8	2020-02-15	キス	3
9	2020-02-19	タイ	2
10	2020-02-28	イカ	1

重複を許可せずに和集合を作成した

5-5-11 UNION ALLの記述例①

```
1  SELECT date, product_name, qty FROM sample.small_jan
2  UNION ALL
3  SELECT date, product_name, qty FROM sample.small_feb
4  ORDER BY 1
```

結果テーブル

行	date	product_name	qty
1	2020-01-10	アジ	3
2	2020-01-11	タコ	1
3	2020-01-15	サバ	3
4	2020-01-18	キス	2
5	2020-01-20	タイ	1
6	2020-02-09	アジ	3
7	2020-02-12	ブリ	3
8	2020-02-15	キス	3
9	2020-02-19	タイ	2
10	2020-02-28	イカ	1

重複を許可して和集合を作成した

意外性を感じてもらうため、あえて両方掲載しましたが、結果はまったく同じになっています。1月・2月テーブルでは、例えば［product_name］フィールドの「アジ」「タイ」が重複しているので、UNION DISTINCTの結果はレコード数が減るはず、と考えたかもしれません。

これは集合演算における重要なポイントですが、**「重複」とは「SELECT句で取得している全フィールドの内容がすべて一致する」**という意味になります。［5-5-10］と［5-5-11］のSQL文でいえば、[date] [product_name] [qty]の3つのフィールドをSELECT句で取得しているので、その3つの組み合わせがすべて同じ場合に重複とみなされます。

再び結果テーブルを見てください。[product_name］が「アジ」のレコードは1行目と6行目にあり、[qty］も「3」と同じですが、[date］が異なります。そのため重複とはみなされず、UNION DISTINCTを使っていても、1行目と6行目が結果テーブルに残ったというわけです。

同様に見渡しても、1レコードを構成する3フィールドが全部同一というケースは、1月・2月テーブルにはいずれも存在しません。よって、UNION DISTINCTもUNION ALLも同一の結果となりました。

続いて、UNION DISTINCTとUNION ALLで結果が異なる例を見てみましょう。1月・2月テーブルの［date］や［qty］を気にせず、販売された商品である［product_name］のみを取得するとします。

以下の［5-5-12］は、UNION DISTINCTによる和集合を作成するSQL文と結果テーブルです。1月と2月に販売された、ユニークな、つまり重複を取り除いた商品リストが完成しました。

5-5-12 UNION DISTINCTの記述例②

```
1  SELECT product_name FROM sample.small_jan
2  UNION DISTINCT
3  SELECT product_name FROM sample.small_feb
4  ORDER BY 1
```

結果テーブル

行	product_name
1	アジ
2	イカ
3	キス
4	サバ
5	タイ
6	タコ
7	ブリ

［product_name］のみを取得し、重複を許可せずに和集合を作成した

一方、UNION ALLでは次ページの［5-5-13］となります。［product_name］フィールドの重複が許され、1月テーブルの5レコード、2月テーブルの5レコード、あわせて10レコードの商品リストを取得できます。1〜2行目の「アジ」、4〜5行目の「キス」などから重複の許可が確認できます。

和集合の作成においては、このようなUNION DISTINCTとUNION ALLの性質と、重複の条件を踏まえたうえで利用してください。

5-5-13 UNION ALLの記述例②

```
1  SELECT product_name FROM sample.small_jan
2  UNION ALL
3  SELECT product_name FROM sample.small_feb
4  ORDER BY 1
```

結果テーブル

行	product_name
1	アジ
2	アジ
3	イカ
4	キス
5	キス
6	サバ
7	タイ
8	タイ
9	タコ
10	ブリ

［product_name］のみを取得し、重複を許可して和集合を作成した

2つのテーブルから積集合を作成する

テーブル同士で重複するレコードだけを取り出し、積集合を作るのがINTERSECTです。構文は以下の［5-5-14］の通りで、必ずDISTINCTと組み合わせて利用します。INTERSECT ALLという使い方はありません。

5-5-14 INTERSECTの構文

```
1  SELECT フィールド名 FROM テーブル名①
2  INTERSECT DISTINCT
3  SELECT フィールド名 FROM テーブル名②
```

INTERSECTにおける重複も、UNIONで学んだ通り、SELECT句で取り出したすべてのフィールドが一致することを意味します。よって、以下の[5-5-15]のSQL文では結果が表示されません。[date] [product_name] [qty] が完全に同一なレコードは存在しないためです。

一方、その下にある［5-5-16］のように［product_name］フィールドだけをINTERSECTの対象にした場合は、結果テーブルとして1月・2月の両方で販売された［product_name］のリストを取得できます。

5-5-15 INTERSECTの記述例①

```
1  SELECT date, product_name, qty FROM sample.small_jan
2  INTERSECT DISTINCT
3  SELECT date, product_name, qty FROM sample.small_feb
```

5-5-16 INTERSECTの記述例②

```
1  SELECT product_name FROM sample.small_jan
2  INTERSECT DISTINCT
3  SELECT product_name FROM sample.small_feb
```

結果テーブル

行	product_name
1	タイ
2	キス
3	アジ

［product_name］のみを取得して積集合を作成した

2つのテーブルから差集合を作成する

ある集合（テーブル）から別の集合を引いた差集合を作るEXCEPTの構文は、以下の［5-5-17］の通りです。INTERSECTと同様、DISTINCTとのみ組み合わせて利用できます。EXCEPT ALLという使い方はありません。

5-5-17 EXCEPTの構文

```
SELECT フィールド名 FROM テーブル名①
EXCEPT DISTINCT
SELECT フィールド名 FROM テーブル名②
```

次ページのSQL文［5-5-18］では、1月テーブルの［product_name］のリストから、2月テーブルにも存在しているレコードを取り除くよう記述されています。結果的に、1月テーブルにだけに存在する［product_name］である「タコ」「サバ」だけが取得されます。

その下のSQL文［5-5-19］では、テーブル名を記述する順序を逆にしています。［product_name］フィールドについて、2月テーブルから1月テーブルを差し引いた結果となるため、異なる結果となります。

5-5-18 EXCEPTの記述例①

```
1  SELECT product_name FROM sample.small_jan
2  EXCEPT DISTINCT
3  SELECT product_name FROM sample.small_feb
```

結果テーブル

行	product_name
1	タコ
2	サバ

[product_name] のみを取得し、1月から2月を引く差集合を作成した

5-5-19 EXCEPTの記述例②

```
1  SELECT product_name FROM sample.small_feb
2  EXCEPT DISTINCT
3  SELECT product_name FROM sample.small_jan
```

結果テーブル

行	product_name
1	ブリ
2	イカ

[product_name] のみを取得し、2月から1月を引く差集合を作成した

テーブルをつなぐJOINやUNIONの学習は以上で終わりですが、本章はもう少し続きます。今度は、つないだ結果として得られるテーブルを利用する方法について見ていきましょう。

STEP UP

3つ以上のテーブルの集合演算

ここまでで2つのテーブルの集合演算を学びましたが、3つ以上のテーブルの演算も、もちろん可能です。例として、以下の［5-5-20］のような3か月分の販売テーブルがあると仮定しましょう。1月・2月の販売テーブルは［5-5-6］からの再掲です。

5-5-20 1月・2月・3月の販売テーブル

1月販売テーブル

date	product_name	qty
2020-01-10	アジ	3
2020-01-11	タコ	1
2020-01-15	サバ	3
2020-01-18	キス	2
2020-01-20	タイ	1

2月販売テーブル

date	product_name	qty
2020-02-09	アジ	3
2020-02-12	ブリ	3
2020-02-15	キス	3
2020-02-19	タイ	2
2020-02-28	イカ	1

3月販売テーブル

date	product_name	qty
2020-03-01	アジ	1
2020-03-03	エビ	3
2020-03-16	サバ	2
2020-03-19	タイ	2
2020-03-22	タコ	1

1月・2月・3月の販売テーブルのうち、いずれかの月に販売されたユニークな［product_name］のリストは、次ページのSQL文［5-5-21］で取得できます。SELECT句で［product_name］を取得し、1月・2月・3月を対象にUNION DISTINCTを実行して重複のない和集合を作成しています。

5-5-21 3つのテーブルから和集合を作成する

```
SELECT product_name FROM sample.small_jan
UNION DISTINCT
SELECT product_name FROM sample.small_feb
UNION DISTINCT
SELECT product_name FROM sample.small_mar
```

また、3か月のすべての月で販売されたユニークな［product_name］のリストは、以下のSQL文［5-5-22］で取得できます。先ほどのUNIONを、積集合のINTERSECTに書き換えています。

5-5-22 3つのテーブルから積集合を作成する

```
SELECT product_name FROM sample.small_jan
INTERSECT DISTINCT
SELECT product_name FROM sample.small_feb
INTERSECT DISTINCT
SELECT product_name FROM sample.small_mar
```

3つのテーブルの演算における複雑な例として、「1月・2月には販売されておらず、3月に初めて販売された商品のリスト」を取得したいとします。これは3月の集合（テーブル）から、1月・2月テーブルの和集合を引くことで求められ、SQL文としては次ページの［5-5-23］となります。結果は「エビ」が該当し、意図通りのレコードが得られています。

5-5-23 和集合の結果から差集合を作成する

```
1  SELECT product_name FROM sample.small_mar
2  EXCEPT DISTINCT
3  (SELECT product_name FROM sample.small_jan
4  UNION DISTINCT
5  SELECT product_name FROM sample.small_feb)
```

結果テーブル

行	product_name
1	エビ

3月テーブルから1月・2月テーブルの和集合を引いた差集合を作成した

上記のSQL文に注目すると、3行目に記述した［sample_jan］テーブル（1月テーブル）と、5行目に記述した［sample_feb］テーブル（2月テーブル）に対してUNION DISTINCTを使い、和集合を作成しています。その和集合を［sample_mar］テーブル（3月テーブル）からEXCEPT DISTINCTを使って引き、差集合を作成しました。

この演算の順序は、3〜5行目を半角カッコで囲むことで実現しています。算数の計算式と同じく、カッコの中が優先されることをあらためて覚えておきましょう。

SECTION

5-6 仮想テーブルの作成

本節では、初めて登場する概念である「仮想テーブル」を学びます。

仮想テーブルですか……。「仮想」と付くと、何だか難しそうなイメージがあります。

SQL文を実行した結果を「仮に」保存しておいて利用するイメージです。仮想テーブルを作成してしまえば、あたかも「そのようなテーブルがある」という前提でSQLを記述すればよいので、複雑なクエリをスッキリと記述できるようになります。ぜひ覚えてください。

あるSQL文の実行結果を「仮のテーブル」とみなす

これまでに学んだSQL文では、すべてFROM句で「実体のあるテーブル」を指定してきました。本節では、実体を伴ったテーブルではなく、あるSQL文を実行した結果テーブルを「仮想のテーブル」、つまり**仮想テーブル**として、そのテーブルを対象に実行するSQLを学びます。

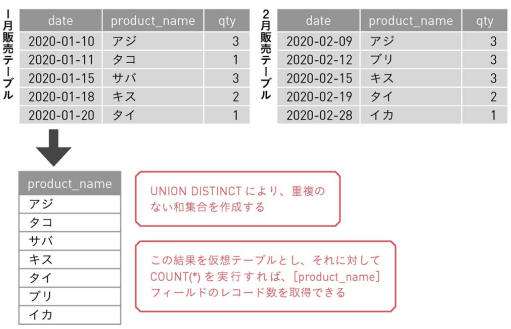
なぜここで仮想テーブルについて学ぶかというと、テーブルの結合や集合演算と組み合わせて利用することが多いからです。次ページの［5-6-1］を見てください。

1月・2月の販売テーブルは、前節までに利用してきた例と同じものです。これらに対してUNION DISTINCTを利用すると、重複のない和集合として1月または2月に販売されたユニークな商品リストを取得できます。

しかし、今はその商品リストが必要なわけではなく、ユニークな商品の数をCOUNT(*)で取得したいとしましょう。その結果を得るには、まずUNION DISTINCTのSQL文を実行し、その結果テーブルに対してCOUNT(*)のSQL文を実行すればよい、ということになります。

このとき、UNION DISTINCTの結果テーブルは「実体のあるテーブル」である必要はなく、「仮想のテーブル」でよい、ということが理解できると思います。これが仮想テーブルの基本的な考え方です。

5-6-1 仮想テーブルの利用イメージ



なお、仮想テーブルを作成する方法には、ここで新しく学ぶ「WITH（ウィズ）」句と、SECTION 2-6でBigQuery上での操作に触れた「ビュー」、そして「サブクエリ」の3つがあります。WITH句とビューは本節で、サブクエリはCHAPTER 6で詳しく学びます。

WITH句で仮想テーブルを作成する

WITH句で仮想テーブルを作成し、その仮想テーブルを利用する構文は、以下の［5-6-2］の通りです。WITH句で仮想テーブルを作成したら、その仮想テーブルがあたかも実在するかのように扱って、通常のSQL文を記述する、と考えてください。

5-6-2 WITH句の構文

```
1  WITH 仮想テーブルの名前 AS (
2  仮想テーブルを作成するSQL文)
3
4  SELECT フィールド名
5  FROM 仮想テーブルの名前
```

この構文を利用して、［5-6-1］で示した「1月または2月に販売されたユニークな商品の数」を取得してみましょう。次ページのSQL文［5-6-3］を見てください。

1～4行目がWITH句で、仮想テーブルの名前は［jan_feb_sales］としました（もちろん任意で付けられます）。ASに続く半角カッコ内で、1月・2月の販売テーブルを対象に、UNION DISTINCTによって重複を許可しない和集合を取得しています。この結果が仮想テーブルとなります。

最後の6行目で、仮想テーブル［jan_feb_sales］を対象にCOUNT(*)を実行し、［unique_prod_number］としてレコード数を取得しました。利用している命令は、これまでにすべて学んだものです。WITH句で作成した仮想テーブルを対象にSQL文を記述するため、メインのクエリは非常にシンプルになりました。

5-6-3 仮想テーブルからレコード数を取得する

```
1  WITH jan_feb_sales AS (
2  SELECT product_name FROM sample.small_jan
3  UNION DISTINCT
4  SELECT product_name FROM sample.small_feb)
5
6  SELECT COUNT(*) AS unique_prod_number FROM jan_feb_sales
```

結果テーブル

行	unique_prod_number
1	7

和集合を作成したうえで
レコード数を取得できた

新しく学んだポイントをまとめると、次のようになります。仮想テーブルを利用することで、SQLで「できること」の幅が大きく広がることが想像できたでしょうか。

- WITH句ではASに続いてSQL文を記述できる
- そのSQL文の実行結果（＝仮想テーブル）に名前を付けられる
- FROM句で仮想テーブルを指定し、別のSQL文を実行できる

仮想テーブルの利用例をもう1つ挙げましょう。[5-6-1]の1月の販売テーブルから、商品の販売個数である［qty］が、テーブル全体の平均値よりも多い商品だけを取得したい、というシナリオを考えてみます。

このシナリオは、次ページの［5-6-4］のような仮想テーブルがあれば解決します。1月の販売テーブルに、販売個数の平均値を格納した［avg_qty］フィールドが追加されています。このテーブルがあれば、WHERE句を利用して「［avg_qty］よりも［qty］が多い」という条件を指定することで、目的のレコードを取り出せるはずです。

5-6-4 1月の平均販売個数を追加した仮想テーブル

date	product_name	qty	avg_qty
2020-01-10	アジ	3	2.0
2020-01-11	タコ	1	2.0
2020-01-15	サバ	3	2.0
2020-01-18	キス	2	2.0
2020-01-20	タイ	1	2.0

[qty] と [avg_qty] の2つのフィールドから目的の条件を指定できる

どのようにSQL文を記述すればよいか、順に考えていきましょう。まず平均販売個数は、GROUP BY句を使わずに、平均値を求めるAVG関数で取り出せます。SQL文は以下の［5-6-5］となります。

5-6-5 平均販売個数を取得する

```
1  SELECT AVG(qty) AS avg_qty FROM sample.small_jan
```

結果テーブル

行	avg_qty
1	2.0

[avg_qty] フィールドとして平均販売個数を取得できた

1月の販売テーブルは5レコードあり、[qty] の合計が10個なので、上記の通り結果は「2.0」です。次に、この平均販売個数のみのテーブルを、1月の販売テーブルと結合します。横につなぐのはJOINですが、4つのタイプのうち、どれになるでしょうか？

2つのテーブルには共通するフィールドがなく、左側（1月の販売テーブル）に右側（平均販売個数のみのテーブル）の全レコードを「総当たり」で結合するのですから、CROSS JOINが正解です。

平均販売個数のみのテーブルを［total_avg_qty］という名前の仮想テーブルとして作成し、1月の販売テーブルに仮想テーブルをCROSS JOINで結合するSQL文は、次ページの［5-6-6］となります。

5-6-6 平均販売個数の仮想テーブルを結合する

```
1  WITH total_avg_qty AS (
2  SELECT AVG(qty) AS avg_qty FROM sample.small_jan)
3
4  SELECT * FROM sample.small_jan
5  CROSS JOIN total_avg_qty
```

結果テーブル

行	date	product_name	qty	avg_qty
1	2020-01-11	タコ	1	2.0
2	2020-01-20	タイ	1	2.0
3	2020-01-18	キス	2	2.0
4	2020-01-10	アジ	3	2.0
5	2020-01-15	サバ	3	2.0

1月の販売テーブルに平均販売個数のテーブルを総当たりで結合できた

あとは［5-6-6］の末尾に「[avg_qty]よりも[qty]が多い」という条件を指定したWHERE句を付け加えれば完成です。以下のSQL文［5-6-7］を実行すると、目的の結果テーブルを取得できます。平均販売個数よりも売れた商品は、「2020-01-10のアジ」と「2020-01-15のサバ」でした。

5-6-7 平均販売個数よりも多く売れた商品を取得する

```
1  WITH total_avg_qty AS (
2  SELECT AVG(qty) AS avg_qty FROM sample.small_jan)
3
4  SELECT * FROM sample.small_jan
5  CROSS JOIN total_avg_qty
6  WHERE qty > avg_qty
```

結果テーブル

行	date	product_name	qty	avg_qty
1	2020-01-10	アジ	3	2.0
2	2020-01-15	サバ	3	2.0

平均販売個数よりも多く売れた商品に絞り込めた

なお、WITH句で作成できる仮想テーブルは、1つとは限りません。以下の構文［5-6-8］に従って、複数の仮想テーブルを作成できます。

その下の［5-6-9］は、2つの仮想テーブルを作成している記述例です。1～4行目のWITH句により、[avg_qty_1］と［avg_qty_2］と名前を付けた2つの仮想テーブルが作成されています。前者は1月の販売テーブルにおける[qty］の平均値、後者は同じく2月の平均値です。

それらを3月の販売テーブル（前節の［5-5-20］を参照）にCROSS JOINで結合し、WHERE句を利用して3月の商品のうち、1月・2月のどちらの平均販売個数よりも多く売れた商品のみに絞り込みました。次ページの通り、意図した結果テーブルが取得できています。

5-6-8 WITH句の構文（複数の仮想テーブルを作成）

```
WITH 仮想テーブル①の名前 AS (
仮想テーブル①を作成するSQL文),
仮想テーブル②の名前 AS (
仮想テーブル②を作成するSQL文)
```

5-6-9 2つの仮想テーブルを作成する

```
WITH avg_qty_1 AS (
SELECT AVG(qty) AS avg_qty_jan
FROM sample.small_jan),
avg_qty_2 AS (
```

```
5   SELECT AVG(qty) AS avg_qty_feb
6   FROM sample.small_feb)
7
8   SELECT * FROM sample.small_mar
9   CROSS JOIN avg_qty_1
10  CROSS JOIN avg_qty_2
11  WHERE qty > avg_qty_jan AND qty > avg_qty_feb
```

結果テーブル

行	date	product_name	qty	avg_qty_jan	avg_qty_feb
1	2020-03-03	エビ	3	2.0	2.4

2つの仮想テーブルを利用して目的のレコードを取得できた

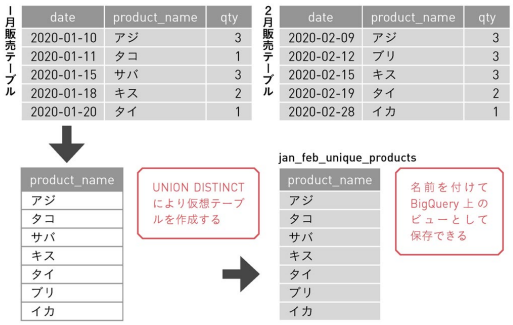
ビューで仮想テーブルを作成する

続いて、ビューを利用した仮想テーブルの作成について解説します。SECTION 1-3のおさらいですが、ビューとはSQL文の実行結果を保存したものです。これは「仮想テーブルを保存したもの」とも表現できます。

WITH句で作成した仮想テーブルは、BigQueryのクエリエディタ内のみに記述されており、保存された状態にありません。そのため、同じ仮想テーブルの使い回しができず、その都度、WITH句を使って記述する必要があります。今、ちょっとだけ分析したいときにはピッタリですが、繰り返し使いたいケースでは最適な方法ではありません。

一方、ビューはBigQuery上に名前を付けて保存されているため、**あたかも実体のあるテーブルのように使い回しができます**。例えば、次ページの［5-6-10］のように、1月・2月の販売テーブルをUNION DISTINCTで結合した仮想テーブルが得られたとします。それをビューとして任意の名前、例えば[jan_feb_unique_products］と名前を付けて保存すれば、ほかのSQL文でも利用することが可能になります。

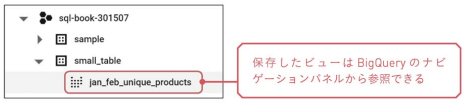
5-6-10 ビューの利用イメージ



ビューの保存方法は、SECTION 2-6を参照してください。そこでも触れた通り、SQL文を実行しなくてもビューは保存できますが、結果テーブルの内容を確認してから保存することをおすすめします。

保存したビューは、[5-6-11]のようにBigQueryのコンソール内にあるナビゲーションパネルに表示されます。ビューを作成したときに指定したデータセットの配下から参照してください。

5-6-11 保存したビュー



［5-6-10］で示したように、1月・2月の販売テーブルをUNION DISTINCTで結合した仮想テーブルをビューとして保存し、そのビューのレコード数を取得するSQL文は［5-6-12］となります。ビューはテーブルのように使い回せるので、ビュー名である［jan_feb_unique_products］をFROM句で指定すれば、あとは通常のSQL文と同じです。

5-6-12 ビューからレコード数を取得する

```
1  SELECT COUNT(*) AS number_of_products
2  FROM small_table.jan_feb_unique_products
```

結果テーブル

行	number_of_products
1	7

ビューを対象にレコード数を取得できた

ビューの参照時には元のSQL文が再実行される

ビューとテーブルは、BigQuery上でのアイコンが異なること以外に、重要な違いがあります。テーブルのように見えても、ビューはあくまでも仮想のテーブルであり、その本質はSQL文の実行によって生成される結果テーブルであるということです。

そのため、ビューを対象としたSQL文は、次ページの図［5-6-13］のような流れで処理されます。まず、FROM句でビューを指定したSQL文を実行すると、ビューを作成した元のSQL文が、実体のあるテーブルを対象として再実行されます。その結果、ビュー（＝仮想テーブル）が更新され、その更新されたビューに対してSQL文を実行した結果が表示されます。図中の例ではCOUNT(*)が実行され、ビューのレコード数が返される形です。

5-6-13 ビューを対象としたSQL文の処理イメージ

FROM 句でビューを指定した SQL 文を実行する

```
1  SELECT COUNT(*)
2  FROM jan_feb_unique_products
```

↓

ビューを作成した元の SQL 文が、実体のあるテーブルに対して実行される

```
1  SELECT product_name FROM sample.small_jan
2  UNION DISTINCT
3  SELECT product_name FROM sample.small_feb
```

↓

jan_feb_unique_products

product_name
アジ
タコ
サバ
キス
タイ
ブリ
イカ

ビュー（＝仮想テーブル）が更新される

↓

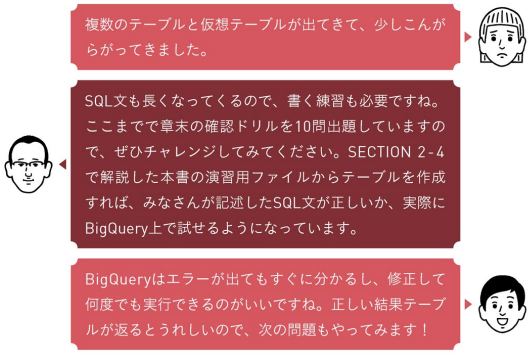
number_of_products
7

更新されたビューに対して SQL 文が実行される

このような「ビューを対象としたSQL文を実行すると、そのビューを作成したSQL文が再実行される」という性質は、次の動作を生む原因になります。

- ビューを作成した元のSQL文の実行に時間がかかると、ビューを対象としたSQL文の実行にも時間がかかる
- 実体のあるテーブルが更新されると、ビューを対象としたSQL文の実行結果が変化する

上記が望ましくない場合、仮想テーブルをビューとしてではなく、実体のあるテーブルとして保存するといいでしょう。SQL文の結果をテーブルとして保存する方法も、SECTION 2-6で解説しています。



SECTION

5-7 確認ドリル

問題 011

[sales] テーブルと [products] テーブルを結合して、カテゴリ（product_category）別の販売金額（revenue）の合計を集計し、金額の大きい順に並べてください。

問題 012

[sales] テーブルから商品ID（product_id）別の販売個数（quantity）の合計と、その合計がテーブル全体の販売個数の何%になっているかの割合（sales_share）を調べてください。[sales_share] は小数で構いません。結果テーブルは、商品ID（product_id）、販売個数の合計（sum_qty）、販売個数の割合（sales_share）の3カラムとし、[sales_share] の高い順に5レコードに絞り込んでください。

問題 013

[sales] テーブル、[products] テーブル、[customers] テーブルを結合して、性別（gender）および商品カテゴリ（product_category）ごとの販売金額の合計を取得してください。性別（gender）は「1」の場合に「男性」、「2」の場合に「女性」、それ以外は「不明」とします。並べ替えは、性別（昇順）、販売金額の合計（降順）として、結果テーブルは、性別（customer_gender）、商品カテゴリ（product_category）、販売金額の合計（sum_revenue）の3カラムとしてください。

問題 014

［sales］テーブルを利用して、2019年7〜9月には10個以上販売されていたのに、2019年10〜12月には10個以上売れなくなってしまった商品のリストを取得してください。結果テーブルは［product_id］だけの1カラムとします。

問題 015

［sales］テーブルと［product］テーブルを結合して、商品名（product_name）別に利益の大きい順に3つの商品を取り出してください。結果テーブルは、商品名（product_name）、販売個数の合計（sum_quantity）、販売金額の合計（sum_revenue）、コストの合計（sum_cost）、利益の合計（profit）の5カラムとします。

解答

解答 011

```sql
SELECT product_category
, SUM(revenue) AS sum_revenue
FROM sample.sales
INNER JOIN sample.products
USING (product_id)
GROUP BY product_category
ORDER BY 2 DESC
```

結果テーブル

行	product_category	sum_revenue
1	ケーキ	1918620
2	クッキー	916410
3	ゼリー	178190

[sales] テーブルの [product_id] と、[products] テーブルの [product_id] をキーとして結合することにより、[products] に存在する商品カテゴリ（product_category）で集計できるようになります。

[sales]テーブルの[product_id]に記録されている値がすべて、[products]テーブルの [product_id] にも存在しているため、上記のINNER JOINはLEFT OUTER JOINとしてもRIGHT OUTER JOINとしても同じ結果になります。

解答 012

```sql
WITH total_qty AS
(SELECT SUM(quantity) AS ttl_qty FROM sample.sales)

SELECT product_id
, SUM(quantity) AS sum_qty
, SUM(quantity) / MAX(ttl_qty) AS sales_share
FROM sample.sales
CROSS JOIN total_qty
GROUP BY product_id
ORDER BY 3 DESC
LIMIT 5
```

結果テーブル

行	product_id	sum_qty	sales_share
1	1	388	0.26575342465753427
2	2	252	0.17260273972602740
3	15	169	0.11575342465753424
4	14	134	0.09178082191780822
5	5	87	0.05958904109589041

まず、WITH句でテーブル全体の販売個数の合計（ttl_qty）を取得します。取得した仮想テーブルは［total_qty］と名付けており、1列1行、つまり1つの値で構成されています。そのうえで［sales］テーブルを［total_qty］を結合します。[sales］テーブルのすべてのレコードに［ttl_qty］の値を持ちたいので、結合のタイプにはCROSS JOINを使います。

結合してしまえば「同一レコードの別フィールド同士の計算」は簡単にできるので、[product_id］でグループ化を行い、SUM関数で［quantity］を合計した値を、販売金額の合計であるMAX(ttl_qty)で割って［sales_share］を求めます。

CROSS JOIN後、すべてのレコードに同じ値の［ttl_qty］が格納されている

ので、ここで利用するMAX関数は集計関数を利用して値を1つだけ取り出していることになり、MIN関数を使っても、AVG関数を使っても結果は同様になります。

別解として、以下のSQL文があります。こちらはWITH句で商品ID別の販売個数の合計と、テーブル全体の販売個数の合計という2つの仮想テーブルを作成し、メインのSQL文でシンプルにCROSS JOINと割り算のみをするという方法です。

```
WITH qty_by_product AS
(SELECT product_id, SUM(quantity) AS sum_qty
FROM sample.sales
GROUP BY product_id)
, total_qty AS
(SELECT SUM(quantity) AS ttl_qty FROM sample.sales)

SELECT product_id, sum_qty
, sum_qty / ttl_qty AS sales_share
FROM qty_by_product
CROSS JOIN total_qty
ORDER BY 3 DESC
LIMIT 5
```

解答 013

```sql
SELECT
CASE gender
WHEN 1 THEN "男性"
WHEN 2 THEN "女性"
ELSE "不明" END AS customer_gender
, product_category
, SUM(sales.revenue) AS sum_revenue
FROM sample.sales
LEFT JOIN sample.customers USING (user_id)
LEFT JOIN sample.products USING (product_id)
GROUP BY gender, product_category
ORDER BY 1, 3 DESC
```

結果テーブル

行	customer_gender	product_category	sum_revenue
1	女性	ケーキ	1612080
2	女性	クッキー	759470
3	女性	ゼリー	157710
4	男性	ケーキ	306540
5	男性	クッキー	156940
6	男性	ゼリー	20480

［sales］テーブルを左側として、顧客情報が格納された［customers］テーブルと商品情報が格納された［products］テーブルの2つを結合します。［customers］テーブルにある［gender］には、「1」「2」と数値が格納されていますが、CASE文で「男性」「女性」と指定することにより、分かりやすい結果テーブルになります。

性別（gender）と商品カテゴリ（product_category）の2つのフィールドで集計するため、GROUP BY句にも［gender］と［product_catgory］を指定することにも留意してください。

解答 014

```
WITH qty_jul_sep AS
(SELECT product_id
FROM sample.sales
WHERE date_time BETWEEN "2019-07-01" AND "2019-09-30"
GROUP BY product_id
HAVING SUM(quantity) >= 10)
, qty_oct_dec AS
(SELECT product_id
FROM sample.sales
WHERE date_time BETWEEN "2019-10-01" AND "2019-12-31"
GROUP BY product_id
HAVING SUM(quantity) >= 10)

SELECT product_id FROM qty_jul_sep
EXCEPT DISTINCT
SELECT product_id FROM qty_oct_dec
```

結果テーブル

行	product_id
1	5
2	11
3	14

2019年7～9月に販売個数（quantity）が「10」以上の商品と、2019年10～12月に販売個数（quantity）が「10」以上の商品を調べることがポイントです。日付の期間はBETWEEN句で指定します。

WITH句で条件に合致する2つの仮想テーブルを作成し、それらを集合演算のEXCEPT DISTINCTで差集合を作って商品リストを取得します。

解答 015

```
SELECT product_name
, SUM(quantity) AS sum_quantity
, SUM(revenue) AS sum_revenue
, SUM(quantity) * MAX(cost) AS sum_cost
, SUM(revenue) -
(SUM(quantity) * MAX(cost)) AS profit
FROM sample.sales
INNER JOIN sample.products
USING (product_id)
GROUP BY product_name
ORDER BY 5 DESC
LIMIT 3
```

結果テーブル

行	product_name	sum_quantity	sum_revenue	sum_cost	profit
1	レアチーズケーキ	388	762800	155200	607600
2	ベイクドチーズケーキ	252	547360	126000	421360
3	クッキー詰め合わせ	169	497400	101400	396000

［sales］テーブルと［products］テーブルを［product_id］をキーに結合したうえで、［products］テーブルにある商品名（product_name）でグループ化しています。単価をMAX(cost)で取得するところ、そしてコスト合計（sum_revenue）を求めるのに、販売個数（quantity）と単価（MAX(cost)）を掛け合わせるところは難しかったのではないでしょうか。

GROUP BY句でグループ化をすると、指標はすべて集計関数で括る必要があります。では、単価を求めるのにどのような集計関数で括ればよいかというと、複数レコードの同一の値から1つに集計できれば何でもよいため、上記解答のMAX(cost)のところは、MIN(cost)でもAVG(cost)でも同じ結果が得られます。集計関数には複数レコードにまたがる同一の値を1つだけの値に集約する機能もあることを、今一度確認してください。

CHAPTER 6

サブクエリ

6-1　サブクエリの記述と戻り値の利用
6-2　サブクエリの実践的な活用例
6-3　UNNEST関数によるGA4テーブルの変換
6-4　確認ドリル

本章で学ぶ「サブクエリ」は「副問い合わせ」とも呼び、仮想テーブルを柔軟に利用する方法です。SQL文を入れ子にして（ネストして）記述するので、SQLが苦手な人は、サブクエリでつまづく人も多いように思います。しかし、仮想テーブルと同様に、SQLでできることが何倍にも広がるので、ぜひついてきてください。

SECTION

6-1 サブクエリの記述と戻り値の利用

本書での学習も後半戦に入ってきました。ここでは「サブクエリ」をマスターしてもらいますが、まずは基本を理解してください。

はい！「クエリ」はSQL文のことでしたよね。サブのSQL文、ということでしょうか？

その通りです。クエリは「問い合わせ」という意味でもあるので、メイン（主）に対するサブ（副）として「副問い合わせ」とも呼ばれます。

仮想テーブルを柔軟に作成し、利用できる

SECTION 5-6では仮想テーブルについて学びましたが、仮想テーブルの作成方法には、WITH句、ビュー、サブクエリの3つがあると述べました。サブクエリは、すでに学んだ2つの方法よりも、さらに柔軟に仮想テーブルを作成・利用できるSQLの機能です。

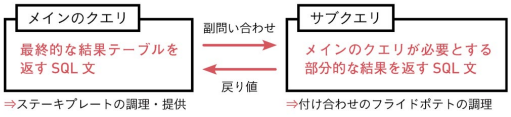
概念的には、次ページの図［6-1-1］のようになります。最終的な結果テーブルを返すSQL文をメインのクエリとすると、サブクエリは**メインのクエリが必要とする部分的な結果を返すSQL文**、と表現できます。

例え話をします。あるレストランで、お客さまに「フライドポテトを添えたステーキプレート」を提供するとしましょう。厨房ではメインのシェフがステーキを焼きますが、同時に、サブのシェフにポテトを揚げるように指示します。このフライドポテトの調理をさせる指示が「副問い合わせ」に当たります。

そして、サブのシェフの調理が終わり、完成したフライドポテトが「戻り値」

に当たります。メインのシェフは、自分が調理したステーキにフライドポテトを添えて、ステーキプレートを提供します。このステーキプレートが、最終的な結果テーブルに当たるわけです。

6-1-1 メインのクエリとサブクエリの関係



SQLに話を戻しましょう。SECTION 5-6の［5-5-6］では、UNIONを解説するための小さなテーブルとして「1月の販売テーブル」(small_jan）を提示しました。また、このテーブルから販売個数（qty）の全体の平均値よりも多く売れた商品を取得するために、仮想テーブルを利用するSQL文として［5-6-7］を記述しました。以下に再掲します。

```
1  WITH total_avg_qty AS(
2  SELECT AVG(qty) AS avg_qty FROM sample.small_jan)
3
4  SELECT * FROM sample.small_jan
5  CROSS JOIN total_avg_qty
6  WHERE qty > avg_qty
```

このSQL文についてはSECTION 5-6で解説した通りですが、①WITH句で[qty] の平均値を格納した仮想テーブルを作成し、②1月の販売テーブルと仮想テーブルをCROSS JOINし、③結合したテーブルのレコードをWHERE句で絞り込む、という3つの手順を踏むのは、やや面倒ではあります。

一方、サブクエリを利用すれば、①と②の手順を省略した以下のSQL文［6-1-2］で、同じ処理を実現できます。

6-1-2 サブクエリの記述例

```
1  SELECT * FROM sample.small_jan
2  WHERE qty > (SELECT AVG(qty) FROM sample.small_jan)
```

何というシンプルさでしょう。上記の2行目に注目すると、WHERE句で［qty］と比較する対象として、半角カッコで囲まれた、SELECT句から始まる別のSQL文が指定されています。この部分こそがサブクエリです。そして、サブクエリを実行した結果として「AVG(qty)」を［qty］の比較対象とすることで、目的の結果テーブルが得られます。

このように、1つのSQL文の中で、別のSQL文を利用するのがサブクエリです。**SQL文の中にSELECT句が複数記述されている場合には、サブクエリが使われている**と考えてください。

サブクエリには戻り値があり、上記の［6-1-2］では「AVG(qty)」による集計値が該当します。これは「1つの値のみ」ですが、サブクエリに記述するSQL文によって、図［6-1-3］に示した「1行n列のリスト」「m行n列の表」の形をとります。以降で1つずつ解説しましょう。

6-1-3 サブクエリの戻り値の種類

1つの値のみ

28
0.38
東京
18254
2109-12-03

1列n行のリスト

pref	value
青森	254
福島	584
奈良	447
愛知	112
長野	321

m列n行の表

pref	value	has_sea
青森	254	true
福島	584	true
奈良	447	false
愛知	112	true
長野	321	false

サブクエリで「1つの値のみ」を取得する

まずは以下のテーブル［6-1-4］を対象に「1つの値のみ」を取得し、サブクエリとして利用する方法を見ていきます。これまで通り、SQLの動作が理解しやすいように、9レコードの小さなテーブルとなっています。

北海道、東京、千葉における平成29年から令和元年の3年間の最低賃金がまとまっており、都道府県が［pref］、年度が［year］、最低賃金が［min_wage］というフィールドにそれぞれ格納されています。テーブル名は［small_wage］とします。ちなみに、データは厚生労働省が発表している本物です。

6-1-4 3年間の都道府県別最低賃金テーブル

pref	year	min_wage
東京	H29	958
東京	H30	985
東京	R1	1013
北海道	H29	810
北海道	H30	835
北海道	R1	861
千葉	H29	868
千葉	H30	895
千葉	R1	923

このテーブルから「1つの値のみ」を取得するには、次の3つの方法が考えられます。いずれも、これまでに学んできたSQL文で実現できるので、一気におさらいしていきます。

- レコードを絞り込む
- 集計関数を利用する
- 絞り込みと集計関数を併用する

レコードを絞り込む方法の具体例としては、次ページの［6-1-5］のSQL文が挙げられます。WHERE句を使って［pref］を「東京」に、［year］を「H29」

に絞り込み、「958」という1つの値のみを取得しています。

集計関数を利用する方法の例は、その下の［6-1-6］のようになります。MAX関数を利用し、[min_wage］の最大値である「1013」を取得しています。

絞り込みと集計関数を併用する方法では、次ページの［6-1-7］のようなSQL文が想定できます。[pref］を「東京」に絞り込み、MIN関数で［min_wage］の最小値を求めています。結果は「958」となります。

6-1-5 レコードを絞り込む

```
1  SELECT min_wage FROM sample.small_wage
2  WHERE pref = "東京" AND year = "H29"
```

結果テーブル

行	min_wage
1	958

東京の平成29年の最低賃金を取得できた

6-1-6 集計関数を利用する

```
1  SELECT MAX(min_wage) AS max_min_wage
2  FROM sample.small_wage
```

結果テーブル

行	max_min_wage
1	1013

すべての最低賃金の最大値を取得できた

6-1-7 絞り込みと集計関数を併用する

```
1  SELECT MIN(min_wage) AS min_tokyo_min_wage
2  FROM sample.small_wage
3  WHERE pref = "東京"
```

結果テーブル

行	min_tokyo_min_wage
1	958

東京の最低賃金の最小値を取得できた

では、このような「1つの値のみ」をサブクエリからの戻り値として、メインのクエリで利用してみます。方法としては、①SELECT句で新しいフィールドを作成する、②SELECT句でフィールド同士を計算する、③WHERE句での絞り込みに利用するの3つがあり、順に解説します。

▶SELECT句で新しいフィールドを作成する

1つ目の方法では、サブクエリとして取得した「1つの値のみ」を、メインのクエリから得られる結果テーブルに新しいフィールドとして作成します。次ページのSQL文［6-1-8］の2行目に注目してください。

半角カッコで囲まれた、SELECT句から始まるサブクエリが記述されています。また、取得する値として最低賃金の平均値「AVG(min_wage)」が指定され、[avg_min_wage] と別名を付けたフィールドとして結果テーブルに格納されました。

既存のテーブルの全フィールドに新しいフィールドを追加するので、メインのクエリは「SELECT *, 追加するフィールド名」の構文となっています（SECTION 3-2を参照）。

6-1-8 サブクエリの戻り値で新しいフィールドを作成する

```
1  SELECT *
2  , (SELECT AVG(min_wage) FROM sample.small_wage)
3  AS avg_min_wage
4  FROM sample.small_wage
```

結果テーブル

行	pref	year	min_wage	avg_min_wage
1	東京	H29	958	905.3333333333334
2	東京	H30	985	905.3333333333334
3	東京	R1	1013	905.3333333333334
4	北海道	H29	810	905.3333333333334
5	北海道	H30	835	905.3333333333334
6	北海道	R1	861	905.3333333333334
7	千葉	H29	868	905.3333333333334
8	千葉	H30	895	905.3333333333334
9	千葉	R1	923	905.3333333333334

最低賃金の平均値が[avg_min_wage]フィールドとして作成された

▶SELECT句でフィールド同士を計算する

2つ目の方法では、サブクエリとして取得した「1つの値のみ」を定数のように扱い、メインのクエリでの演算に利用します。次ページのSQL文[6-1-9]の、同じく2行目に注目してください。

[min_wage]からサブクエリの戻り値である「AVG(min_wage)」の結果を差し引き、都道府県ごとの最低賃金と、最低賃金の平均値との差を求めています。その結果は[diff_from_table_avg]フィールドとして、結果テーブルに格納されました。

6-1-9 サブクエリの戻り値で演算する

```
1  SELECT *
2  , min_wage - (SELECT AVG(min_wage) FROM sample.small_wage)
3  AS diff_from_table_avg
4  FROM sample.small_wage
```

結果テーブル

行	pref	year	min_wage	diff_from_table_avg
1	東京	H29	958	52.66666666666663
2	東京	H30	985	79.66666666666663
3	東京	R1	1013	107.66666666666663
4	北海道	H29	810	-95.33333333333337
5	北海道	H30	835	-70.33333333333337
6	北海道	R1	861	-44.33333333333337
7	千葉	H29	868	-37.33333333333337
8	千葉	H30	895	-10.333333333333371
9	千葉	R1	923	17.66666666666663

各レコードの最低賃金と、テーブル全体の最低賃金の平均値の差を取得できた

▶WHERE句での絞り込みに利用する

3つ目の方法では、サブクエリの戻り値を定数と見立てて、メインのクエリのWHERE句における絞り込みの条件として組み込みます。次ページのSQL文[6-1-10]の3〜4行目に注目してください。

WHERE句で元のテーブルの［min_wage］と、サブクエリの戻り値である「AVG(min_wage)」を比較し、最低賃金が全体の平均値よりも高いレコードだけに結果テーブルを絞り込んでいます。サブクエリの戻り値を、あたかも定数のように利用していることが理解できると思います。

6-1-10 サブクエリの戻り値で絞り込む

```
1  SELECT *
2  FROM sample.small_wage
3  WHERE min_wage
4  > (SELECT AVG(min_wage) FROM sample.small_wage)
```

結果テーブル

行	pref	year	min_wage
1	東京	H29	958
2	東京	H30	985
3	東京	R1	1013
4	千葉	R1	923

最低賃金が全体の平均値よりも高い都道府県のみに絞り込んだ

サブクエリで「1列n行」のリストを取得する

続いて、サブクエリの戻り値の種類のうち、「1列n行のリスト」を利用する方法を見ていきます。

戻り値が「1列n行のリスト」なので、SELECT句にはフィールド名を1つだけ記述すればよく、「1つの値のみ」のように集計関数を使って工夫する必要はありません。しかし、使い方は少々複雑です。

例として、[6-1-4]で提示した3年間の都道府県別最低賃金テーブルを対象に、「最低賃金が1年でも全体の平均値を上回ったことのある都道府県の、3年間での最低賃金の平均値」を取得したいとします。「最低賃金が1年でも全体の平均値を上回ったことのある都道府県」に該当するのは、[6-1-10]の結果テーブルから「東京」と「千葉」であることが分かります。

よって、その2県に絞り込んだうえで、それぞれの3年間での最低賃金の平均値を取得すればよい、という思いに至ります。そのためのSQL文は、WHERE句またはHAVING句にIN演算子を組み合わせた、次ページの[6-1-11]または[6-1-12]として記述できるでしょう。

6-1-11 WHERE句で特定の値を指定する

```
SELECT pref, AVG(min_wage) AS avg_min_wage_by_pref
FROM sample.small_wage
WHERE pref IN ("東京", "千葉")
GROUP BY pref
```

6-1-12 HAVING句で特定の値を指定する

```
SELECT pref, AVG(min_wage) AS avg_min_wage_by_pref
FROM sample.small_wage
GROUP BY pref
HAVING pref IN ("東京", "千葉")
```

しかし、上記のWHERE句およびHAVING句の条件として指定している「東京」「千葉」は、ここでの例が小さなテーブルだから自明なだけで、実務で利用するような大きなテーブルでは不明なはずです。

よって、「東京」「千葉」といった特定の値ではなく、「最低賃金が1年でも全体の平均値を上回ったことのある都道府県のリスト」を取得する必要があります。それを実現するSQL文は、次ページの［6-1-13］となります。

6-1-13 条件を満たす値のリストを取得する

```
1  SELECT pref
2  FROM sample.small_wage
3  WHERE min_wage
4  > (SELECT AVG(min_wage) FROM sample.small_wage)
```

結果テーブル

行	pref
1	東京
2	東京
3	東京
4	千葉

最低賃金が１年でも全体の平均値を上回ったことのある都道府県のリストを取得できた

上記の結果テーブルの通り、「東京」「千葉」という値が「1列n行のリスト」として取得できました。

これをサブクエリの戻り値として利用し、「最低賃金が１年でも全体の平均値を上回ったことのある都道府県の、３年間での最低賃金の平均値」を取得するSQL文は、次ページの［6-1-14］または［6-1-15］となります。

前者はWHERE句、後者はHAVING句を利用した場合で、それぞれのIN演算子での比較対象として前掲の［6-1-13］をサブクエリとして記述しています。［6-1-13］でもサブクエリが使われていたので、二重に利用されていることが分かるでしょうか。結果、「東京」「千葉」の３年間での最低賃金の平均値が［avg_min_wage］フィールドとして取得できました。

複数の構文が組み合わさっているので、少々難しいと感じたかもしれません。今、新しく学んでいることを整理すると、**IN演算子の比較対象には、サブクエリで取得した「1列n行のリスト」を利用できる**、となります。この点をしっかり覚えてください。

6-1-14 WHERE句にサブクエリを組み込む

```
SELECT pref, AVG(min_wage) AS avg_min_wage
FROM sample.small_wage
WHERE pref IN (
SELECT pref
FROM sample.small_wage
WHERE min_wage
> (SELECT AVG(min_wage) FROM sample.small_wage)
)
GROUP BY pref
```

6-1-15 HAVING句にサブクエリを組み込む

```
SELECT pref, AVG(min_wage) AS avg_min_wage
FROM sample.small_wage
GROUP BY pref
HAVING pref IN (
SELECT pref
FROM sample.small_wage
WHERE min_wage
> (SELECT AVG(min_wage) FROM sample.small_wage)
)
```

結果テーブル

行	pref	avg_min_wage
1	東京	985.3333333333334
2	千葉	895.3333333333334

「東京」と「千葉」の３年間での最低賃金の平均値を取得できた

サブクエリで「m列n行の表」を取得する

サブクエリからの戻り値として「1つのみの値」と「1列n行のリスト」を学んできましたが、利用頻度がもっとも高いのが、ここで学ぶ「m列n行の表」です。このサブクエリは「集計の集計」を行うために使われます。

まず、基本的な構文として以下の［6-1-16］を見てください。ポイントは、3行目に記述されているサブクエリが、メインのクエリである1行目のSELECT句を対象とするFROM句として記述されている点です。つまり、**サブクエリからの戻り値である「m列n行の表」を仮想テーブルと見立て、それを対象としてメインのクエリを実行する**形になっています。

6-1-16 「m列n行の表」を戻り値とするサブクエリの構文

```
1  SELECT フィールド名
2  FROM (
3  SELECT フィールド名 FROM テーブル名
4  )
```

SQL文の対象とする実体のあるテーブルの例として、次ページの［6-1-17］を利用します。これは令和元年における都道府県別の最低賃金をまとめたもので、47都道府県に相当するレコードがあります。

このテーブルから「最低賃金が800円を超える都道府県の数と、それらの最低賃金の平均値」を取得したいとしましょう。テーブル名は［small_wage_reiwa1］とします。

6-1-17 令和元年の都道府県別最低賃金テーブル

pref_id	pref	min_wage
1	北海道	861
2	青森	790
3	岩手	790
4	宮城	824
5	秋田	790
⋮	⋮	⋮
43	熊本	790
44	大分	790
45	宮崎	790
46	鹿児島	790
47	沖縄	790

もし、知りたいことが「最低賃金が800円を超える都道府県名と、それぞれの最低賃金」なのであれば、サブクエリを利用することなしに、以下の［6-1-18］の通りに記述できます。SELECT句で［pref］と［min_wage］の2つのフィールドを指定し、WHERE句で［min_wage］が「800」より多いという条件を指定すればいいので、簡単です。

6-1-18 取得するフィールドと条件を指定する

```
1  SELECT pref, min_wage FROM sample.small_wage_reiwa1
2  WHERE min_wage > 800
```

しかし、本当に取得したいのは「都道府県の数」と「最低賃金の平均値」なので、COUNT関数とAVG関数による集計が必要になります。2つの関数の対象となるのは、［6-1-18］の結果テーブルです。

よって、それがメインのクエリの実行対象である仮想テーブルとなるよう、FROM句のサブクエリとして［6-1-18］を記述します。そのうえで、メインのクエリで全体のレコード数と、［min_wage］の平均値を求めればよい、と

いうことになります。具体的なSQL文は以下の［6-1-19］です。

なお、1行目にあるSELECT句では、「COUNT(*)」の代わりに「COUNT(pref)」または「COUNT(DISTINCT pref)」として、都道府県の値の個数を数えても同じ結果になります。4〜5行目のサブクエリと、1〜3行目のメインのクエリを切り離して考えると理解しやすいと思います。

6-1-19 FROM句にサブクエリを組み込む

```
1  SELECT COUNT(*) AS number_of_pref
2  , AVG(min_wage) AS avg_min_wage
3  FROM (
4  SELECT pref, min_wage FROM sample.small_wage_reiwa1
5  WHERE min_wage > 800
6  )
```

結果テーブル

行	number_of_pref	avg_min_wage
1	30	871.6999999999999

最低賃金が 800 円を超える都道府県の数と、それらの最低賃金の平均値を取得できた

STEP UP

複数のサブクエリを入れ子にする

サブクエリは複数を入れ子にすることもできます。次ページの［6-1-20］では、各都道府県の最低賃金をテーブル全体の最低賃金の平均値と比較した結果に基づき、都道府県を「平均未満都道府県」と「平均以上都道府県」にグループ化し、「各都道府県の最低賃金とテーブル全体の平均値の差の平均」を求めています。複数のサブクエリをネストで

きることを示すため、あえて過剰なほどに入れ子にしたサブクエリを記述しています。

このように**他人が書いたSQL文を読み解くには、「内側から読み解く」のが基本**です。入れ子のいちばん内側にあるクエリを読み解き、それが1つ外側にあるクエリにどのような戻り値を返しているかを考えていきます。実務で使うような複雑なSQL文を外側から読み解いていては、非常に時間がかかることがあります。この基本はぜひ覚えてください。

6-1-20 多重の入れ子にしたサブクエリ

```
SELECT pref_cat, AVG(diff) AS diff_from_avg
FROM
(
  SELECT *
    ,CASE
     WHEN diff >=0 THEN "平均以上都道府県"
     ELSE "平均未満都道府県"
    END AS pref_cat
  FROM (
    SELECT *, min_wage - avg_min_wage AS diff
    FROM (
     SELECT pref
            , min_wage
            , (SELECT AVG(min_wage)
               FROM sample.small_wage_reiwa1
               ) AS avg_min_wage
     FROM sample.small_wage_reiwa1
        )
    )
)
GROUP BY pref_cat
```

結果テーブル

行	pref_cat	diff_from_avg
1	平均未満都道府県	-34.93470286133537
2	平均以上都道府県	56.28368794326233

最低賃金の平均値との差の平均値を取得できた

［6-1-20］のSQL文は、BigQueryのクエリエディタに記述したスクリーンショットとしていますが、これは「**インデント**」を理解してもらうためです。もっとも左から始まっている行と、3文字目、5文字目から始まっている行がありますが、それらの行のように、左から数文字分を空けて行を記述することを「インデントを入れる」といいます。

適切にインデントを入れることで、SQL文の読みやすさを向上させることができます。絶対的な決まりはないのですが、ここでは次のルールでインデントを入れています。

- 内側のクエリを始めるときは2文字分インデントする
- 同一のレベルにある命令は同じ文字数分インデントする
- 終了カッコは開始カッコがある行と同じ文字数分インデントする

では、SQL文を読み解いてみましょう。内側から順に説明します。

1. もっとも内側：14〜16行目
   ⇒[6-1-17]のテーブル全体から最低賃金の平均値を取得し、[avg_min_wage]という別名で外側のクエリに「1つの値のみ」を返している。
2. 上記1の外側：12〜17行
   ⇒[6-1-17]のテーブルの[pref][min_wage]、および内側のクエリから渡された[avg_min_wage]という3カラムのテーブルを作っている。
3. 上記2の外側：10〜18行
   ⇒内側のクエリから渡されたテーブルの全フィールドに加えて、[min_wage]と[avg_min_wage]の差を[diff]という名前で取得した4列47行のテーブルを作っている。
4. 上記3の外側：4〜19行
   ⇒内側のクエリから渡されたテーブルの各行について、[diff]が「0」以上であれば「平均以上都道府県」という文字列を、そうでなければ「平均未満都道府県」の文字列を格納する、[pref_cat]というフィールドを追加している。
5. もっとも外側：1〜21行
   ⇒内側のクエリから渡されたテーブルを[pref_cat]でグループ化し、[diff]の平均値を計算して[diff_from_avg]というフィールドで表示している。

結果として、[pref_cat] ごとに最低賃金の平均値との差の平均値である [diff_from_avg] を取得できています。

これと同じ処理を、テーブル全体の平均を求めるサブクエリ以外は使わず、以下の [6-1-21] のように記述しても同じ結果が取り出せます。より少ない行数で同じ結果が導けるので、以下のSQL文のほうが優れているともいえます。

ただ、サブクエリを利用したほうが、中間的な値を確認しながらSQL文を完成させていけるので、間違いが少なく、何をしているのかが分かりやすいというメリットもあります。適宜使い分けてください。

6-1-21 もっとも少ない入れ子にしたサブクエリ

```
SELECT
CASE
WHEN min_wage
- (SELECT AVG(min_wage)
FROM sample.small_wage_reiwa1) >= 0
THEN "平均以上都道府県"
ELSE "平均未満都道府県"
END AS pref_cat
, AVG(min_wage - (SELECT AVG(min_wage)
FROM sample.small_wage_reiwa1))
AS avg_diff
FROM sample.small_wage_reiwa1
GROUP BY pref_cat
```

SECTION

6-2 サブクエリの実践的な活用例

本節では、サブクエリの応用について学びます。「縦持ち」「横持ち」に分けられるテーブル構造の変換と、それを利用した3つの実践例の、大きく分けて2本立てになっています。

サブクエリはちょっと難しいのですが、実践例は気になりますね。どんな業務で使えるのでしょうか？

ここで紹介するのは、注文履歴などECサイトを想定した例と、カレンダーを使った例です。まずはテーブル構造の変換から理解してください。

サブクエリの応用でできること

本節ではサブクエリのより実践的な活用例として、次に挙げるテクニックについて学びます。最初に大きな流れを理解してください。

- ●サブクエリを利用したテーブル構造の変換
  - ○データの持ち方の2つの種類
  - ○横持ちのテーブルを縦持ちに変換する
  - ○縦持ちのテーブルを横持ちに変換する
- ●サブクエリを利用した実践的なSQLの例
  - ○注文履歴をユーザー単位のテーブルに変換する
  - ○特定商品の有無別に平均注文金額を比較する
  - ○作業記録とカレンダーから土日以外の休業日を調べる

データの持ち方の2つの種類

まず理解してほしいのが、テーブルにおける「データの持ち方」についてです。データベース上のデータの持ち方には「**縦持ち**」と「**横持ち**」と呼ばれる2つの種類があり、以下の［6-2-1］のように表せます。

左側にある縦持ちのテーブルは、前節の［6-1-4］として提示した3年間の都道府県別最低賃金テーブルと同じものです。そして、右側にある横持ちのテーブルも、3年間の都道府県別最低賃金テーブルです。構造が異なるだけで、持っているデータとしては同じことが分かるでしょうか。

6-2-1 「縦持ち」と「横持ち」の違い

縦持ち

pref	year	min_wage
東京	H29	958
東京	H30	985
東京	R1	1013
北海道	H29	810
北海道	H30	835
北海道	R1	861
千葉	H29	868
千葉	H30	895
千葉	R1	923

横持ち

pref	H29 min_wage	H30 min_wage	R1 min_wage
東京	958	985	1013
北海道	810	985	861
千葉	868	895	923

例えば、このテーブルに令和2年のデータを加えるとき、縦持ちではカラム数はそのままで、レコード数が下に増えていきます。逆に、横持ちではレコード数はそのままで、カラム数が右に増えていきます。

データベースの世界では、縦持ちが一般的です。データベースはレコード数を増やすのは得意ですが、カラム数を増やすのは得意ではないためです。

しかし、アプリケーションから出力したCSVファイルや、印刷したときに見やすいように作成したExcelファイルをCSV形式に変換したファイルなどは、横持ちの構造になっている場合があります。そのようなファイルをアップロードしてBigQuery上にテーブルを作成すると、横持ちのテーブルが作成されることもあるでしょう。

縦持ちのテーブルと横持ちのテーブルの、どちらかが優れているわけではありません。ただ、次のように「SQLでやりやすいことが異なる」「縦持ちのほうがより一般的」ということを覚えてください。

▶縦持ちのテーブル

前節でも登場した都道府県別の最低賃金の平均値を取得するなど、「集計計算がしやすい」のが特徴です。横持ちのテーブルで同様の集計をしようとすると、「(フィールド1の値＋フィールド2の値＋フィールド3の値) ÷3」といった計算式の記述が必要になり、面倒です。

▶横持ちのテーブル

例えば、都道府県ごとの最低賃金について「R1」の「H30」に対する伸び率を求めるような、「フィールド同士の計算がしやすい」という特徴があります。一方、縦持ちのテーブルで同じことをやろうとすると、サブクエリやSELF JOIN（SECTION 5-4を参照）を利用する必要があります。

横持ちのテーブルを縦持ちに変換する

得られたデータから作成したテーブルが横持ちだったとき、集計作業の効率を上げたい場合や、TableauなどのBIツールで集計を行いたい場合には、縦持ちに変換したほうが望ましいでしょう。そのような場合に使われる定番が「縦持ち・横持ち変換」と呼ばれるSQL文です。

対象となる横持ちのテーブルを以下の［6-2-2］として紹介します。

6-2-2 横持ちのテーブル

order_id	fashion_revenue	zakka_revenue	food_revenue
123	18600	null	5800
124	null	2400	8800
125	6900	2900	11200
126	4200	3800	4500
127	null	9800	null

この［small_revenue］テーブルを、［order_id］［product_cat］［revenue］という3つのフィールドを持つ、縦持ちのテーブルに変換するとしましょう。

基本的な考え方は、ファッションのみ、雑貨のみ、食品のみの結果テーブルをそれぞれサブクエリで作成し、それらをUNION ALLで「縦につなぐ」ことです。ひとまずファッションのみの結果テーブルを取得するSQL文を記述すると、以下の［6-2-3］となります。

［order_id］はそのままで、新しく作成する［product_cat］フィールドに「fasion」という値を格納、［revenue］フィールドに［fashion_revenue］の値を格納しています。結果、ファッションのみを縦持ちに変換したテーブルが作成されました。

6-2-3 ファッションのみを縦持ちに変換する

```
1  SELECT order_id
2  , "fashion" AS product_cat
3  , fashion_revenue AS revenue
4  FROM sample.small_revenue
5  ORDER BY order_id
```

結果テーブル

行	order_id	product_cat	revenue
1	123	fashion	18600
2	124	fashion	*null*
3	125	fashion	6900
4	126	fashion	4200
5	127	fashion	*null*

ファッションのみを縦持ちのテーブルに変換できた

同様に、雑貨のみの結果テーブル、食品のみの結果テーブルをサブクエリで作成し、UNION ALLで重複を許可した和集合を作成するSQL文は次ページの［6-2-4］となります。結果、5つの注文IDと3つの商品カテゴリによる、全15レコードの縦持ちのテーブルが完成しました。

6-2-4 縦持ちにしたテーブルから和集合を作成する

```
SELECT order_id
, "fashion" AS product_cat
, fashion_revenue AS revenue
FROM sample.small_revenue
UNION ALL
SELECT order_id
, "zakka"
, zakka_revenue
FROM sample.small_revenue
UNION ALL
SELECT order_id
, "food"
, food_revenue
FROM sample.small_revenue
ORDER BY 1, 2
```

結果テーブル

行	order_id	product_cat	revenue
1	123	fashion	18600
2	123	food	5800
	123	zak	null
13	127	fashion	null
14	127	food	null
15	127	zakka	9800

すべての商品カテゴリを縦持ちのテーブルに変換できた

UNION の場合、カラム名は最初の SELECT 句で指定した別名となるため、２つ目以降の SELECT 句で指定する必要はない

このように横持ちから縦持ちに変換できれば、例えば「商品カテゴリごとの販売金額の合計値と平均値を求める」といった集計が簡単に行えます。次ページの［6-2-5］は、横持ちから縦持ちへの変換と、商品カテゴリごとの販売金額の合計値・平均値の集計を一気に行うSQL文です。サブクエリが読み解き

やすいようにインデントを入れています。

6〜19行目がサブクエリで、[6-2-4] と同じSQL文になっています。このサブクエリの戻り値として得られる縦持ちのテーブル（m行n列の表）を対象に、[product_cat] でグループ化してSUM関数とAVG関数で [revenue] の合計値・平均値を取得しています。

6-2-5 サブクエリで縦持ちに変換して集計値を取得する

```
SELECT product_cat
, SUM(revenue) AS sum_rev
, AVG(revenue) AS avg_rev
FROM
(
  SELECT order_id
  , "fashion" AS product_cat
  , fashion_revenue AS revenue
  FROM sample.small_revenue
  UNION ALL
  SELECT order_id
  , "zakka"
  , zakka_revenue
  FROM sample.small_revenue
  UNION ALL
  SELECT order_id
  , "food"
  , food_revenue
  FROM sample.small_revenue
)
GROUP BY product_cat
```

結果テーブル

行	product_cat	sum_rev	avg_rev
1	food	30300	7575.0
2	zakka	18900	4725.0
3	fashion	29700	9900.0

縦持ちに変換したテーブルから
合計値・平均値を取得できた

縦持ちのテーブルを横持ちに変換する

今度は、縦持ちのテーブルを横持ちに変換してみましょう。対象となるテーブルは以下の［6-2-6］で、名前は［small_ad］とします。

3つのフィールドがあり、[ad_id］は広告の識別ID、[phase］は改善施策の実施前後、[bounce_rate] は直帰率のことです。広告「A」は直帰率が改善、「B」はやや悪化したことが見てとれますが、「before」と「after」の変化率は、このテーブルの状態から直接は取り出せません。

そこで、テーブルを縦持ちから横持ちに変換し、フィールド同士の計算で変化率を求めてみましょう。完成した横持ちのテーブルをイメージすると、[ad_id]、[before］の直帰率、[after］の直帰率という3カラムを持つテーブルを戻り値として返すサブクエリがあれば、変化率を簡単に計算できそうです。

6-2-6 縦持ちのテーブル

ad_id	phase	bounce_rate
A	before	0.68
A	after	0.62
B	before	0.52
B	after	0.55

そこで、サブクエリとするSQL文では、SELECT句で3つのフィールドを作成します。[ad_id］はそのままで、[before_index］に改善施策実施前の直帰率、[after_index］に改善施策実施後の直帰率を格納します。

SQL文は次ページの［6-2-7］の通りで、IF句を利用したグループ化によって［before_index］と［after_index］という2つのフィールドを作成しています。結果として、縦持ちから横持ちに変換したテーブルを取得できました。

一部に「null」が残っていますが、[ad_id] ごとにグループ化し、集計関数でまとめてしまえば消えます。

6-2-7 横持ちに変換したテーブルを取得する

```
1  SELECT ad_id
2  , IF (phase = "before", bounce_rate, null)
3  AS before_index
4  , IF (phase = "after", bounce_rate, null)
5  AS after_index
6  FROM sample.small_ad
```

結果テーブル

行	ad_id	before_index	after_index
1	A	0.68	*null*
2	A	*null*	0.62
3	B	0.52	*null*
4	B	*null*	0.55

3つのフィールドを持つ横持ちのテーブルに変換できた

最終的に、元のテーブルを縦持ちから横持ちに変換しつつ、広告IDごとに改善施策実施前後の直帰率の変化率を求めるSQL文は、次ページの［6-2-8］となります。こちらもインデントを入れています。

8〜13行目がサブクエリで、［6-2-7］のSQL文が記述されています。その外側にあるメインのクエリで［ad_id］によるグループ化を行い、4行目でフィールド同士の計算をした結果、直帰率の変化率が［change_point］フィールドとして求められています。結果テーブルでは、広告「A」は直帰率が6ポイント下がり（改善し）、広告「B」は3ポイント上がった（悪化した）ことが確認できました。

なお、グループ化した［ad_id］から唯一の値を取り出すために、MAX関数を利用しています。「null」は集計関数で無視されるため、MIN関数でもAVG関数でも同じ結果になります（この例ではSUM関数でも同じ）。

6-2-8 サブクエリで縦持ちに変換してフィールド同士を計算する

```
SELECT ad_id
, MAX(before_index) AS before_index
, MAX(after_index) AS after_index
, (MAX(after_index) - MAX(before_index)) * 100
AS change_point
FROM
(
  SELECT ad_id
  ,IF (phase = "before", bounce_rate, null)
  AS before_index
  ,IF (phase = "after", bounce_rate, null)
  AS after_index
  FROM sample.small_ad
)
GROUP BY ad_id
```

結果テーブル

行	ad_id	before_index	after_index	change_point
1	A	0.68	0.62	-6.000000000000005
2	B	0.52	0.55	3.0000000000000027

横持ちに変換したテーブルからフィールド同士の集計値を取得できた

ここまでで、縦持ち・横持ち変換の解説は終了です。次からは、サブクエリの応用による業務での実践例を見ていきます。

注文履歴をユーザー単位のテーブルに変換する

サブクエリを応用しつつ、グループ化、集計関数、UNIONといった、これまで学んだテクニックを利用してテーブル構造の変換ができるようになりました。ここからは、より業務に近い実践例を3つ見ていきます。

実践例の1つ目は、以下の［6-2-9］のように注文履歴がまとまったテーブル（[small_order_date] テーブル）を対象にします。現実世界の販売を記録するデータベースも、このような体裁になっていることが一般的です。

6-2-9 注文履歴テーブル

user_id	order_id	order_date	item_cat	revenue
ABC	123	2020-3-5	ファッション	10000
ABC	124	2020-4-10	ファッション	12000
ABC	124	2020-4-10	雑貨	3900
ABC	125	2020-5-12	グルメ	5800
STU	126	2020-5-13	グルメ	8600
STU	127	2020-5-27	ファッション	3900
STU	127	2020-5-27	雑貨	6600
XYZ	128	2020-6-1	ファッション	2900
XYZ	128	2020-6-1	ファッション	6900
XYZ	129	2020-6-19	雑貨	38000

[user_id] フィールドにユニークな値が3種類あることから、顧客は3人いると分かります。また、[order_id] によって注文が区別されていること、同一の注文でも複数の商品カテゴリ（item_cat）で購入されていればレコードが分かれることも理解できます。

この全部で10レコードのテーブルを元に、1ユーザーに対して1レコードとなるテーブルを作成し、ユーザーの特性に応じたカテゴライズをしたいとします。実務において、こうしたユーザーごとのデータを得たいケースとしては、次の2つが考えられるでしょう。

- 特性に応じたアプローチができるよう、ユーザーのカテゴリに対してメールマガジンの内容を変えて配信したい

●「このユーザーは休眠するか？」「このユーザーは優良顧客になるか？」など、機械学習でユーザー単位での予測を行いたい

基本的な進め方は［user_id］でグループ化し、集計によって値を1つにまとめるという流れになります。最終的な結果テーブルにおいて、どのようなフィールドと値を取得したいのかを整理すると、次のようになります。番号は左から数えた列番号です。

1. ［user_id］
   ⇒ユーザーを識別するID。
2. ［first_purchase_date］
   ⇒初回購入日。1ユーザーが複数の購入日（order_date）を持っている場合、もっとも小さい日付を取得して求める。
3. ［orders］
   ⇒購入回数。ユニークな［order_id］の個数をカウントすることで求める。
4. ［sum_revenue］
   ⇒購入金額合計。ユーザーごとの購入金額を合計して求める。
5. ［fashion_revenue］
   ⇒「ファッション」カテゴリの購入金額合計。
6. ［zakka_revenue］
   ⇒「雑貨」カテゴリの購入金額合計。
7. ［gourumet_revenue］
   ⇒「グルメ」カテゴリの購入金額合計。
8. ［user_cat］
   ⇒ユーザーを4つのカテゴリに分類。特定カテゴリの購入金額が、そのユーザーの購入金額合計の50%以上を占める場合は「ファッションユーザー」「雑貨ユーザー」「グルメユーザー」のいずれか、どのカテゴリも50%より少ない場合は「バランスユーザー」の文字列を格納。

上記のフィールドを持つ結果テーブルを得るSQL文は、次ページの［6-2-10］となります。12〜23行目にあるインデント部分が、注文履歴テーブルを1

ユーザー1レコードに変換しつつ集計しているサブクエリです。初回購入日は13行目、購入回数は14行目、カテゴリごとの購入金額合計は16〜21行目で求められています。

あとは外側にあるメインのクエリで、ユーザーをカテゴライズするCASE文（2〜10行目）を記述すれば完成です。結果テーブルを見ると、ユーザーごとの分析に役立つ情報が得られていることが分かると思います。

なお、この例ではサブクエリを使わなくても同じ結果を得られますが、クエリ全体がスッキリするので、あえて使っています。

6-2-10 1ユーザー1レコードにしてカテゴライズする

```
1  SELECT *
2  , CASE
3  WHEN (fashion_revenue / sum_revenue) >= 0.5
4  THEN "ファッションユーザー"
5  WHEN (zakka_revenue / sum_revenue) >= 0.5
6  THEN "雑貨ユーザー"
7  WHEN (gourmet_revenue / sum_revenue) >= 0.5
8  THEN "グルメユーザー"
9  ELSE "バランスユーザー"
10 END AS user_cat
11 FROM (
12   SELECT user_id
13   , MIN(order_date) AS first_purchase_date
14   , COUNT(DISTINCT order_id) AS orders
15   , SUM(revenue) AS sum_revenue
16   , SUM(IF (item_cat = "ファッション", revenue, 0))
17   AS fashion_revenue
18   , SUM(IF (item_cat = "雑貨", revenue, 0))
19   AS zakka_revenue
```

```
    , SUM(IF (item_cat = "グルメ", revenue, 0))
    AS gourmet_revenue
    FROM sample.small_order_date
    GROUP BY user_id
  )
```

結果テーブル

行	user_id	first_purchase_date	orders	sum_revenue	fashion_revenue	zakka_revenue	gourmet_revenue	user_cat
1	ABC	2020-03-05	3	31700	22000	3900	5800	ファッションユーザー
2	STU	2020-05-13	2	19100	3900	6600	8600	バランスユーザー
3	XYZ	2020-06-01	2	47800	9800	38000	0	雑貨ユーザー

1ユーザー1レコードに変換し、ユーザーの特性に応じたカテゴライズができた

特定商品の有無別に平均注文金額を比較する

実践例の2つ目は、ECサイトのように1回の注文（ユニークな［order_id］）で複数の商品（item）が注文できるシステムから出力した、以下の［6-2-11］のような販売実績テーブルを対象にします。このテーブルの名前は［small_item］とします。

6-2-11 販売実績テーブル

order_id	item	revenue
123	A	5800
123	B	5800
124	A	7200
124	D	7200
124	E	7200
125	C	4800
125	E	4800
126	B	3200
126	C	3200

このテーブルを見て、『商品「A」が含まれている注文は、「A」を含まない注文よりも販売金額合計が高いのではないか？』という仮説が浮かんだとします。その仮説をSQLで検証してみましょう。

具体的には、①商品「A」を含む注文の販売金額合計、②商品「A」を含まない注文の販売金額合計、③それぞれの販売金額合計の平均値を求め、比較します。①が大きければ仮説は正しいとします。

記述するSQL文の考え方としては、まず、注文ごとの販売金額の合計を取得するとともに、それぞれの注文に対して『商品「A」を含むか、含まないか』というフラグを立てます。次に、そのフラグをグループ化の対象として、販売金額合計の平均値を取得する、という流れになります。

ただ、注文ごとにグループ化した後では、商品「A」を購入したというフラグを立てられません。そこで、元のテーブルに［item_a_flag］というフィールドを追加し、レコードごとにフラグを立てます。そのSQL文が以下の［6-2-12］です。IF文を使い、［item］が「A」ならば「1」、「A」でなければ「0」というフラグを立てています。

6-2-12 特定商品を含む注文にフラグを立てる

```
1  SELECT *
2  , IF (item = "A", 1, 0) AS item_a_flag
3  FROM sample.small_item
4  ORDER BY 1, 2
```

結果テーブル

行	order_id	item	revenue	item_a_flag
1	123	A	5800	1
2	123	B	3200	0
3	124	A	5800	1
4	124	C	4800	0

商品「A」を含む注文にフラグ「1」が格納された

結果、[order_id] が「123」と「124」の注文に商品「A」が含まれていたことが分かりました。そして、これをサブクエリとして [order_id] でのグループ化を行うSQL文は、以下の [6-2-13] となります。

[item_a_flag] フィールドが、商品「A」を含んだ注文かどうかのフラグになっています。ポイントはSQL文の2行目と3行目です。

2行目では、SUM関数で各注文の販売金額合計を集計しています。結果テーブルを見ると [revenue] が [item] 単位ではなく [order_id] 単位になっていることが分かります。3行目では、レコード単位で付与されていた商品「A」を含んだ注文かどうかのフラグをMAX関数で集計しています。結果、フラグが [order_id] 単位にまとまりました。

商品「A」を含むレコードに「true」、商品「A」を含まないレコードに「false」のフラグを立てて、ブール型のカラムとし、MAX関数を利用することもできます。その場合の「MAX(item_a_flag)」は、[order_id] でグループ化したときに1つでも「true」があれば「true」、1つも「true」がなければ「false」となります。

6-2-13 特定商品を含むフラグを注文ごとに集計する

```
SELECT order_id
, SUM(revenue) AS sum_rev
, MAX(item_a_flag) AS item_a_flag
FROM
(
  SELECT *
  , IF (item = "A", 1, 0) AS item_a_flag
  FROM sample.small_item
)
GROUP BY order_id
ORDER BY 1
```

結果テーブル

行	order_id	sum_rev	item_a_flag
1	123	9000	1
2	124	12400	1
3	125	6600	0
4	126	10400	0

注文ごとに販売金額合計と商品「A」を含むフラグがまとまった

最後に、[item_a_flag] フィールドを利用してグループ化を行い、[sum_rev] の平均値を取得するメインのクエリを外側に記述します。以下のSQL文 [6-2-14] を実行すると、「Aを含む注文」「Aを含まない注文」の2レコードに対して、販売金額合計の平均値が [avg_revenue_per_order_id] フィールドに格納されます。

結果、商品「A」が含まれている注文は、含まれていない注文よりも販売金額が高く、仮説は正しかったと検証できました。

6-2-14 特定商品の有無別に平均注文金額を取得する

```
SELECT
IF (item_a_flag = 1, "Aを含む注文", "Aを含まない注文")
AS order_cat
, AVG(sum_rev) AS avg_revenue_per_order_id
FROM
(
  SELECT order_id
  , SUM(revenue) AS sum_rev
  , MAX(item_a_flag) AS item_a_flag
  FROM
  (
    SELECT *
    , IF (item = "A", 1, 0) AS item_a_flag
    FROM sample.small_item
```

```
    ORDER BY 1
  )
  GROUP BY order_id
)
GROUP BY order_cat
```

結果テーブル

行	order_cat	avg_revenue_per_order_id
1	Aを含む注文	10700.0
2	Aを含まない注文	8500.0

商品「A」を含む注文と含まない注文で平均注文金額を比較できた

作業記録とカレンダーから土日以外の休業日を調べる

実践例の3つ目は、カレンダーのテーブルを利用したテクニックです。以下の［6-2-15］の左側にある作業記録テーブルは、5部屋ある集合住宅で内装作業を行う職人が、どの部屋でいつから作業を始め（start_date）、いつまでに終わらせたのか（end_date）を示しています。

右側にあるのは、職人が作業した月のカレンダーのテーブルです。これら2つのテーブルから「職人が6月中に土日以外で休んだ（作業がなかった）日はあるか？」「あるとすれば、それは6月何日か？」をSQLで調べていきます。テーブル名は作業記録テーブルが［small_workdate］、カレンダーテーブルが［small_cal］としましょう。

6-2-15 作業記録テーブルとカレンダーテーブル

作業記録テーブル

room	start_date	end_date
101	2020-06-01	2020-06-05
102	2020-06-08	2020-06-11
103	2020-06-12	2020-06-16
104	2020-06-19	2020-06-23
105	2020-06-24	2020-06-30

カレンダーテーブル

date	day_of_week
2020-06-01	Mon
2020-06-02	Tue
2020-06-03	Wed
⋮	⋮
2020-06-28	Sun
2020-06-29	Mon
2020-06-30	Tue

まず、いったん2つのテーブルを横につなぎます。以下の［6-2-16］は、カレンダーを左側、作業記録を右側としてCROSS JOINするSQL文です。総当たりで結合するため、作業記録の5レコード、カレンダーの30レコードを掛け合わせた150レコードの結果テーブルが返されます。カレンダー側の日付である［date］と［room］の昇順でソートしました。

6-2-16 作業記録・カレンダーテーブルを交差結合する

```
1  SELECT * FROM sample.small_cal
2  CROSS JOIN sample.small_workdate
3  ORDER BY date, room
```

結果テーブル

行	date	day_of_week	room	start_date	end_date
1	2020-06-01	Mon	101	2020-06-01	2020-06-05
2	2020-06-01	Mon	102	2020-06-08	2020-06-11
3	2020-06-01	Mon	103	2020-06-12	2020-06-16
4	2020-06-01	Mon	104	2020-06-19	2020-06-23
5	2020-06-01	Mon	105	2020-06-24	2020-06-30

ページあたりの行数: 100 1 - 100 / 150 最初のペー

作業記録とカレンダーを総当たりで結合した

結果を見ると、1行目は［date］(日付）が6月1日、[start_date]（作業開始日）が6月1日、[end_date]（作業終了日）が6月5日となっています。日付が作業開始日・終了日の間にあるため、職人が作業していたことを表すレコードです。一方、2行目は日付が6月1日、作業開始日が6月24日、作業終了日が6月30日で、作業開始日・終了日の間にない日付なので、職人が作業していなかったことを表しています。

つまり、[date］の日付が［start_date］と［end_date］の間なら作業日、そうでなければ休業日となります。このルールで各レコードにフラグを立てていきましょう。

具体的には、以下の［6-1-17］のように記述します。WITH句を利用して「calendar」という仮想テーブルを作り、そのテーブルに［working_flag］というフラグを格納するフィールドを作成します。

フラグを立てる条件は「［date］が［start_date］と［end_date］の間にある」です。もうWITH句の結果を確認する必要はないので、［6-2-16］の3行目にあったORDER BY句は削除しています。

6-2-17 作業記録に含まれる日付にフラグを立てる

```
1  WITH calendar AS (
2  SELECT * FROM sample.small_cal
3  CROSS JOIN sample.small_workdate
4  )
5
6  SELECT *
7  , IF (date BETWEEN start_date AND end_date, 1, 0)
8  AS working_flag FROM calendar
9  ORDER BY 1
```

結果テーブル

行	date	day_of_week	room	start_date	end_date	working_flag
1	2020-06-01	Mon	101	2020-06-01	2020-06-05	1
2	2020-06-01	Mon	103	2020-06-12	2020-06-16	0
3	2020-06-01	Mon	102	2020-06-08	2020-06-11	0
4	2020-06-01	Mon	104	2020-06-19	2020-06-23	0
5	2020-06-01	Mon	105	2020-06-24	2020-06-30	0
6	2020-06-02	Tue	101	2020-06-01	2020-06-05	1
7	2020-06-02	Tue	103	2020-06-12	2020-06-16	0
8	2020-06-02	Tue	102	2020-06-08	2020-06-11	0
9	2020-06-02	Tue	104	2020-06-19	2020-06-23	0
10	2020-06-02	Tue	105	2020-06-24	2020-06-30	0

作業日を表すフラグが格納された

結果テーブルのうち、前ページの画面で見える範囲では、6月1日の月曜日と6月2日の火曜日は作業していたことが分かりました。

次に、日付でグループ化を行い、作業日のフラグを「MAX(working_flag)」で取り出します。同時に「MAX(day_of_week)」で唯一の値（曜日）を取得します。SQL文は以下の［6-2-18］となり、6月の日付でグループ化を行うので、結果テーブルは30レコードになります。

6-2-18 作業日を表すフラグを日付ごとに集計する

```
WITH calendar AS (
SELECT * FROM sample.small_cal
CROSS JOIN sample.small_workdate
)

SELECT date
, MAX(day_of_week) AS day_of_week
, MAX(working_flag) AS working_flag
FROM (
SELECT *
, IF (date BETWEEN start_date AND end_date, 1, 0)
AS working_flag
FROM calendar
)
GROUP BY date
ORDER BY date
```

結果テーブル

行	date	day_of_week	working_flag
1	2020-06-01	Mon	1
2	2020-06-02	Tue	1
3	2020-06-03	Wed	1
4	2020-06-04	Thu	1
5	2020-06-05	Fri	1
6	2020-06-06	Sat	0
7	2020-06-07	Sun	0
8	2020-06-08	Mon	1
9	2020-06-09	Tue	1
10	2020-06-10	Wed	1

ページあたりの行数: 100 1 - 30 / 30

日付ごとに作業日を表すフラグがまとまった

日付・曜日の単位で、[working_flag] フィールドに作業日は「1」、休業日は「0」のフラグが格納されています。この結果テーブルに対して『土日でない、かつ [working_flag] が「0」である』という条件で絞り込めば、土日以外の休業日を取得できます。

SQL文は次ページの [6-2-19] の通りで、追加したのは15〜16行目のみです。グループ化したフィールドの絞り込みにWHERE句は使えないため、HAVING句を利用しています。結果、土日以外では6月17日の水曜日と6月18日の木曜日が休業日だったことが分かり、勤怠管理などの実務に役立てられるかと思います。

この分析は、作業記録テーブルだけでは実現できません。そのため、自分でカレンダーテーブルを用意して、CROSS JOINとサブクエリを利用して分析した、というところがポイントになっています。

6-2-19 土日以外の休業日を取得する

```
WITH calendar AS (
SELECT * FROM sample.small_cal
CROSS JOIN sample.small_workdate
)

SELECT date
, MAX(day_of_week) AS day_of_week
, MAX(working_flag) AS working_flag
FROM (
SELECT *
, IF (date BETWEEN start_date AND end_date, 1, 0)
AS working_flag
FROM calendar
)
GROUP BY date
HAVING day_of_week
NOT IN ("Sat", "Sun") AND working_flag = 0
ORDER BY date
```

結果テーブル

行	date	day_of_week	working_flag
1	2020-06-17	Wed	0
2	2020-06-18	Thu	0

土日以外の休業日が分かった

STEP UP

結果テーブルをグラフで表現する

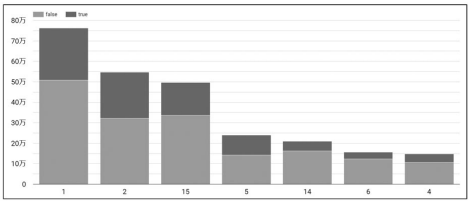
筆者は、記述したSQL文から思い通りの結果テーブルが返ってきた瞬間に、大きな喜びを感じます。また、その結果テーブルによって自分が立てた仮説が証明された瞬間にも、大きな感動があるものです。

しかし、時に物足りなく感じるのが、SQL文の実行結果はあくまでも「1つの値」、もしくは「テーブル」(表) だということです。ビジュアル面でのアピール力は、決して高くありません。

そこで、Googleが提供する無料のBIツール「Googleデータポータル」で、結果テーブルをビジュアルに表現してみることをおすすめします。データポータルとの連携手順についてはSECTION 2-6で解説したので、ここでは具体的なビジュアライズの例を紹介しましょう。

以下の［6-2-20］は、演習用の［sales］テーブルと［customers］テーブルを利用し、販売金額の上位7位までの商品についてプレミアム会員（is_premium = true）と一般会員（is_premium = false）の販売金額を取得したテーブルを元に、データポータルの積み上げ縦棒グラフで視覚化したものです。

6-2-20 プレミアム会員と一般会員の販売金額のグラフ



また、以下の［6-2-21］は、グラフの元になった結果テーブルを取得するためのSQL文です。みなさんの環境でも実行し、その結果テーブルとグラフを見比べてみてください。グラフとして視覚化することで、情報の受け手が「何が起きているのか？」を理解するのに要する時間を短くでき、認知的負荷を少なくできることが実感できるのではないかと思います。

6-2-21 商品別の販売金額を会員種別ごとに取得する

```
SELECT product_id
, is_premium
, SUM(revenue) AS sum_revenue
FROM (
SELECT * FROM sample.sales
LEFT JOIN sample.customers
USING (user_id)
)
GROUP BY product_id, is_premium
HAVING product_id IN (
SELECT product_id
FROM sample.sales
GROUP BY product_id
ORDER BY SUM(revenue) DESC
LIMIT 7
)
```

SECTION

6-3 UNNEST関数による GA4テーブルの変換

サブクエリについて、ひと通り学びましたね。本章の最後として、やや発展的なSQLの知識を紹介します。GA4からエクスポートしたテーブルの扱いです。

GA4はまだ僕も使ったことがありません。でも、今後勉強しないと…と思っていたところでした。

GA4はBigQueryと簡単に連携できるため、SQLを駆使した分析が可能です。ただ、データの構造が特殊なので、ここでベースとなる知識を身につけてください。

GA4のデータはすべて「イベント」でできている

2020年10月、それまで「App＋ウェブプロパティ」という名称でベータ版として提供されていたGoogleアナリティクスのバージョンが、「Googleアナリティクス4プロパティ」（GA4）として正式版となりました。その概要はSECTION 1-1でも述べた通りです。

GA4では、無料版でありながら、生データをBigQueryにエクスポートできるようになりました。これは、本書で学んできたSQLを使って、**GAが蓄積する生データから自在にデータを抽出・整形できるようになった**ことを意味しています。SQLの利用シーンがさらに広がり、スキルとしての重要度が高まったともいえるでしょう。

GA4からBigQueryにエクスポートされるテーブルから値を取り出すにあたっては、サブクエリに加えて「**UNNEST関数**（アンネスト）」の知識が必要となります。それらについて、本節で学んでいきましょう。

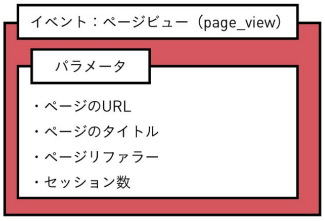
まず、GA4からエクスポートされるデータの構造について理解してください。**GA4では、収集するデータをすべて「イベント」として扱います**。したがって、これまで慣れ親しんでいたページビューも、数種類あるイベントの1つとしての「ページビュー」(page_view) という扱いになります。

ほかにも「初回訪問」(first_visit) や「セッションの開始」(session_start)、「ページにおける90%の高さまでのスクロール」(scroll) といった多様なイベントを自動で取得できますが、ここでは「ページビュー」に絞って解説します。

また、イベントには属性があります。例えば、ページビューイベントには「ページのURL」や「ページのタイトル」のほか、「ページリファラー」(そのページの直前のページ)、「セッション数」(そのページを閲覧したセッションが、ユーザーにとって何回目のセッションだったかを表す整数) などが存在します。これらの属性は「パラメータ」と呼びます。

こうしたGA4のデータ構造を単純化して表すと、以下の図 [6-3-1] のようになります。

6-3-1 GA4のデータ構造



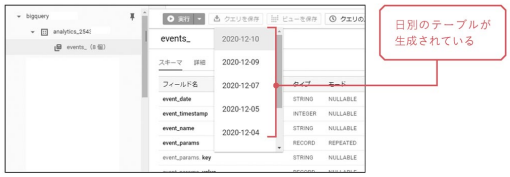
1レコード内に仮想的なテーブルが存在する

では、実際にGA4からBigQueryにエクスポートしたテーブルを見てみましょう。以下の画面［6-3-2］は筆者のテスト環境です。

この画面では「bigquery」というプロジェクト名が見えますが、エクスポート先となるプロジェクトはユーザーが任意に選択できます。データセット名は「analytics_数字」となり、数字には自動的に採番された9桁が入ります。どのような数字であるかを気にする必要はありません。

テーブル名は、ナビゲーションパネルの一覧では「event_」と表示されますが、実際には「event_20201201」「event_20201202」など、「event_」に続いてYYYYMMDD形式で日付が入った名前のテーブルが格納されています。そのようなテーブルが1日1個ずつ生成されていく形です。［6-3-2］では12月10、9日…と日別のテーブルがデータセットに存在することが見てとれます。

6-3-2 GA4からエクスポートしたテーブル



テーブルのスキーマを確認すると、[event_params] カラムの「タイプ」として、見慣れない「RECORD」があるのが分かります。タイプはデータ型なので、「**RECORD型**」というデータ型であることを意味しています。GA4からエクスポートしたテーブルは、このRECORD型のカラムが存在する、というのが1つの特徴になっています。

RECORD型のカラムを詳細に見ていきましょう。次ページの［6-3-3］には、多数ある、すべてのカラムから［event_timestamp］［event_name］に加え、

「event_params」で始まるカラムだけを掲載しています。

［event_params］は、その名前で始まる［event_params.key］［event_params.value.string_value］［event_params.value.int_value］という3カラムに分かれています。1つのカラムには1つのデータ型しか格納できないため、［event_params.value.string_value］には文字列が、［event_params.value.int_value］には整数が格納されています。

なお、表示の関係で見えていませんが、実際にはFLOAT64型データを格納する［event_params.value.float_value］と［event_params.value.double_value］というカラムも存在します。ただし、本書執筆時点では値が格納されていないため、［6-3-3］で見えている範囲のカラムを例にとって説明します。

6-3-3 RECORD型のカラムと格納された値

行	event_timestamp	event_name	event_params.key	event_params.value.string_value	event_params.value.int_value
1	1607566126790760	page_view	engaged_session_event	null	1
			source	999.oops.jp	null
			medium	referral	null
			ga_session_id	null	1607565828
			page_location	http://999.oops.jp/chikuhaku2	null
			campaign	(referral)	null
			session_engaged	1	null
			page_referrer	http://999.oops.jp/chikuhaku1.html	null
			page_title	竹柏記2	null
			ga_session_number	null	1

イベント「page_view」に複数のパラメータが存在している

［6-3-3］には10レコードあるように見えますが、［event_timestamp］と［event_name］の2行目以降はグレーで塗られていることから、これが1レコードを表しているのだと気づきます。

つまり、**RECORD型とは「1レコード内に別の仮想的なテーブルが埋め込まれているデータ型」**だと理解できます。そして、1レコードが1つのイベントを、その中に埋め込まれたテーブルがそのイベントの属性であるパラメータを格納しています。図で表すと次ページの［6-3-4］となります。

6-3-4 イベントとパラメータの関係

イベント

event_timestamp	event_name	このレコードに含まれる［event_params］テーブル
1607566051046924	page_view	

パラメータ

event_params.key	event_params.value.string_value	event_params.value.int_value
source	999.oops.jp	null
medium	referral	null
ga_session_id	null	160756828
page_location	http://999.oops.jp/chikuhaku2	null

UNNEST関数でテーブルの入れ子をほどく

こうした構造のテーブルから、SQL文が成立する形でデータを取得するために必要となるのが、サブクエリとUNNEST関数です。「NEST」（ネスト）は「入れ子」という意味ですので、「UNNEST」は「入れ子をほどく」といった意味になります。

サブクエリとUNNEST関数の利用例として、次ページの［6-3-5］を紹介します。結果テーブルから分かる通り、このSQL文で［6-3-3］から「RECORD型を利用しない通常のテーブル」を取得できます。なお、RECORD型を利用しない通常のテーブルのことを「フラットテーブル」と呼びます。

6-3-5 ページビューイベントからページのURLを取得する

```
1  WITH master AS
2  (
3  SELECT event_timestamp, event_name, event_params
4  FROM {データセット名}.events_20201210
5  WHERE event_name = "page_view"
6  AND event_timestamp = 1607566126790760
```

```
7   )
8
9   SELECT event_timestamp, event_name,
10  (
11  SELECT value.string_value FROM UNNEST(event_params)
12  WHERE key = "page_location"
13  )
14  AS page_location
15  FROM master
```

結果テーブル

行	event_timestamp	event_name	page_location
1	1607566126790760	page_view	http://999.oops.jp/chikuhaku2

[page_location]を通常のテーブルで取得できた

このSQL文では、[6-3-3]のテーブルがWITH句によって「master」という名前で生成されています。結果テーブルを取得しているのは、9行目以降にあるメインのクエリです。

11行目にあるSELECT句が1つの値「value.string_value」を戻すように記述され、14行目のAS句によって［page_location］という名前で取得されていることが分かります。また、11行目のFROM句でUNNEST関数の引数として「event_params」を呼び出すことで、[event_params］テーブルを通常のテーブルのように扱えるようにしています。

12行目のWHERE句では「key」が「page_location」に一致するという条件が指定されているため、「value.string_value」から「page_location」に一致する値として「http://999.oops.jp/chikuhaku2」が戻されることが分かるのではないでしょうか。

このテクニックが理解できれば、11行目以降のクエリを次ページの[6-3-6]のように書き換えれば、[page_location］に加えて［page_title］も取得できます。

6-3-6 ページビューイベントからURLとタイトルを取得する

```
WITH master AS
(
SELECT event_timestamp, event_name, event_params
FROM {データセット名}.events_20201210
WHERE event_name = "page_view"
AND event_timestamp = 1607566126790760
)

SELECT event_timestamp, event_name,
(
SELECT value.string_value FROM UNNEST(event_params)
WHERE key = "page_location"
)
AS page_location,
(
SELECT value.string_value FROM UNNEST(event_params)
WHERE key = "page_title")
AS page_title
FROM master
```

結果テーブル

行	event_timestamp	event_name	page_location	page_title
1	1607566126790760	page_view	http://999.oops.jp/chikuhaku2	竹柏記2

[page_location] [page_title] をフラットテーブルとして取得できた

日別の複数のテーブルはまとめて指定できる

さらに実践的な利用例も見てみましょう。[6-3-5] では4行目のFROM句で対象のテーブルを12月10日分だけとしているところを、例えば11月28日から12月10日の複数テーブルに広げます。

加えて、WHERE句のAND演算子（6行目）で1レコードだけに絞り込んでいるのをやめ、ページタイトル別のページビューを求めるクエリが以下の [6-3-7] となります。

6-3-7 ページタイトル別のページビューを取得する

```
1  SELECT page_title, COUNT(*) AS page_views
2  FROM (
3  SELECT
4  (
5  SELECT value.string_value FROM UNNEST(event_params)
6  WHERE key = "page_title") AS page_title
7  FROM {データセット名}.events_*
8  WHERE _TABLE_SUFFIX BETWEEN "20201128" AND "20201210"
9  AND event_name = "page_view"
10 )
11 GROUP BY page_title
```

結果テーブル

行	page_title	page_views
1	Kaz Kida テストサイトトップページ	67
2	竹柏記1	18
3	竹柏記2	18

[page_title] 別のページビュー数をフラットテーブルとして取得できた

前ページのSQL文で注目してほしいのが、7〜8行目にあるFROM句とWHERE句です。前述の通り、GA4からエクスポートされるBigQuery上のテーブルは1日1個、自動的に生成されるため、複数の日付を対象にデータを取得したい場合は、複数テーブルに対して実行する必要があります。

BigQueryでは、以下の［6-3-8］で示した書式で記述することで、**複数の日のテーブルをあたかも1つのテーブルのように扱うことが可能**です。日付の表現はいずれもYYYYMMDD形式の8桁となります。

FROM句の末尾にある「*」はワイルドカードと呼ばれる変動部分で、その変動部分にどのような値を当てはめるかを、次行のWHERE句で指定しています。「_TABLE_SUFFIX」は「テーブルの接尾辞」という意味の変数で、文字通りテーブルの末尾にくる値を指定するために利用します。

よって、［6-3-8］の全体としては「テーブル名の末尾が開始日から終了日に該当するテーブル」を指定することになります。GA4からエクスポートされたBigQuery上のテーブルを対象にSQLを記述するときには、ほぼ毎回必要になるテクニックなので、ぜひ覚えておいてください。

6-3-8 複数のGA4テーブルを対象にする書式

```
1  FROM {データセット名}.テーブル名の不変部分*
2  WHERE _TABLE_SUFFIX BETWEEN "開始日" AND "終了日"
```

SECTION

6-4 確認ドリル

問題 016

［sales］テーブルには、15種類の商品ID（product_id）が販売された記録が格納されています。商品ID（product_id）別に平均単価を計算し、これを①とします。さらに、15種類の①について平均したものを②とし、［avg_unit_price_by_product］とします。

そのうえで、［sales］テーブルにおける商品ID別の平均単価（avg_unit_price ⇒①）が②より大きい商品IDに絞り込み、以下の5カラムの結果テーブルを取得してください。並べ替えは［avg_unit_price］の大きい順とし、最低1か所でサブクエリを利用してください。

1. 商品ID（product_id）
2. 販売個数合計（sum_quantity）
3. 販売金額合計（sum_revenue）
4. 商品ID別の単価（avg_unit_price） ⇒①
5. 商品ID別の平均単価の平均（avg_unit_price_by_product） ⇒②

問題 017

［customers］テーブルから、4人以上のプレミアム顧客（is_premium = true）がいる都道府県における、もっとも高齢の顧客の情報を全カラム取り出してください。結果テーブルは、7カラム1レコードになります。

問題 018

［web_log］テーブルから、「セッションの総数」（session）と「ページビュー数の合計」（sum_pageviews）を取得してください。セッションは、ユニークな［cid］と［session_count］の組み合わせと定義します。ページビューは、1レコードが1ページビューを表しています。結果テーブルは、2フィールド1レコードになります。

問題 019

［sales］テーブルには［is_proper］フィールドがあり、「true」は定価での販売、「false」は割引価格での販売であることを示しています。定価での販売金額合計から、割引価格での販売金額合計を差し引いた値を求めてください。

問題 020

［sales］テーブルから、商品ID（product_id）が「1」の商品について割引販売（is_proper = false）を行わなかった場合、販売金額（revenue）がいくら増加するかを求めてください。結果は、1カラム1レコードの「1つの値」になります。

解答

解答 016

```sql
WITH master AS
(SELECT product_id
, SUM(quantity) AS sum_qty
, SUM(revenue) AS sum_revenue
, SUM(revenue) / SUM(quantity) AS avg_unit_price
FROM sample.sales
GROUP BY product_id
)

SELECT *
, (SELECT AVG(avg_unit_price) FROM master)
AS avg_unit_price_by_product
FROM master
WHERE avg_unit_price >
(SELECT AVG(avg_unit_price) FROM master)
ORDER BY 4 DESC
```

結果テーブル

行	product_id	sum_qty	sum_revenue	avg_unit_price	avg_unit_price_by_product
1	6	44	156600	3559.090909090909	1913.003820243751
2	15	169	497400	2943.195266272189	1913.003820243751
3	5	87	240240	2761.3793103448274	1913.003820243751
4	10	37	89750	2425.675675675676	1913.003820243751
5	4	63	148080	2350.4761904761904	1913.003820243751
6	2	252	547360	2172.063492063492	1913.003820243751
7	1	388	762800	1965.979381443299	1913.003820243751

［sales］テーブルを［product_id］でグループ化し、販売個数合計、販売金額合計、それらを割って求めた平均単価のテーブルを作ったものを、WITH句で「master」という名前の仮想テーブルとして利用しています。［master］テーブルの作成時に、商品ID別のグループ化と集計を行っているため、メインのSQLでの集計は不要です。商品ID別の平均単価の平均について、1つの値を戻り値とするサブクエリで取り出して、5番目のフィールドに加えると同時に、WHERE句の絞り込みに利用しています。

解答 017

```sql
WITH master AS
(SELECT *
FROM sample.customers
WHERE prefecture IN
(SELECT prefecture
FROM sample.customers
GROUP BY prefecture
HAVING SUM(IF (is_premium IS TRUE, 1, 0)) >= 4
))

SELECT * FROM master
WHERE birthday = (SELECT MIN(birthday) FROM master)
```

結果テーブル

行	user_id	name	birthday	gender	prefecture	register_date	is_premium
1	19935	大須賀 珠英	1964-08-20	2	東京	2017-05-10	true

WITH句では［customers］テーブルから「プレミアム顧客が4名以上いる都道府県」に合致するレコードを取得しています。WHERE句にて、IN演算子で比較する対象にサブクエリを記述し、そのサブクエリでは、条件に該当する都道府県のリストを返しています。表形式の戻り値を「master」として仮想テーブルで利用します。

「もっとも高齢の顧客を取得する」ということは、誕生日（birthday）が［master］テーブル全体の最小値と一致する、ということなので、「WHERE birthday = (SELECT MIN(birthday) FROM master)」で個別の顧客の誕生日（birthday）と［master］テーブル全体の最小の［birthday］を比較して、同じ顧客だけに絞り込んでいます。

なお、メインのクエリは以下でもよいでしょう。

```
SELECT * FROM master
WHERE birthday IS NOT NULL
ORDER BY birthday
LIMIT 1
```

解答 018

```
SELECT COUNT(*) AS session
, SUM(pageviews) AS sum_pageviews
FROM (
SELECT cid, session_count, COUNT(*) AS pageviews
FROM sample.web_log
GROUP BY cid, session_count
)
```

結果テーブル

行	session	sum_pageviews
1	987	2604

「手ごわいクエリを記述する必要があるのでは？」と心配した人もいるかもしれませんが、実際は上記の通り、数行で取得できます。

内側のクエリから見ていきましょう。セッション数は「ユニークな［cid］と［session_count］の組み合わせ」と定義されたので、それらでグループ化

します。SECTION 4-1で学んだ通り、グループ化により、フィールドの構成要素は1つが1レコードになるからです。ページビュー数はレコード数と定義されているので、「COUNT(*)」で取得できます。

外側のクエリは、セッション数を求めるためにレコード数をカウントし、ページビュー数の合計を求めるためにSUM関数で［pageviews］を合算しています。

解答 019

```
SELECT sum_proper_revenue - sum_non_proper_revenue
AS revenue_diff
FROM (SELECT
MAX((SELECT SUM(revenue) FROM sample.sales
WHERE is_proper IS TRUE))
AS sum_proper_revenue
, MAX((SELECT SUM(revenue) FROM sample.sales
WHERE is_proper IS FALSE))
AS sum_non_proper_revenue
FROM sample.sales
)
```

結果テーブル

行	revenue_diff
1	2320380

［sales］テーブルから1つだけの値を返すサブクエリとして、まずは定価で販売された場合の販売金額の合計を求めます。この問題のポイントは、4〜5行目にある以下のサブクエリです。

```
SELECT SUM(revenue) FROM sample.sales
WHERE is_proper IS TRUE
```

これが記述できれば、WHERE句の条件を変えて「割引価格で販売された場合の販売金額の合計」も求めることができます。

しかし、ここまで考えついたうえで、以下のようなSQLを記述し、レコード数の分だけ数値が並んでしまった人もいるでしょう。

```
SELECT
(SELECT SUM(revenue) FROM sample.sales
WHERE is_proper IS TRUE) AS sum_proper_revenue
, (SELECT SUM(revenue) FROM sample.sales
WHERE is_proper IS FALSE) AS sum_non_proper_revenue
FROM sample.sales
```

SUM関数での集計は問題ないので、WHERE句で絞り込んだレコードの合計値については、2カラムとも正しく取得できます。しかし、グループ化されていないため、元のテーブルのレコード数分、値が返ってきてしまいます。ただ、この問題では「○○別の」というグループ化のフィールドを含められないので、グループ化が利用できません。そこで、MAX関数でサブクエリからの戻り値を1つにまとめます。

以下が別解となります。6〜12行目では1レコードに集約せず、外側のクエリ（1行目）の記述で、MAX関数を利用して値を1つにまとめて引き算します。

```
1  SELECT
2  MAX(sum_proper_revenue) - MAX(sum_non_proper_revenue)
3  AS revenue_diff
4  FROM (
5  SELECT
6  (SELECT SUM(revenue) FROM sample.sales
7  WHERE is_proper IS TRUE)
8  AS sum_proper_revenue
9  , (SELECT SUM(revenue) FROM sample.sales
10 WHERE is_proper IS FALSE)
11 AS sum_non_proper_revenue
12 FROM sample.sales
13 )
```

解答 020

```
1  SELECT SUM(lost_revenue) AS sum_lost_revenue
2  FROM (
3  SELECT *
4  , quantity * (proper_unit_price - discount_unit_price)
5  AS lost_revenue
6  FROM (
7  SELECT *, revenue / quantity AS discount_unit_price
8  , (
9  SELECT
10 SUM(revenue) / SUM(quantity)
11 FROM sample.sales
12 WHERE product_id = 1 AND is_proper IS TRUE
13 ) AS proper_unit_price
14 FROM sample.sales
15 WHERE product_id = 1 AND is_proper IS FALSE
16 )
17 )
```

結果テーブル

行	sum_lost_revenue
1	13200.0

[product_id]が「1」の商品について、数量(quantity)、定価の単価(proper_unit_price)、割引価格の単価（discount_unit_price）を1つのテーブルにまとめて、「数量×（定価の単価ー割引価格の単価)」を計算した結果を合計するのが、基本的な考え方です。

9〜13行目のクエリでは「product_id = 1 AND is_proper IS TRUE」という条件で、商品IDが「1」の定価での販売にレコードを絞り込んでいます。そのうえで［revenue］の合計額と［quantity］の合計数を割り算して求めた「1つの値」を［proper_unit_price］として取得しています。この値が商品ID

(product_id) が「1」の定価の単価です。

次に、その外側のクエリ（7〜8行目、14〜15行目）では、「product_id = 1 AND is_proper IS FALSE」で、割引販売された商品IDが「1」のレコードに絞り込んだうえで、もともと存在していたすべてのフィールドを「*」で取得し、かつ［revenue］を［quantity］で割った割引販売の単価（discount_unit_price）のフィールドと、9〜13行目のクエリで取得した［proper_unit_price］を追加しています。

結果的に、7〜15行目では、もともとの全フィールド、割引販売の単価（discount_unit_price）、定価での単価（proper_unit_price）が1つのテーブルにそろったことになります。あとは「数量×（定価の単価ー割引価格の単価）」で、定価で販売していたら得られた売上を計算しています。

STEP UP

BigQueryの最新仕様を確認する

BigQueryは一般のアプリケーションと同様に、仕様の追加や変更があります。それらの情報はリリースノートとして、以下のページに集約されて発表されています。本書では2020年10月の仕様変更までを反映していますが、その後の仕様変更について知りたい場合は、リリースノートのページを参照してください。

Release notes | BigQuery
https://cloud.google.com/bigquery/docs/release-notes?hl=ja

CHAPTER 7

さまざまな関数

7-1　数値を扱う関数
7-2　文字列を扱う関数
7-3　正規表現を利用する関数
7-4　日付・時刻を扱う関数
7-5　統計集計関数
7-6　確認ドリル

これまでの章で、「ビジネスデータの分析」のために必要なSQLの命令と基本的な関数をいくつか学びました。本章と次章のCHAPTER 8では、さらに多くの関数について学びます。まずは本章で、数値の切り捨てや切り上げ、文字列の連結や抽出、日付の丸めや2つの日付の間隔の取得などを行う関数を見ていきましょう。

SECTION

7-1 数値を扱う関数

SQLで使える、さまざまな関数の学習を始めましょう。

関数はExcelなどでも使うので、なじみがありますね。

すべてを暗記する必要はありませんが、「どのような関数が存在するのか？」を知っていることは重要です。

数値を丸めるFLOOR/CEIL/ROUND関数

まずは「数値の丸め」にまつわる関数から見ていきましょう。Excelでもよく利用するので、特になじみ深い関数群だと思います。以下の表［7-1-1］にまとめたように、切り捨てには**FLOOR関数**（フロア）、切り上げには**CEIL関数**（シール）、四捨五入には**ROUND関数**（ラウンド）を使います。

7-1-1 数値の丸めに利用する関数

構文	処理
FLOOR(フィールド名)	該当フィールド内の値よりも小さい最大の整数値を返す
CEIL(フィールド名) CEILING(フィールド名)	該当フィールド内の値以上で最小の整数値を返す
ROUND(フィールド名, 桁数)	該当フィールド内の値を指定した桁数で丸める ●桁数は省略可能（一の位で丸める） ●中間の値（一の位に丸める場合の「.5」や十の位に丸める場合の「5」）の場合は「0」から遠ざかるように丸める ●桁数が正の場合は小数部、負の場合は整数部を丸める

さっそくSQL文を記述してみましょう。以下の［7-1-2］では、関数ごとの処理の違いが一覧できるように、3つの関数を一度に記述しました。

結果テーブルを見ると、それぞれの関数で数値を丸めていることが分かると思います。戻り値のデータ型はいずれも「FLOAT64」型なので、小数点以下が残っています。

7-1-2 さまざまな数値を丸める

```
WITH master AS (
SELECT 12345.6789 AS number
UNION ALL
SELECT 98765.4321
UNION ALL
SELECT -12345.6789
UNION ALL
SELECT -98765.4321
)
SELECT
FLOOR(number) AS floor_number
, CEIL(number) AS ceil_number
, ROUND(number) AS round_number
, ROUND(number, 2) AS round_number2
, ROUND(number, -2) AS round_number3
FROM master
```

結果テーブル

行	floor_number	ceil_number	round_number	round_number2	round_number3
1	12345.0	12346.0	12346.0	12345.68	12300.0
2	98765.0	98766.0	98765.0	98765.43	98800.0
3	-12346.0	-12345.0	-12346.0	-12345.68	-12300.0
4	-98766.0	-98765.0	-98765.0	-98765.43	-98800.0

FLOOR/CEIL/ROUND 関数で数値を丸めた

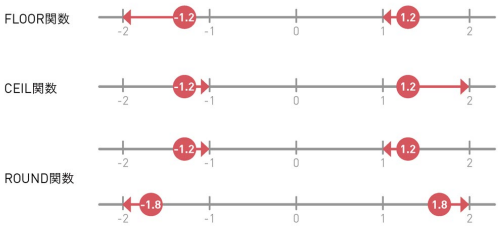
冒頭のWITH句では［number］フィールドとして、「12345.6789」などの4つの値が格納された仮想テーブルをUNION ALLで作成しています。縦につなぐ理由は、実体のあるテーブルを参照しない場合、SELECT句は1レコードしか作らないためです。

このテクニックは、動作確認に適当なテーブルがなく、わざわざCSVファイルをアップロードしてテーブルを作るまでもないときに有効です。関数とは関係ありませんが、ぜひ覚えておいてください。

さて、FLOOR関数、CEIL関数、ROUND関数は、それぞれ数値を丸める方向が異なります。特にイメージしにくいのが、丸めたい数値がマイナス（負）の場合です。その疑問に答える図を以下の［7-1-3］で示します。

切り捨てのFLOOR関数は「より小さい」方向、切り上げのCEIL関数は「より大きい」方向、四捨五入のROUND関数は「近い整数」に丸められていることが分かるでしょうか。［7-1-2］にはマイナスの数値を丸めた結果も含まれているので、あらためて確認しておいてください。

7-1-3 FLOOR/CEIL/ROUND関数で数値を丸める方向



絶対値を求めるABS関数

ビジネスデータの分析において、絶対値が必要とされるシーンはそう多くはありません。しかし、平均との差異について「乖離の幅」だけが必要であって、

上回っているか、下回っているかは問わない場合、絶対値で戻り値を取得するようなケースが考えられます。

SQLで絶対値を求めるには、**ABS関数**（エービーエス）を使います。ABSは「絶対の」という意味を表す英語の「absolute」（アブソリュート）からきています。

例えば、[pref] フィールドに都道府県名、[area] フィールドに面積の値が格納された［small_prefarea］テーブルが存在すると考えてください。このテーブルを元に、都道府県別の面積を全都道府県の平均面積と比較し、その差が大きい順に並べたいとします。SQL文と結果テーブルは、以下の［7-1-4］の通りです。

内側のサブクエリで、元のテーブルに全都道府県の平均面積を集計した［avg_area］フィールドを加えた仮想テーブルを作成しています。そのうえで、3行目にあるメインのクエリのABS関数で、平均面積との差の絶対値を［abs_diff_from_avg］フィールドに格納し、すべてのレコードを［abs_diff_from_avg］フィールドの降順に並べ替えています。

次ページの結果テーブルを見ると、北海道や岩手県といった平均面積から大きいほうに乖離している都道府県に交ざって、香川県、大阪府、東京都といった平均面積から小さいほうに乖離している都道府県も上位に入り、意図通りのデータが得られていることが確認できます。

7-1-4 差の絶対値を求めて並べ替える

```
SELECT *
, area - avg_area AS diff_from_avg
, ABS(area - avg_area) AS abs_diff_from_avg
FROM (SELECT *
, (SELECT AVG(area) FROM sample.small_prefarea)
AS avg_area
FROM sample.small_prefarea)
ORDER BY 5 DESC
```

結果テーブル

行	pref	area	avg_area	diff_from_avg	abs_diff_from_avg
1	北海道	83456.2	8039.619574468084	75416.58042553191	75416.58042553191
2	岩手県	15278.77	8039.619574468084	7239.150425531917	7239.150425531917
3	香川県	1876.47	8039.619574468084	-6163.149574468083	6163.149574468083
4	大阪府	1896.83	8039.619574468084	-6142.789574468084	6142.789574468084
5	東京都	2187.42	8039.619574468084	-5852.1995744680835	5852.1995744680835
6	沖縄県	2275.28	8039.619574468084	-5764.339574468084	5764.339574468084

ABS 関数により、差の絶対値に基づく順序で並べ替えできた

割り算の余りを求めるMOD関数

MOD関数（モッド）は、割り算の余りを求める関数です。例えば「10」を「3」で割ったときの余り「1」が取得できます。この関数がどのようなときに使われるのか、疑問に思うかもしれません。

確かに、あまり頻繁には利用しませんが、筆者の場合は「サンプリング」するときに利用します。例えば、会員番号や、Googleアナリティクスがブラウザーを識別するIDである「cid」のように、時系列で順番に、あるいはランダムに採番されているフィールドを対象に「レコード数を○分の1にしたい」と考えるときに利用します。

今、データをざっくりと半分にしたいとしましょう。会員番号を「2」で割った余りを「0」または「1」で絞り込めば、偶数か奇数かで会員番号を分別して、ほぼ半分のレコードに絞り込めます。

同じように、レコード数をおよそ5分の1に絞り込みたいときは、会員番号を「5」で割って『余りが「0」に等しい』という条件で絞り込めば、会員番号が「5」で割り切れる数字のユーザーだけに絞り込まれます（余りは「0」～「4」の数値であれば5分の1になります）。

次ページのSQL文［7-1-5］では、演習用の［customers］テーブルに対して『会員番号を「7」で割って余りが「0」になる』という条件で絞り込んでいます。もともと497レコードあったテーブルが67レコードと、およそ7分の1になりますので、みなさんも試してみてください。

7-1-5 レコード数をおよそ1/7に絞り込む

```
SELECT * FROM sample.customers
WHERE MOD(user_id, 7) = 0
```

乱数を発生させるRAND関数

RAND関数（ランド）は、擬似乱数を発生させる関数です。この「擬似」とは、数学的な厳密さで発生させた乱数ではないという意味です。ただ、通常のマーケティングなどの業務における分析には、規則性のないランダムな値として活用できます。戻り値は「0」以上「1」未満のFLOAT64型となります。

以下のSQL文[7-1-6]を実行するたびに「RAND()」の結果は異なる値を返し、「LIMIT 10」で絞り込むユーザーが変わります。演習用テーブルを対象としているので、みなさんも試してもらうと、ユーザーがランダムに選ばれた10人に絞り込まれることが分かるはずです。

なお、ランダムに取得したデータを残しておきたい場合は、結果テーブルをビューではなく、テーブルとして保存する必要があります。

7-1-6 全レコードからランダムに10件を取得する

```
SELECT * FROM sample.customers
ORDER BY RAND()
LIMIT 10
```

STEP UP

データ型を変換するCAST関数

本節で学んだ数値に対する関数は、戻り値のデータ型がFLOAT64型となるものがほとんどでした。ただ、FLOAT64型のデータは小数点以下が0であっても小数部が残るため、結果テーブルでの値の表現が冗長だったり、見にくかったりすることがあります。

例を見てみましょう。以下のSQL文［7-1-7］は、「5.2」という値をROUND関数で四捨五入し、整数に変換しています。結果は5ですが、戻り値がFLOAT64型のデータとなるため、結果テーブルに格納されるのは「5.0」という小数部がある値となります。

7-1-7 四捨五入する

```
1  SELECT ROUND(5.2) AS value
```

小数点以下の「0」を表示させないようにするには、戻り値のデータ型をFLOAT64型から、整数型である「INT64」型に変換する必要があります。そのときに利用するのが**CAST関数**（キャスト）です。

例えば、「5.2」を四捨五入した値をINT64型のデータとして取得したい場合は、次ページの［7-1-8］のように記述します。CAST関数の引数として、変換する対象とASに続いてデータ型を指定する形です。変換対象にはフィールド名も指定できます。

みなさんも実行してみてください。結果テーブルに「5」が返され、小数点以下が取り除かれて整数となることが分かるはずです。

7-1-8 四捨五入してデータ型を変換する

```
CAST(ROUND(5.2) AS INT64)
```

CAST関数は、FLOAT64型からINT64型だけでなく、ほかのデータ型への変換にも利用できます。以下のSQL文［7-1-9］で動作を確認してください。このSQL文の中で変換しているデータは、次ページの表［7-1-10］に一覧としてまとめています。

注意点としては、数値ではない「東京」などの文字列をINT64型に変換したり、「20200707」といった日付の体裁が整っていない値（BigQueryでは「二千二十万七百七」という整数と解釈されます）をDATE型に変換したりすることはできません。

7-1-9 さまざまな値のデータ型を変換する

```
SELECT
CAST(ROUND(5.2) AS INT64) AS value1
, CAST(5 AS FLOAT64) AS value2
, CAST(5.2 AS STRING) AS value3
, CAST(5.2 AS INT64) AS value4
, CAST("5.23" AS FLOAT64) AS value5
, CAST(TRUE AS STRING) AS value6
, CAST("FALSE" AS BOOL) AS value7
, CAST(DATE("2020-07-07") AS STRING) AS value8
, CAST("2020-07-07" AS DATE) AS value9
```

結果テーブル

行	value1	value2	value3	value4	value5	value6	value7	value8	value9
1	5	5.0	5.2	5	5.23	true	false	2020-07-07	2020-07-07

CAST 関数により各種データ型に変換できた

7-1-10 データ型の変換内容

行数	変換前のデータ型	変換後のデータ型
2行目	FLOAT64	INT64
3行目	INT64	FLOAT64
4行目	FLOAT64	STRING
5行目	FLOAT64	INT64
6行目	STRING	FLOAT64
7行目	BOOL	STRING
8行目	STRING	BOOL
9行目	DATE	STRING
10行目	STRING	DATE

結果テーブルの各フィールドのデータ型は、結果テーブル上部の［ジョブ情報］から［一時テーブル］に進むと確認できます（SECTION 2-6を参照）。

SECTION

7-2 文字列を扱う関数

本節では文字列を扱う関数を学んでいきましょう。

数値のように足したり掛けたりではなく、文字列を見やすいように加工したいことが確かにありますね。

そうですね。Excelでもおなじみですが、文字列と文字列を連結したり、文字列中の一部だけを取り出したりといった処理を、SQLの関数で実行してみましょう。

文字列やフィールドの値を連結するCONCAT関数

文字列同士を連結するには、CONCAT関数（コンカティネート）を使います。CONCATは「連結する」という意味の「concatenate」に由来しています。

引数としては、連結する文字列を「,」でつないで記述し、固定文字列は「'」または「"」で囲みます。引数にはフィールド名も指定可能です。

例えば、演習用の［customers］テーブルの［name］フィールドには顧客名が格納されていますが、その値の末尾に「様」を追加するSQL文は以下の［7-2-1］のようになります。

7-2-1 文字列を追加する

```
1  SELECT CONCAT(name, "様") AS atesaki
2  FROM sample.customers
```

結果テーブル

行	atesaki
1	石塚 拓様
2	長坂 賢介様
3	野中 裕之様
4	高松 龍様

CONCAT 関数により顧客名の末尾に「様」が追加された

また、フィールドの値同士の連結も可能です。基本的には文字列型のフィールドが連結の対象ですが、文字列型以外のフィールドはBigQueryが自動的に「文字列だとみなして」連結してくれる場合があります。

以下の［7-2-2］では、[web_log］テーブルにある文字列型の［cid］フィールドと数値型の［session_count］を、間に固定文字列「-」(ハイフン）を入れてCONCAT関数で連結し、「セッションのID」の性質を持つ文字列を作成しています。

7-2-2 文字列型と数値型のデータを連結する

```
1  SELECT CONCAT(cid, "-", session_count) AS session_id
2  FROM sample.web_log
```

結果テーブル

行	session_id
1	1104331433.1508336-1
2	1104331433.1508336-1
3	1104331433.1508336-1
4	1205881588.1391075-1
5	1207001700.1514213-1

文字列型の［cid］と整数型の［session_count］を連結できた

文字列の一部を取り出すSUBSTR関数

文字列から一部を取り出すには、**SUBSTR関数**を使います。この関数名は、英語で「部分文字列」を表す「substring」(サブストリング）から来ています。

まったく同一の機能でSUBSTRING関数も使えますが、ここではSUBSTR関数として説明します。

3つの引数があるため、まずは構文を以下の［7-2-3］で示します。

7-2-3 SUBSTR関数の構文

```
1  SUBSTR(対象文字列, 位置, 文字数)
```

1つ目の引数［文字列］には、固定文字列のほかフィールド名も指定できます。2つ目の引数［位置］に指定した場所から、3つ目の引数［文字数］で指定した文字数分を取り出します。[文字数]は省略が可能で、省略した場合は[位置］から末尾までの文字列を取得します。

以下の表［7-2-4］では、[文字列］として「本日は晴天なり」を記述し、[位置]と[文字数]を変化させた場合に取得される戻り値の例をまとめました。[位置］にマイナスの値を記述すると、文字列の右側から数えた場所を指定することができます。

実際の記述例としては、次ページのSQL文［7-2-5］となります。表と同じ戻り値が結果テーブルで得られていることが分かると思います。

7-2-4 SUBSTR関数の記述例

記述例	戻り値
SUBSTR("本日は晴天なり", 1, 2)	本日
SUBSTR("本日は晴天なり", 4, 2)	晴天
SUBSTR("本日は晴天なり", 4)	晴天なり
SUBSTR("本日は晴天なり", -4, 2)	晴天
SUBSTR("本日は晴天なり", -4)	晴天なり

7-2-5 文字列の一部を取り出す

```
WITH master AS (
SELECT "本日は晴天なり" AS string
)

SELECT SUBSTR(string, 1, 2) AS from1_2letters
, SUBSTR(string, 4, 2) AS from4_2letters
, SUBSTR(string, 4) AS from4_all_letters
, SUBSTR(string, -4, 2) AS from_reverse4_2letters
, SUBSTR(string, -4) AS from_reverse4_all_letters
FROM master
```

結果テーブル

行	from1_2letters	from4_2letters	from4_all_letters	from_reverse4_2letters	from_reverse4_all_letters
1	本日	晴天	晴天なり	晴天	晴天なり

SUBSTR 関数で文字列の一部を取得できた

なお、SUBSTR関数は文字列の好きな位置から部分文字列を取得できましたが、取得開始位置が左端の場合には**LEFT関数**(レフト)が、右端の場合には**RIGHT関数**(ライト)が利用できます。それぞれ記述例と戻り値は表［7-2-6］の通りです。

7-2-6 LEFT/RIGHT関数の記述例

記述例	戻り値
LEFT("本日は晴天なり", 3)	本日は
RIGHT("本日は晴天なり", 3)	天なり

特定文字列の出現位置を取得するINSTR関数

SUBSTR関数と組み合わせて利用することの多い、**INSTR関数**も覚えておきましょう。文字列の中での、特定文字列の出現位置を整数で返す関数です。ExcelのFIND関数と似た働きをし、構文は以下の［7-2-7］になります。

7-2-7 INSTR関数の構文

```
1  INSTR(対象文字列, 検索文字列, 検索開始位置, 出現回数)
```

対象文字列の中に検索文字列が見つからなかった場合は「0」が返ります。引数のうち［検索開始位置］と［出現回数］はオプションです。省略すると1文字目から検索し、最初に見つかった位置を返します。記述例と戻り値をまとめると、以下の表［7-2-8］のようになります。SQL文の記述例は、次ページの［7-2-9］を参照してください。

7-2-8 INSTR関数の記述例

記述例	戻り値
INSTR("本日は晴天晴天なり" , "晴天")	4
INSTR("本日は晴天晴天なり" , "晴天", 5)	6
INSTR("本日は晴天晴天なり", "日", 3)	0
INSTR("本日は晴天晴天なり", "晴天", 1, 2)	4
INSTR("本日は晴天晴天なり", "晴天", 4, 2)	6
INSTR("本日は晴天晴天なり", "晴天", 5, 2)	0
INSTR("本日は晴天晴天なり", "晴天",1, 3)	0
INSTR("本日は晴天なり晴天", "晴天", 1, 2)	8

7-2-9 特定文字列の出現位置を取得する

```
1  SELECT INSTR("本日は晴天晴天なり", "晴天") AS result1
2  , INSTR("本日は晴天晴天なり", "晴天", 5) AS result2
3  , INSTR("本日は晴天晴天なり", "日", 3) AS result3
4  , INSTR("本日は晴天晴天なり", "晴天", 1, 2) AS result4
5  , INSTR("本日は晴天晴天なり", "晴天", 4, 2) AS result5
6  , INSTR("本日は晴天晴天なり", "晴天", 5, 2) AS result6
7  , INSTR("本日は晴天晴天なり", "晴天", 1, 3) AS result7
8  , INSTR("本日は晴天なり晴天", "晴天", 1, 2) AS result8
```

結果テーブル

行	result1	result2	result3	result4	result5	result6	result7	result8
1	4	6	0	6	6	0	0	8

INSTR 関数により文字列の出現位置を取得できた

「0」は「存在しない」を示すため、特定の文字列が含まれているかどうかの識別にも利用できる

SUBSTR関数とINSTR関数を組み合わせる具体例を見てみましょう。以下の［7-2-10］にあるSQL文と結果テーブルを見てください。2行目と4行目にあるURLから、第一ディレクトリ名である「company」と「column」という文字列を取り出しています。

8行目にSUBSTR関数があり、その引数として9〜10行目でINSTR関数が記述されています。やや読み解くのが難しいかもしれませんが、みなさんも実行してみて、ほかのURLでも試してみてください。

7-2-10 URLから一部の文字列を取り出す

```
1  WITH master AS (
2  SELECT "www.principle-c.com/company/index.html" AS url
```

```
UNION ALL
SELECT "www.principle-c.com/column/seo/index.html"
)

SELECT *,
SUBSTR(URL,
INSTR(URL, "/") + 1,
INSTR(URL, "/", 1, 2) - INSTR(URL, "/", 1, 1) -1)
AS directory_name
FROM master
```

結果テーブル

行	url	directory_name
1	www.principle-c.com/company/index.html	company
2	www.principle-c.com/column/seo/index.html	column

SUBSTR 関数と INSTR 関数により、
文字列の一部を取得できた

なお、INSTR関数の簡易版として**STRPOS関数**があります。STRPOS関数では、INSTR関数の引数である［検索開始位置］と［出現回数］を指定できないので、出現回数が１回だけの条件で、対象文字の中の先頭から特定文字を検索するときに利用してください。構文と記述例は［7-2-11］と［7-2-12］の通りです。

7-2-11 STRPOS関数の構文

```
STRPOS(対象文字列, 検索文字列)
```

7-2-12 STRPOS関数の記述例

記述例	戻り値
STRPOS("本日は晴天なり", "日")	2
STRPOS("本日は晴天なり", "山")	0
STRPOS("本日は晴天なり", "なり")	6
STRPOS("本日は晴天なり", "なる")	0

特定の文字列を置換するREPLACE関数

文字列中の特定の文字列を別の文字列で置き換えるのが、**REPLACE関数**（リプレイス）です。構文は以下の［7-2-13］の通りです。

7-2-13 REPLACE関数の構文

```
1  REPLACE(対象文字列, 検索文字列, 置換後の文字列)
```

引数［対象文字列］に含まれる［検索文字列］を探し、それを［置換後の文字列］に置換した文字列が戻り値となります。［対象文字列］に［検索文字列］が含まれていない場合は、何も起きずに元の文字列（対象文字列）が返されます。

次ページのSQL文［7-2-14］では、2行目と4行目にある文字列の一部を、8〜10行目のREPLACE関数で置換しています。「本日は晴天なり」の「晴」が「曇」に、「principle-c.com/company/」の「principle-c.com」が「sample.jp」に置換されていることが分かります。

同時に、対象文字列に検索文字列が含まれない場合、対象文字列がそのまま返されることも理解できるかと思います。

7-2-14 特定の文字列を置換する

```
1  WITH master AS(
2  SELECT "本日は晴天なり" AS string
3  UNION ALL
4  SELECT "principle-c.com/compamy/"
5  )
6
7  SELECT string AS original
8  , REPLACE(string, "晴", "曇") AS sei_to_don
9  , REPLACE(string, "principle-c.com", "sample.jp")
10 AS principle_to_sample
11 FROM master
```

結果テーブル

行	original	sei_to_don	principle_to_sample
1	本日は晴天なり	本日は曇天なり	本日は晴天なり
2	principle-c.com/compamy/	principle-c.com/compamy/	sample.jp/compamy/

REPLACE 関数により文字列の一部を置換できた

文字列の長さを取得するLENGTH関数

LENGTH関数（レングス）は、文字列の長さを取得します。取り出しや置換といった文字列の加工とは異なり、長さを数値として取得する関数です。

単体で使うことはあまりないかもしれませんが、REPLACE関数と組み合わせた面白い活用法を紹介しましょう。例えば、セミナーのアンケート結果が［small_seminar］テーブルにあり、自由回答の感想が［text］フィールドに格納されているとします。このフィールド内の文字列から「満足」という言葉が何回出現するかを調べてみます。

REPLACE関数を利用して、対象文字列の「満足」という単語を「""」で置換すると、「満足」をすべて消去できます。元の文字列の長さから「満足」を消去した後の長さを差し引くと、「満足」を消去したことで削減された文字数が分かります。文字列に「満足」が1つもなければ長さは変わりませんが、2つあったのであれば4文字削減されているでしょう。

つまり、差分は「満足」の2文字×出現回数となるので、「満足」の文字数「2」で割れば出現回数が求められるわけです。以下のSQL文 [7-2-15] を参照して、実際の方法を確認してください。

アンケートの回答などで、特定の単語が何回出現したかを調べたいことは、ままあります。LENGTH関数とREPLACE関数を使ったこの方法であれば、簡単に調べることができます。

7-2-15 文字列の長さと出現回数を取得する

```
1  SELECT text
2  , LENGTH(text) AS text_length
3  , LENGTH(text) - LENGTH(REPLACE(text, "満足", ""))
4  AS length_diff
5  , (LENGTH(text) - LENGTH(REPLACE(text, "満足", "")))/2
6  AS manzoku_no_kosuu
7  FROM sample.small_seminar
```

結果テーブル

行	text	text_length	length_diff	manzoku_no_kosuu
1	大変満足の行くセミナーでした。ありがとうございます。	26	2	1.0
2	満足感が半端ない。この満足を味わえるのだったら有料でも良いです。	32	4	2.0

LENGTH 関数と REPLACE 関数により、
文字列の長さと出現回数を取得できた

SECTION

7-3 正規表現を利用する関数

前節で学んだ文字列に対する関数では、固定的な文字を扱いました。本節では、より柔軟な扱いが可能な「正規表現」を利用する関数について学びましょう。

正規表現は聞いたことがあります。でも、使ったことはありません……。

非常にパワフルな機能です。例えば「Tから始まる文字列」「連続する8桁の数字」など、パターンで文字列を扱うことができます。

特定のパターンに合致する文字列を抽出できる

「正規表現」とは、**「メタ文字」と呼ばれる記号を使って、文字列の中にある特定のパターンを表現するための記述方法**のことです。正規表現を利用することで、ある文字列から任意のパターンに合致する文字列の有無を判定したり、抽出したりすることができます。

前節で学んだ文字列の出現位置を調べるINSTR関数では、対象文字列の中の「/」の出現位置を調べるなど、固定的な文字列を検索の対象としていました。一方、正規表現を使った関数では、文字列のパターンを処理の対象とするため、より柔軟に文字列を取り扱うことができます。

正規表現を理解するための、ごく簡単な例を1つ紹介しましょう。正規表現で使えるメタ文字として「^」(ハット) があり、「次の文字で始まる」ことを意味します。「^T」と記述すると「Tで始まる」という意味になるのですが、次の文字列のうち「^T」に合致するのはどれでしょうか？

- TOKYO
- CHIBA
- KYOTO
- TOCHIGI

合致するのは「TOKYO」「TOCHIGI」です。「KYOTO」にも「T」が含まれていますが、「T」で始まってはいないので、合致しません。「^T」というたった2文字を記述するだけで、「Tで始まる」どのような文字列も探すことができるわけです。

ここではSQLで正規表現を利用できる関数として、3つの関数を学びます。順に見ていきましょう。

正規表現を利用できる関数名の冒頭につく「REGEXP」は「Regular Expression」の略で、「正規表現」という意味です。

文字列の有無を判定するREGEXP_CONTAINS関数

REGEXP_CONTAINS関数は、対象とする文字列に正規表現で指定したパターンの文字列が含まれているかどうかを判定します。CONTAINSは「含む」という意味なので、分かりやすいですね。含まれていれば「true」が返されます。構文は以下の［7-3-1］の通りです。

7-3-1 REGEXP_CONTAINS関数の構文

```
REGEXP_CONTAINS(対象文字列, 正規表現)
```

利用例として、Webサイトのユーザー投稿を想定してみましょう。投稿内容は追ってサイト上で公開するため、「電話番号を記述してはいけない」という規定を設けています。その規定に違反している投稿がないかをチェックしたいとします。電話番号の疑いがある文字列のパターンとしては、次の2点が考えられるでしょう。

- 8つ以上連続した数字
- 4桁の数字と「-」と4桁の数字が連続した文字列

投稿は［small_webpost］テーブルの［post］フィールドに格納されているとすると、REGEXP_CONTAINS関数を記述したSQL文と結果テーブルは以下の［7-3-2］のようになります。

7-3-2 特定のパターンに合致する文字列を判定する

```
1  SELECT post
2  , REGEXP_CONTAINS(post, r"\d{8}\d*|\d{4}-\d{4}")
3  AS phone_number_violation
4  FROM sample.small_webpost
```

結果テーブル

行	post	phone_number_violation
1	この映画、最高だった。888888！	false
2	この映画のファンは、09012345678までショートMください。03-1234-1234でもOK	true
3	パンフレットをプレゼントするので03-9999-1111まで連ください。	true

電話番号の疑いがある文字列を含む投稿を判定できた

REGEXP_CONTAINS関数は、SQL文の2行目にあります。第1引数には対象のフィールド名、第2引数には「r"\d{8}\d*|\d{4}-\d{4}"」と記述しています。**「r」は正規表現であることを示す目印**です。

正規表現を読み解いてみましょう。まず、中央にある「|」(パイプ)は「または」を表すメタ文字です。よって、正規表現全体を「\d{8}\d*」または「\d{4}-\d{4}」と分解できます。

「\d{8}\d*」のうち「\d」は任意の数字を表し、「{8}」は「直前の文字が8回連続している」という意味になります。その後ろの「\d*」は「0回以上の数字の繰り返し」です。したがって「\d{8}\d*」は「最低8個連続した数字で、さらに数字が続いてもよい」となります。

同様に「\d{4}-\d{4}」を読み解くと、『4回連続した数字と「-」と4回連続した数字』となります。

結果テーブルを見ると、レコードの1行目は数字の「8」が連続していますが、6個なので「false」と判定されました。2〜3行目は正規表現に合致する文字列が含まれているので「true」が返されています。

合致する文字列を返すREGEXP_EXTRACT関数

REGEXP_EXTRACT関数は、対象とする文字列に正規表現で指定したパターンの文字列が含まれている場合、合致した文字列を1つ返します。構文は以下の［7-3-3］となります。引数のうち［検索開始位置］と［出現回数］はオプションで、省略が可能です。

7-3-3 REGEXP_EXTRACT関数の構文

```
1  REGEXP_EXTRACT(対象文字列，正規表現，検索開始位置，出現回数)
```

正規表現に合致する文字列が2つ以上あるとき、オプションを指定しない場合は、最初に合致する文字列が取得できます。オプションの［検索開始位置］は、対象文字列を先頭から検索せず、特定の位置以降で検索する場合に指定します。同じくオプションの［出現回数］は、正規表現に合致する表現が複数存在するとき、その何個目を取得するかを指定します。

先ほどのユーザー投稿の例を続け、投稿内容のどの部分が規定違反に当たるのかを確認してみましょう。以下の［7-3-4］の通りにSQL文を記述すると、電話番号と疑われる文字列を取得できます。

SQL文では2行目と4行目にREGEXP_EXTRACT関数を記述しており、2行目はオプションなし、4行目はオプションありとしています。結果テーブルを見ると、[violating_number1］フィールドでは合致した最初の文字列、[violating_number2］では2個目に合致した文字列が返されていることが分かります。

7-3-4 特定のパターンに合致した文字列を取得する

```
1  SELECT post
2  , REGEXP_EXTRACT(post, r"\d{8}\d*|\d{4}-\d{4}")
3  AS violating_number1
4  , REGEXP_EXTRACT(post, r"\d{8}\d*|\d{4}-\d{4}", 1, 2)
5  AS violating_number2
6  FROM sample.small_webpost
```

結果テーブル

行	post	violating_number1	violating_number2
1	この映画、最高だった。888888！	null	null
2	この映画のファンは、09012345678までショートMください。03-1234-1234でもOK	09012345678	1234-1234
3	パンフレットをプレゼントするので03-9999-1111まで連ください。	9999-1111	null

電話番号の疑いがある文字列を取得できた

合致する文字列を置換するREGEXP_REPLACE関数

REGEXP_REPLACE関数は、対象とする文字列に正規表現で指定したパターンの文字列が含まれている場合、任意の文字列に置換します。構文は次ページの［7-3-5］となります。

7-3-5 REGEXP_REPLACE関数の構文

```
REGEXP_REPLACE(対象文字列, 正規表現, 置換後の文字列)
```

先ほどのユーザー投稿の例で、今度は規約違反の投稿内容のうち、電話番号と疑われる部分を伏せ字にしたいとしましょう。以下のSQL文［7-3-6］では、該当部分を「--------」で置換しています。

REGEXP_REPLACE関数は、正規表現に合致する文字列をすべて置換します。結果テーブルの2レコード目を見ると、最初の文字列「09012345678」だけでなく、2番目の文字列「1234-1234」も「--------」に置換されていることが分かります。

7-3-6 特定のパターンに合致した文字列を置換する

```
SELECT post
, REGEXP_REPLACE(post, r"\d{8}\d*|\d{4}-\d{4}"
, "--------")
AS replaced_number
FROM sample.small_webpost
```

結果テーブル

行	post	replaced_number
1	この映画、最高だった。888888！	この映画、最高だった。888888！
2	この映画のファンは、09012345678までショートMください。03-1234-1234でもOK	この映画のファンは、----までショートMください。03----でもOK
3	パンフレットをプレゼントするので03-9999-1111まで連ください。	パンフレットをプレゼントするので03----まで連ください。

電話番号の疑いがある文字列を
別の文字列に置換できた

STEP UP

正規表現を使ったテクニック

正規表現を利用した多様な文字列をまとめる実践例を2つ紹介します。

▶都道府県を地域にまとめる

[prefecture]フィールドに格納されている都道府県名を「東北」「関東」などの地域にまとめたいとします。このとき、正規表現を使わないと、47都道府県に対してCASE句を記述する必要があります。

一方、REGEXP_CONTAIN関数を使うと、以下の［7-3-7］の通りに記述できます。「|」でつながれた複数の県が「東北」にまとめられていることが確認できます。

7-3-7 正規表現で都道府県を地域にまとめる

```
SELECT pref,
CASE
WHEN REGEXP_CONTAINS(pref, r"青森|岩手|宮城|秋田")
IS TRUE THEN "東北"
ELSE "その他"
END AS region
FROM sample.small_pref
```

結果テーブル

行	pref	region
1	青森	東北
2	岩手	東北
3	秋田	東北
4	宮城	東北
5	北海道	その他

東北地方の県を「東北」としてまとめられた

▶多様なパターンのURIを1つにまとめる

サイト訪問者が同じWebページを表示していても、リクエストURIは意外と揺らぐものです。そのため、直帰率などを適切に計算することが難しいケースがあります。

以下の［7-3-8］にあるSQL文と結果テーブルを見てください。結果テーブルの［uri］フィールドには4つのリクエストURIがありますが、ユーザーが見ているWebページはすべて同一です。そこで、REGEXP_REPLACE関数により、すべてを「/」にまとめた値を［united_uri］フィールドとして取得しました。

7-3-8 正規表現でリクエストURIをまとめる

```
1  SELECT *
2  , REGEXP_REPLACE (uri, r"^/(index\.html)?\??.*"
3  , "/")
4  AS united_uri
5  FROM sample.small_uri
```

結果テーブル

行	uri	session	bounce	united_uri
1	/	100	40	/
2	/?utm_source=google	150	50	/
3	/index.html	200	80	/
4	/index.html?utm_source=yahoo	250	100	/

不統一なリクエストURIを「/」としてまとめられた

SECTION

7-4 日付・時刻を扱う関数

続いて見ていくのは、日付・時刻を扱う関数です。日付・時刻というと単純に思えますが、実はかなりの種類があります。

どのようなことができるのですか？

実務では、現在の日付・時刻を取得して最終購入日からの経過日数を求めたり、2つの日付の間の日数を求めたりする用途が想定できます。

現在の日付・時刻を取得する関数

BigQueryには、「今、この瞬間」の日付・時刻を取得する関数として、「CURRENT」と付く4つの関数が用意されています。以下の表［7-4-1］にまとめたように、これらは扱うデータ型によって使い分けます。

7-4-1 日付・時刻の取得に利用する関数

関数	データ型	戻り値の例
CURRENT_DATE	DATE型	2016-12-25
CURRENT_DATETIME	DATETIME型	2016-05-19T10:38:47.046465
CURRENT_TIME	TIME型	15:31:38.776361
CURRENT_TIMESTAMP	TIMESTAMP型	2020-06-02 23:57:12.120174 UTC

まず、よく利用されるCURRENT_DATE関数（カレント・デイト）とCURRENT_DATETIME関数（カレント・デイトタイム）を使って、現在の日付・時刻を取得してみましょう。以下のSQL文［7-4-2］をみなさんの環境で実行してみてください。「今、この瞬間」の日付・時刻が結果テーブルに表示されるはずです。

これらの関数の戻り値は「UTC」（協定世界時）のため、日本での日付・時刻を取得するには「CURRENT_DATE("Asia/Tokyo")」のようにタイムゾーン表記を付与する必要があります。日本時間はUTCからプラス９時間となるため、「CURRENT_DATE("+９")」でも同じ結果が得られます。

7-4-2 日本の日付・時刻を取得する

```
1  SELECT
2  CURRENT_DATE("Asia/Tokyo") AS today
3  , CURRENT_DATE("+9") AS today2
4  , CURRENT_DATETIME("Asia/Tokyo") AS now
5  , CURRENT_DATETIME("+9") AS now2
```

結果テーブル

行	today	today2	now	now2
1	2021-01-26	2021-01-26	2021-01-26T16:40:50.362731	2021-01-26T16:40:50.362731

日本の現在の日付・時刻を取得できた

日付・時刻に一定の値を加える関数

「本日から3週間後」「現在時刻から6時間後」など、日付や日付・時刻に一定の値を加えたいときに使うのが、DATE_ADD関数（デイト・アド）とDATETIME_ADD関数（デイトタイム・アド）です。関数はそれぞれ［7-4-3］と［7-4-4］となります。

7-4-3 DATE_ADD関数の構文

```
1  DATE_ADD(DATE型の値, INTERVAL 加える整数 デイトパート)
```

7-4-4 DATETIME_ADD関数の構文

```
1  DATETIME_ADD(DATETIME型の値, INTERVAL 加える整数 パート)
```

引数の［加える数値］の直前にある「INTERVAL」は定型句です。［デイトパート］と［パート］には、数値を加える対象のキーワードを指定します。「日」を加えるなら「DAY」、「時」を加えるなら「HOUR」といった具合です。指定できるキーワードは次の通りです。

●デイトパート
⇒YEAR / QUARTER / MONTH / WEEK / DAY
●パート
⇒YEAR / QUARTER / MONTH / WEEK / DAY / HOUR / MINUTE / SECOND / MILLISECOND / MICROSECOND

例えば、本日から3週間後（21日後と同じ意味）、現在時刻から6時間後を取得するには、以下の［7-4-5］のように記述します。

7-4-5 3週間後の日付と6時間後の時間を取得する

```
SELECT
CURRENT_DATE("+9") AS today
, DATE_ADD(CURRENT_DATE("+9"), INTERVAL 3 WEEK)
AS three_weeks_later
, CURRENT_DATETIME("+9") AS now
, DATETIME_ADD(CURRENT_DATETIME("+9"), INTERVAL 6 HOUR)
AS six_hours_later
```

ここで覚えておきたい点が2つあります。1つは「1か月後」の戻り値です。以下の［7-4-6］のSQL文を実行し、DATE_ADD関数で「5月31日」に1か月加えると、いつが返ってくるでしょうか？ 6月に31日はないので「6月30日」でしょうか？ それとも31日後の「7月1日」でしょうか？ 正解は「6月30日」です。つまり、翌月の末日が求められます。

もう1つは「INTERVAL」の後ろの数値についてです。これにはマイナスの整数も指定できます。日付・時刻から一定の値を引くにはDATE_SUB関数やDATETIME_SUB関数を利用する方法もありますが、「INTERVAL」の後にマイナスの整数を指定すれば、これらの関数は必要はありません。

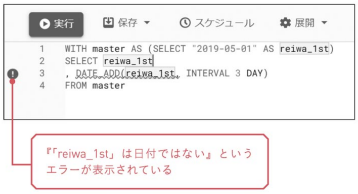
7-4-6 1か月後の日付を取得する

```
SELECT DATE_ADD("2021-5-31", INTERVAL 1 MONTH)
AS one_month_later
```

DATE型やDATETIME型のオブジェクトを作成する関数

定数のように、特定の日付・時刻の値が必要なことがあります。例えば、令和元年は2019年5月1日から始まりました。その日付を定数として取得しようとして、以下の［7-4-7］のようにWITH句を使って記述しました。しかし、DATE_ADD関数の第1引数［DATE型の値］がSTRING型になり、エラーとなってしまいました。

7-4-7 データ型の間違いによるエラー



このようなときに使うのがDATE関数（デイト）です。エラーは「2019-05-01」が文字列として認識されていることが原因なので、次ページの［7-4-8］のように、DATE関数を使ってDATE型として認識させるとエラーは解消します。

同様に、文字列をDATETIME型として認識させるには、DATETME関数（デイト・タイム）を利用します。認識されているデータ型は、結果テーブルのスキーマを見ることで確認できます。

この例においても、データベースでは「データ型」が重要だということが理解できます。

7-4-8 文字列をDATE型として認識させる

```
WITH master AS (SELECT DATE "2019-05-01" AS reiwa_1st)

SELECT reiwa_1st
, DATE_ADD(reiwa_1st, INTERVAL 3 DAY)
FROM master
```

そのほか、DATE関数とDATETIME関数を使ったオブジェクトの作成方法も覚えておきましょう。以下の表［7-4-9］のようにDATETIME型のオブジェクトをDATE型に、［7-4-10］のようにDATE型のオブジェクトをDATETIME型オブジェクトに変換することも可能です。なお、TIME関数は「TIME "HH:MM:SS"」の形式でTIME型のオブジェクトを取得できます。

7-4-9 DATE型のオブジェクトを作成する

記述例	戻り値
DATE(2020,7,12)	2020-07-12
DATE(DATETIME "2020-07-12 15:00:00")	

7-4-10 DATETIME型のオブジェクトを作成する

記述例	戻り値
DATETIME(2020,7,12,15,0,0)	2020-07-12 T15:00:00
DATETIME(DATE "2020-07-12", TIME "15:00:00")	

日付・時刻の差分を取得する関数

2つの日付・時刻の差分は、DATE_DIFF関数（デイト・ディフ）とDATETIME_DIFF関数（デイトタイム・ディフ）で取得できます。それぞれの構文は［7-4-11］と［7-4-12］の通りです。

7-4-11 DATE_DIFF関数の構文

```
DATE_DIFF(DATE型の新しい日付, DATE型の古い日付, デイトパート)
```

7-4-12 DATETIME_DIFF関数の構文

```
DATETIME_DIFF(DATETIME型の新しい日時,
              DATETIME型の古い日時, パート)
```

［デイトパート］と［パート］に利用できるキーワードは、DATE_ADD関数やDATETIME_ADD関数と同じです。例えば、差分の日数は「DAY」、時間は「HOUR」と指定します。以下のSQL文［7-4-13］では、2020年7月12日が令和になってから何日経過したのか、2020年7月12日 15:00:00が令和になって何時間経過したのかを取得します。また、DATE関数とDATETIME関数で、文字列をDATE型の日付とDATETIME型の日時に変換しています。

7-4-13 2つの日付・時刻から経過日数・時間を取得する

```
SELECT DATE_DIFF(DATE "2020-07-12",
DATE "2019-05-01", DAY)
AS reiwa_days
```

```
4   , DATETIME_DIFF(DATETIME "2020-07-12 15:00:00",
5   DATETIME "2019-05-01 00:00:00", HOUR)
6   AS reiwa_hours
```

結果テーブル

行	reiwa_days	reiwa_hours
1	438	10527

2つの日時から「日」と「時」の差分を取得できた

日付・時刻を丸める関数

日付や時刻は「秒→分→時→日→月→四半期→年」という階層構造になっているので、細かい単位をより大きい単位に丸められます。例えば「2020年7月12日 15:21:42」を徐々に丸めると、次のようになります。

- ●分　⇒2020年7月12日 15:21:00
- ●時　⇒2020年7月12日 15:00:00
- ●日　⇒2020年7月12日
- ●月　⇒2020年7月1日
- ●四半期　⇒2020年第三四半期
- ●年　⇒2020年

上記をDATE型の値に対して行うのが**DATE_TRUNC関数**（デイト・トランク）、DATETIME型の値に対して行うのが**DATETIME_TRUNC関数**（デイトタイム・トランク）です。TRUNCは「truncate」（切り詰める、切り捨てる）に由来します。構文はそれぞれ［7-4-14］と［7-4-15］となります。

7-4-14 DATE_TRUNC関数の構文

```
1   DATE_TRUNC(DATE型の値, デイトパート)
```

7-4-15 DATETIME_TRUNC関数の構文

```
1  DATETIME_TRUNC(DATETIME型の値, パート)
```

［デイトパート］と［パート］に利用できるキーワードは、DATE_ADD関数やDATETIME_ADD関数と同じです。

これらの関数を利用して、日単位や時間単位などの細かい粒度で記録された販売データを、月単位や年単位に丸めるニーズは非常に高く、頻繁に利用します。例えば、以下の［7-4-16］のような販売日時テーブルが存在し、日時ごとの販売個数が記録されていたとします。

このテーブル名を［small_ordertime］とし、「月ごとの販売個数」を集計するSQL文は以下の［7-4-17］となります。［order_time］フィールドの粒度がどんなに細かくても、希望する粒度（ここでは「月」）で丸めて集計することが可能です。

7-4-16 販売日時テーブル

order_time	qty
2020-01-11 20:18:12	6
2020-01-18 05:11:10	2
2020-01-25 17:07:12	4
2020-02-02 09:00:05	9
2020-02-25 10:47:59	5
2020-02-28 22:48:06	6

7-4-17 日単位を月単位に丸める

```
1  SELECT
2  DATETIME_TRUNC(order_time, MONTH) AS year_month
```

```
3   , SUM(qty) AS sum_qty
4   FROM sample.small_ordertime
5   GROUP BY year_month
6   ORDER BY 1
```

結果テーブル

行	year_month	sum_qty
1	2020-01-01T00:00:00	12
2	2020-02-01T00:00:00	20

「分単位」のデータを「月単位」でまとめて集計できた

日付の属性を取得するEXTRACT関数

日付は「年」「月」「年の中で何週目か？」「日」「年の中で何日目か？」「曜日」などの属性を持ちます。長期的なデータ分析において、日付の属性でグループ化する考え方は重要です。その属性を抜き出すのがEXTRACT関数（エクストラクト）です。構文は以下の［7-4-18］となります。

7-4-18 EXTRACT関数の構文

```
1   EXTRACT(日付の属性 FROM DATE型の値)
```

［日付の属性］に指定できるキーワードは次の通りです。EXTRACT関数の戻り値は、すべて整数型（INT64）となります。この関数はDATE型の値にのみ対応しており、DATETIME型には対応していません。

- YEAR　⇒年
- QUARTER　⇒四半期を表す整数（1月～3月が「1」、10月～12月が「4」）
- MONTH　⇒月
- WEEK　⇒年の中の何週目かを表す整数

- DAYOFYEAR ⇒年の中の何日目かを表す整数
- DAY ⇒日
- DAYOFWEEK ⇒日曜日を「1」、土曜日を「7」とする曜日を表す整数

具体例で見ていきましょう。以下の［7-4-19］は日別の販売個数がまとまった販売日付テーブルで、2020年1月1日～7日、2020年2月1日～7日の日付（order_date）で14レコードが格納されています。

このテーブル名を［small_orderdate］と仮定し、EXTRACT関数を使って日付の属性としての「日」を抽出して「日別の販売個数」を求めてみたいと思います。SQL文は［7-4-20］の通りです。結果は、元のテーブルの半数となる7レコードになりました。

これは、異なる月でも同じ日付が「日」の属性でグループ化されたためです。例えば、1月1日と2月1日は同じ「1」でグループ化されます。「1か月のうち、月初の販売個数が少ないのではないか？」など、月をまたがって検証したいときに非常に便利です。

7-4-19 販売日付テーブル

order_date	qty
2020-01-01	6
2020-01-02	2
2020-01-03	4
2020-01-04	6
2020-01-05	2
2020-01-06	8
2020-01-07	6
2020-02-01	5
2020-02-02	9
2020-02-03	5
2020-02-04	6
2020-02-05	2
2020-02-06	4
2020-02-07	6

7-4-20 「日」の属性で販売個数を集計する

```
SELECT EXTRACT(DAY FROM order_date) AS day
, SUM(qty) AS sum_qty
FROM sample.small_orderdate
GROUP BY day
ORDER BY 1
```

結果テーブル

行	day	sum_qty
1	1	11
2	2	11
3	3	9
4	4	12
5	5	4
6	6	12
7	7	12

「月」を問わず、「日」ごとに販売個数を集計できた

日付・時刻の表示形式を変更する関数

例えば「2020/7/18」という日付を、月を最初にして「7/18/2020」や「July 18, 2020」と表示したいこともあります。BigQueryには、元のデータ型を変えずに、DATE型やDATETIME型のまま、その表示形式を柔軟に変える関数が用意されています。それが**FORMAT_DATE関数**（フォーマット・デイト）と**FORMAT_DATETIME関数**（フォーマット・デイトタイム）で、構文は［7-4-21］と［7-4-22］の通りです。

7-4-21 FORMAT_DATE関数の構文

```
FORMAT_DATE(表示形式を指定するキーワード, DATE型の値)
```

7-4-22 FORMAT_DATETIME関数の構文

```
1  FORMAT_DATETIME(表示形式を指定するキーワード,
2                  DATETIME型の値)
```

引数の［表示形式を指定するキーワード］のうち、利用頻度の高いものをまとめたのが以下の表［7-4-23］です。

7-4-23 日付・時刻の表示形式を指定するキーワード

キーワード	表示形式	戻り値の例
%A	完全な曜日名	Monday
%a	省略された曜日名	Mon
%B	完全な月の名前	January
%b	省略された月の名前	Jan
%c	日付および時刻の表記 *	Sat Jan 18 05:11:10 2020
%d	10進数として表示される月内の日付	09（一桁台の場合は先頭に「0」）
%e	10進数として表示される月内の日付	9（一桁台の場合は先頭にスペース）
%F	「%Y-%m-%d」形式の日付	2020-07-13
%H	10進数として表示される時間 *	09
%M	10進数として表示される分 *	55
%m	10進数として表示される月	07
%S	10進数として表示される秒 *	21
%T, %X	「%H:%M:%S」形式の時刻 *	05:11:10
%Y	10進数として表示される世紀を含む年	2020

※FORMAT_DATE関数では利用不可

これらのキーワードには、①複数の表示形式を同時に利用できる、②スペース、「-」(ハイフン)、「,」(カンマ) などの記号を含められるという性質もあります。例えば「%F %T」とすると、「YYYY-MM-DD」と「HH:MM:SS」の表示形式をスペースで連結できます。実際の記述例については、以下のSQL文 [7-4-24] を参照してください。[7-4-16] として提示した販売日時テーブルを対象に、[order_time] フィールドをさまざまな表示形式に変更しています。

なお、FORMAT_DATE関数やFORMAT_DATETIME関数の結果はSTRING型で返されるので、DATE型やDATETIME型への変換は、CAST関数を利用してください (SECTION 7-1を参照)。

7-4-24 日付・時刻の表示形式を指定する

```
SELECT order_time AS original
, FORMAT_DATETIME("%A", order_time) AS upper_A
, FORMAT_DATETIME("%a", order_time) AS lower_a
, FORMAT_DATETIME("%B", order_time) AS upper_B
, FORMAT_DATETIME("%b", order_time) AS lower_b
, FORMAT_DATETIME("%D", order_time) AS upper_D
, FORMAT_DATETIME("%c", order_time) AS lower_c
, FORMAT_DATETIME("%F", order_time) AS upper_F
, FORMAT_DATETIME("%T", order_time) AS upper_T
FROM sample.small_ordertime
LIMIT 3
```

行	original	upper_A	lower_a	upper_B	lower_b	upper_D	lower_c	upper_F	upper_T
1	2020-01-18T05:11:10	Saturday	Sat	January	Jan	01/18/20	Sat Jan 18 05:11:10 2020	2020-01-18	05:11:10
2	2020-01-25T17:07:12	Saturday	Sat	January	Jan	01/25/20	Sat Jan 25 17:07:12 2020	2020-01-25	17:07:12
3	2020-02-25T10:47:59	Tuesday	Tue	February	Feb	02/25/20	Tue Feb 25 10:47:59 2020	2020-02-25	10:47:59

日付・時刻のデータをさまざまな表示形式に変更できた

STEP UP

算術演算子でも「日」の足し引きが可能

DATE型のフィールドの値への「日」の足し引きには、算術演算子を利用することもできます。例えば、2020年11月1日に「7日」を足したり引いたりするSQL文は、以下の［7-4-25］となります。見ての通り、直感的に記述できるので、おすすめの方法です。

また、この算術演算子による「日」の足し引きは、CURRENT_DATE関数やDATE_TRUNC関数に対する戻り値としての日付に対しても利用できます。［7-4-26］は、それが検証できるSQL文です。

7-4-25 算術演算子で「日」を足し引きする

```
SELECT DATE "2020-11-01" + 7 AS plus_7_days
, DATE "2020-11-01" - 7 AS minus_7_days
```

7-4-26 関数の戻り値に対して「日」を足し引きする

```
SELECT CURRENT_DATE("Asia/Tokyo") AS today
, CURRENT_DATE("Asia/Tokyo") + 7 AS plus_7_days
, CURRENT_DATE("Asia/Tokyo") - 7 AS minus_7_days
, DATE_TRUNC(DATE "2020-11-15", MONTH) + 7
AS plus_7_from_trancated_date
, DATE_TRUNC(DATE "2020-11-15", MONTH) - 7
AS minus_7_from_trancated_date
```

SECTION

7-5 統計集計関数

本章もこれで最後です。ここでは統計集計関数として、分散や標準偏差を求める関数を学んでいきます。

関数も、もうお腹いっぱいですね。仕上げのつもりで学習します！

まだ次章で「ウィンドウ関数」を学びますが、SQLで一般的に覚えるべき関数は、これで完了ですね。

分散、標準偏差、相関係数をSQLで取得できる

統計集計関数とは、統計学的な指標の取得に便利な関数です。分析の目的により、利用する関数が異なります。

- ●分散を求める ⇒VAR_POP関数／VAR_SAMP関数
- ●標準偏差を求める ⇒STDDEV_POP関数／STDDEV_SAMP関数
- ●2変数の相関係数を求める ⇒CORR関数

「VAR」と「STDDEV」の後ろに続く「POP」と「SAMP」は、分散や標準偏差を求める対象とするデータが「全数」か「標本」(サンプル）かの違いで使い分けます。統計学的に表現すると、「VAR_POP」は分散、「VAR_SAMP」は不偏分散、「STDDEV_POP」は分散に基づく標準偏差、「STDDEV_SAMP」は不偏分散に基づく標準偏差です。

標本のサイズ（＝レコード数）が多いときは、どちらの関数で計算してもほとんど同じ戻り値が得られるので、どちらを使うか気にする必要はありませんが、データサイズが小さい場合には慎重に選択しましょう。

分散を求めるVAR_POP/VAR_SAMP関数

VAR_POP関数とVAR_SAMP関数は、それぞれ分散と不偏分散を求める関数です。分散とはデータの散らばり度合いを示す指標で、値が大きいほど散らばりが大きいと判断できます。

ここでは、データが「全数」のときを想定し、VAR_POP関数の利用例を紹介しますが、対象のデータが「標本」(サンプル）の場合はVAR_SAMP関数を利用してください。指定する引数はどちらも同じです。

以下のSQL文［7-5-1］は、以前に［6-1-4］として提示した3年間の都道府県別最低賃金テーブルを対象に、平成29年、平成30年、令和元年の最低賃金の分散を求めています。結果テーブルを見ると、年を追うごとに都道府県別の最低賃金の散らばり度合いが大きくなっていることが確認できます。

7-5-1 分散を求める

```
1  SELECT year, VAR_POP(min_wage) AS variance
2  FROM sample.small_min_wage
3  GROUP BY year
```

結果テーブル

行	year	variance
1	H29	3707.5555555555557
2	H30	3800.0
3	R1	3894.2222222222226

VAR_POP 関数により分散が求められた

標準偏差を求めるSTDDEV_POP/STDDEV_SAMP関数

STDDEV_POP関数は分散に基づく標準偏差、STDDEV_SAMP関数は不偏分散に基づく標準偏差を求める関数です。標準偏差は英語で「standard deviation」なので、関数名の由来といえます。

対象のデータが「全数」の場合はSTDDEV_POP関数、「標本」（サンプル）の場合はSTDDEV_SAMP関数を利用してください。指定する引数はどちらも同じです。

なお、標準偏差は分散の平方根で求めます。各レコードの値と平均の差の2乗を平均して求める分散には単位がないのに対し、標準偏差は平方根をとることで、元のデータの単位にそろえることが可能です。つまり、元のデータが年収であれば標準偏差の単位は「円」、テストの点数であれば「点」と表現できます。

STDDEV_POP関数の具体的な記述方法は、以下の［7-5-2］の通りです。先ほどの例と同じテーブルを対象に、平成29年から令和元年の3年間の各年の最低賃金の標準偏差を求めています。分散が示していたのと同様に、年を追うごとに散らばり度合いが大きくなることが分かります。

7-5-2 標準偏差を求める

```
1  SELECT year, STDDEV_POP(min_wage) AS standard_deviation
2  FROM sample.small_min_wage
3  GROUP BY year
```

結果テーブル

行	year	standard_deviation
1	H29	60.889699913495676
2	H30	61.644140029689765
3	R0	62.403703593794994

STDDEV_POP 関数により標準偏差が求められた

2変数の相関係数を求めるCORR関数

CORR関数（コリレーション）は、2変数の相関係数を求めるために利用します。2変数とは「身長と体重」「数学と英語のテストの点」「気温とビールの販売量」「セッションとコンバージョン」など、「人」や「日」などを対象とした2種類の指標を指します。

片方が大きくなるともう一方も大きくなる関係を「正の相関がある」、片方が大きくなるともう一方が小さくなる関係を「負の相関がある」といい、それらの相関の方向と強さを表すのが相関係数です。

相関係数は「-1」から「1」までの値をとり、マイナスが負の相関、プラスが正の相関を表し、絶対値の大きさが相関の強さを表します。「0」付近は相関がない、もしくは非常に小さいことを示します。

例えば、時給が一定のアルバイト店員が働いた「時間と報酬」のように、散布図にすると完全に直線状にデータがプロットされる場合、2変数は完全相関する（この場合は正の完全相関）といいます。この相関係数を取得する関数がCORR関数です。相関を表す英語の「correlation」が由来と思われます。

具体的な利用例として、平成29年度の都道府県別推定人口（千人単位）と最低賃金が記録された以下の［7-5-3］のテーブルを対象にします。このテーブル名は［small_population］とします。

7-5-3 都道府県別の推定人口・最低賃金テーブル

pref	population	min_wage
宮城	2323	772
和歌山	945	777
山口	1383	777
新潟	2267	778
福井	779	778
⋮	⋮	⋮
愛媛	1364	739
島根	685	740
徳島	743	740
福島	1882	748
香川	967	766

2つの変数、この例では推定人口と最低賃金の相関係数をCORR関数で求めてみましょう。人口が多い都道府県ほど最低賃金が高い傾向にあれば「1」に近い値が、逆に人口が多い都道府県ほど最低賃金が低い傾向にあれば「-1」に近い値が、無関係であれば「0」に近い相関係数となるはずです。

SQL文と結果テーブルは、以下の［7-5-4］の通りとなります。相関係数が「0.88」もあるので、都道府県別の推定人口と最低賃金は、かなりの程度で相関しているといえるでしょう。そのようなことがCORR関数で一発で取得できるところが、この関数の醍醐味です。

なお、散布図を用いると、どの程度の相関があるのかを直観的に理解できて便利です。以下の［7-5-5］は、元のデータをGoogleデータポータルで散布図としてビジュアライズしています。

7-5-4 2変数の相関係数を求める

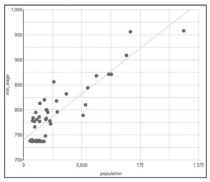
```
1  SELECT CORR(population, min_wage) AS correlation
2  FROM sample.small_population
```

結果テーブル

行	correlation
1	0.8843437698152642

CORR 関数により相関係数が求められた

7-5-5 データポータルで作成した散布図



縦軸を最低賃金、横軸を推定人口として散布図を作成した

STEP UP

偏差値を求める計算式

本節で学んだ標準偏差は、受験の指標となる偏差値の計算に用いられるので、なじみのある人も多いでしょう。偏差値を求める計算式は以下の通りです。

(得点 − 平均点) / 標準偏差 × 10 + 50

今、8人分のテスト「a」「b」の点数が記録されている［small_test］テーブルがあるとします。両方のテストともに平均点は「4.0」です。上記の計算式に基づき、各人の得点の偏差値を求めるSQL文は以下の［7-7-6］の通りです。同じ5点を取った人でも、「a」のテストなら偏差値「54」、「b」のテストなら偏差値「59」ということになります。

7-5-6 2つのテストでの偏差値を求める

```
SELECT name, score_a
, ROUND((score_a - (SELECT AVG(score_a)
FROM sample.small_test)) /
(SELECT STDDEV_SAMP(score_a)
FROM sample.small_test) * 10 + 50)
AS hensachi_a
, score_b
, ROUND((score_b - (SELECT AVG(score_b)
FROM sample.small_test)) /
(SELECT STDDEV_SAMP(score_b)
FROM sample.small_test) * 10 + 50)
```

```
12  AS hensachi_b
13  FROM sample.small_test
```

結果テーブル

行	name	score_a	hensachi_a	score_b	hensachi_b
1	Aさん	1	37.0	3	41.0
2	Bさん	1	37.0	3	41.0
3	Cさん	3	46.0	3	41.0
4	Dさん	3	46.0	3	41.0
5	Eさん	5	54.0	5	59.0
6	Fさん	5	54.0	5	59.0
7	Gさん	7	63.0	5	59.0
8	Hさん	7	63.0	5	59.0

計算式により偏差値が求められた

この例は8レコードしかないので臨場感に欠けますが、大人数が受験するテストのように、点数が正規分布に近い形で分布する場合、偏差値「60」以上は全体の約16%、偏差値「70」以上は全体の約2.5%しかいないので、偏差値により、その点数を取った人が全体で上位何%くらいなのかの目安になります。

ちなみに、標準偏差が示すデータの散らばり度合いは、A/Bテストで「どちらのパターンが勝者なのか？」「決定的に差があるといってよいのか？」を判断するときにも参照すべき指標です。

SECTION

7-6 確認ドリル

問題 021

[customers] テーブルの [name] フィールドには、氏名が半角スペース区切りで格納されています。[name] とは別に、姓を [family_name]、名を [first_name] として取り出してください。

結果テーブルは3カラムになります。全員分を確認する必要はないので、5名分だけ表示してください。

問題 022

[sales] テーブルを対象に、商品ID (product_id) ごとの販売金額 (revenue) を合計して [sum_revenue] というフィールド名で取得してください。[sum_revenue] は百の位で四捨五入し、千円単位に丸めて整数にしてください。結果テーブルは [sum_revenue] の大きい順に5つに絞り込みます。

問題 023

[customers] テーブルから、50歳以上の顧客には名前 (name) の末尾に「さま」を、そうでない場合には「さん」を付けて、誕生日 (birthday) とともにリストにしてください。年齢は、2019年12月31日時点での満年齢とし、誕生日が「null」の人は除外します。

結果テーブルは、敬称付きの名前 (name_with_keisho) と年齢 (age)と、誕生日 (birthday) の3カラムとしてください。正しく敬称がつけられているかどうかを確認するために、名前に「美馬」もしくは「森村」を含む顧客に絞り込んでください。

問題 024

［web_log］テーブルから年月別のページビュー数を取得し、ページビュー数が多い順にトップ3の年月（year_month）と、同月のページビュー数（pageviews）を取り出してください。結果テーブルは、年月（year_month）とページビュー数（pageviews）の2カラムとなります。

問題 025

［customers］テーブルの［name］フィールドには、氏名が半角スペース区切りで格納されています。姓が「○川」(川の前に1文字存在し、「川」で終わる)、名が「○○子」(子の前に2文字存在し、「子」で終わる）に該当する女性について、フルネーム（name)、姓名の名（first_name）の2カラムを取得してください。正規表現を使わなくても記述できますが、できれば正規表現にチャレンジしてください。

解答

解答 021

```
SELECT name
, SUBSTR(name, 1, STRPOS(name, " ") - 1) AS family_name
, SUBSTR(name, STRPOS(name, " ") + 1) AS first_name
FROM sample.customers
LIMIT 5
```

結果テーブル

行	name	family_name	first_name
1	石塚 拓	石塚	拓
2	長坂 賢介	長坂	賢介
3	野中 裕之	野中	裕之
4	髙松 龍	髙松	龍
5	小倉 慎太郎	小倉	慎太郎

STRPOS関数を使って姓と名を区切る半角スペースの位置を見つけ、SUBSTR関数で姓は「1文字目からスペースの手前まで」、名は「スペースの次の文字から文字列の最後まで」を取得します。

なお、正規表現を使った別解は以下の通りです。

```
SELECT name
, REGEXP_EXTRACT(name, r"^([^\s]+)") AS family_name
, REGEXP_EXTRACT(name, r"\s(.+)") AS first_name
FROM sample.customers
LIMIT 5
```

解答 022

```
SELECT product_id
, CAST(ROUND(SUM(revenue), -3) AS INT64)
AS sum_revenue
FROM sample.sales
GROUP BY product_id
ORDER BY 2 DESC
LIMIT 5
```

結果テーブル

行	product_id	sum_revenue
1	1	763000
2	2	547000
3	15	497000
4	5	240000
5	14	210000

2行目がポイントです。SUM関数で［revenue］を合計し、ROUND関数で百の位で四捨五入します。戻り値はFLOAT64型となり、小数点以下第一位に「0」が残ってしまため、CAST関数で整数にデータ型を変換しています。

解答 023

```
SELECT CONCAT(name, keisho) AS name_with_keisho
, birthday
, age
FROM (
SELECT name, birthday
, DATE_DIFF(DATE "2019-12-31", birthday, YEAR) AS age
, IF(DATE_DIFF(DATE "2019-12-31", birthday, YEAR) >= 50
```

```
    , "さま", "さん") AS keisho
    FROM sample.customers
    WHERE birthday IS NOT NULL
    )
    WHERE REGEXP_CONTAINS(name, "美馬|森村") IS TRUE
    ORDER BY birthday
```

結果テーブル

行	name_with_keisho	birthday	age
1	美馬 素美さま	1969-05-09	50
2	森村 陽さん	1970-08-29	49

DATE_DIFF関数とCONCAT関数、REGEXP_CONTAINS関数の3つを使います。DATE_DIFF関数では、満年齢を決める2019年12月31日から、誕生日を引いた結果を「年」で表示することで、年齢を取得しています。

7〜8行目のIF文では「年齢が50以上か、そうでないか」を条件に、「さま」か「さん」かを[keisho]として取り出します。最後に、メインのクエリで[name]と［keisho］を連結して［name_with_keisyo］とし、[birthday][age]とともに出力しています。

解答 024

```
SELECT
DATETIME_TRUNC(date_time, MONTH) AS year_month
, COUNT(page) AS pageviews
FROM sample.web_log
GROUP BY year_month
ORDER BY 2 DESC
LIMIT 3
```

結果テーブル

行	year_month	pageviews
1	2018-01-01T00:00:00	252
2	2017-12-01T00:00:00	199
3	2019-01-01T00:00:00	175

年月日時分秒が記録されているDATETIME型の［date_time］カラムを、DATETIME_TRUNC関数を使って「年月」で丸め、丸めた値でグループ化しています。

なお、[date_time］カラムを、FORMAT_DATETIME関数を使って「年月」で丸める以下の別解も考えられます。

```
SELECT
FORMAT_DATETIME("%Y-%m", date_time) AS year_month
, COUNT(page) AS pageviews
FROM sample.web_log
GROUP BY year_month
ORDER BY 2 DESC
LIMIT 3
```

ページビュー数を取得するのに「COUNT(page)」を利用しています。この意味合いは「ユニークでない」ページのカウントです。

「年月」で丸める場合、同じWebページ（page）に何回もアクセスされることが考えられます。仮に「COUNT(DISTINCT page)」とすると、重複が除外されて、この問題の意図から外れてしまいます。COUNT関数で「DISTINCT」の使用については、よく考慮してください。

なお、[page］カラムに「null」がない、もしくは［page］が「null」であればページビューにカウントしない、という条件があれば、「COUNT(page)」ではなく「COUNT(*)」でグループ化された行数をカウントすることでも正解を導くことができます。

年月日を「丸める」ケースは非常に多く、このような問題を解くと理解が深まるのではないかと思います。このテクニックは日ごとに丸めてグループ化する、曜日ごとに丸めてグループ化するなど、広く応用が利きます。

解答 025

```
SELECT name, first_name
FROM (
SELECT name
, REGEXP_EXTRACT(name, r"^.川\s(..子)$") AS first_name
FROM sample.customers
WHERE gender = 2
)
WHERE first_name IS NOT NULL
```

結果テーブル

行	name	first_name
1	堀川 華奈子	華奈子
2	緑川 美輝子	美輝子
3	小川 美和子	美和子

正規表現を使う場合は、REGEXP_EXTRACT関数を利用し、[name] フィールドから該当する文字列を直接［first_name］として取得します。

上記SQL文で利用している正規表現は以下の通りです。

^　⇒次の文字で始まる
.　⇒任意の１文字
\s　⇒半角スペース
()　⇒グループ。囲まれた部分に該当する文字列を取得する
$　⇒直前の文字で終わる

前ページのSQL文の4行目を丁寧に読み解くと、題意に沿った正規表現が記述されていることが理解できると思います。

なお、正規表現を利用しない場合、WHERE句でLIKE演算子を使って、姓が「○川」、名が「○○子」のレコードに絞り込みます。以下が別解です。

［name］フィールドに格納されている氏名は半角スペースで区切られていることが分かっているので、STRPOS関数を使って半角スペースが出現する位置を取得し、SUBSTR関数でその位置を利用して姓名を別個に取り出します。

```
SELECT name, first_name
FROM (
SELECT name
, SUBSTR(name, 1, STRPOS(name, " ") - 1) AS family_name
, SUBSTR(name, STRPOS(name, " ") + 1) AS first_name
FROM sample.customers
WHERE gender = 2
)
WHERE family_name LIKE ("_川")
AND first_name LIKE ("__子")
```

CHAPTER 8

ウィンドウ関数

8-1　ウィンドウ関数とは
8-2　ランキング・行番号の取得
8-3　最初の値・別レコードの値の取得
8-4　累計・移動平均の取得
8-5　確認ドリル

次章は問題のみの演習ドリルとなるので、本章が実質的な最終章です。ここで学ぶ「ウィンドウ関数」は、CHAPTER 4で身につけた集計関数、CHAPTER 7で見てきた数値や文字列、日付・時刻を扱う関数よりも少々複雑ではありますが、SQLでの分析力を飛躍的に高めてくれます。柔軟で強力な機能をマスターしてください。

SECTION

8-1 ウィンドウ関数とは

それでは「ウィンドウ関数」について学んでいきましょう。別名「分析関数」「OLAP関数」とも呼びます。

関数はたくさん学んできましたが、もっとほかのことができるのでしょうか？

例えば、販売金額について各月のベスト3の商品や、ランディングページ別の直帰率を取り出せます。分析の幅が大きく広がりますよ。

複数のレコードの値を保持しながら集計できる

「ウィンドウ関数」とは、任意のフィールドに基づき、指定したグループに対して元のデータを保持しながら集計を行う関数です。ウィンドウ関数において、グループ化する範囲のことを「パーティション」と呼びます。このパーティションに含まれる**複数のレコードが持つ値を単純に集計せず、各レコードの値を保持しながら集計できる**ことが特徴です。

集計関数とウィンドウ関数の違いを見るために、次ページの2つの図［8-1-1］と［8-1-2］を示します。いずれもMAX関数によって最大値を求める例で、MAX関数は集計関数としてすでに解説していますが、ウィンドウ関数（集計分析関数）として利用することもできます。

［8-1-1］は、集計関数としてのMAX関数の処理です。[user_id]でグループ化し、「MAX(qty)」で集計すると、各レコードの値は集計され、1つの最大値にまとめられてユニークな［user_id］の数だけが結果テーブルに残ります。

このとき、各レコードが保持していた［qty］の値の情報は完全に消失し、

結果テーブルには反映されません。例えば［user_id］が2種類なら、2レコードになります。

［8-1-2］は、ウィンドウ関数としてのMAX関数の処理です。「MAX(qty)」によって、［user_id］を基準としてパーティションを区切る（グループ化する）ように指定しています。

すると、［user_id］が「1」の［qty］の値について、各レコードの「5」「7」「3」は保持されたまま、最大値である「7」が結果テーブルの1〜3行目すべてに記録されます。［user_id］が「2」についても同様に処理されて、元のテーブルと結果テーブルのレコード数は変わりません。

8-1-1 集計関数としてのMAX関数の処理

user_id	date	qty
1	1/1	5
1	1/10	7
1	1/12	3
2	1/3	2
2	1/11	4

集計関数
MAX(qty)

user_id	max_qty
1	7
2	4

元のレコードの値は結果テーブルに残らない

8-1-2 ウィンドウ関数としてのMAX関数の処理

user_id	date	qty
1	1/1	5
1	1/10	7
1	1/12	3
2	1/3	2
2	1/11	4

ウィンドウ関数
MAX(qty)

user_id	qty	max_qty
1	5	7
1	7	7
1	3	7
2	2	4
2	4	4

元のレコードの値が結果テーブルに残る

上記が「複数のレコードが持つ値を単純に集計せず、各レコードの値を保持しながら集計できる」という、ウインドウ関数のもっとも基本的な特徴についての図解です。

続いて、「結果テーブルに元のレコードの値が残る」というウィンドウ関数の特徴が、どう有効に働くのかを例で紹介しましょう。以下の［8-1-3］は、代表的なウィンドウ関数であるRANK関数の処理を図示したものです。

RANK関数のパーティションとして［user_id］を指定してユーザーごとのランク（順位）を取得し、結果テーブルの［rank］フィールドに格納しています。[user_id］の元のレコードの値が参照できるため、ユーザーごとのランクが一目瞭然であることが分かると思います。

8-1-3 RANK関数の処理

user_id	qty	revenue
1	5	2000
1	7	3500
1	3	900
2	2	800
2	4	2000

ウィンドウ関数
RANK(qty)

user_id	qty	revenue	rank
1	5	2000	2
1	7	3500	1
1	3	900	3
2	2	800	2
2	4	2000	1

元のレコードの値を残しながらランクを取得できる

このような、通常の演算や関数では実現できない処理を可能にするのが、ウィンドウ関数です。ウィンドウ関数には多くの関数がありますが、本書では利用頻度の高い11の関数に絞って解説していきます。

また、ウィンドウ関数は処理の内容によって、大きく3つの種類に分かれます。11の関数とウィンドウ関数としての処理、種類を次ページの表［8-1-4］にまとめたので、およその全体像を理解してください。

表内の「処理」の列は、すべて「任意に設定したパーティション内の」を文頭に追加して解釈してください。つまり、RANK関数の「ランクを取得する」は、「任意に設定したパーティション内の」「ランクを取得する」という意味です。

8-1-4 代表的なウィンドウ関数と処理・種類

関数	処理	種類
RANK	ランクを取得する	番号付け関数
ROW_NUMBER	行番号を取得する	
NTILE	均等な数に分割する	
FIRST_VALUE	最初の値を取得する	ナビゲーション関数
LAST_VALUE	最後の値を取得する	
NTH_VALUE	任意の順番の値を取得する	
LEAD	直後の値を取得する	
LAG	直前の値を取得する	
PERCENTILE_CONT	五数要約の指標を取得する	
SUM	累計を取得する	集計分析関数
AVG	移動平均を取得する	

ウィンドウ関数は「分析関数」とも呼ばれる通り、SQLを駆使したビジネスデータの分析に欠かせない機能です。どのような利用シーンがあるのか、番号付け関数、ナビゲーション関数、集計分析関数の3つの種類について、それぞれの典型例を次にまとめます。

ただ、実際には幅広い利用シーンがあるので、本章の確認ドリルや、次章の演習ドリルにもチャレンジして理解を深めてください。

▶番号付け関数

●ランキングによる人気商品分析

⇒人気商品の分析は販売金額の多寡で行うことが多いですが、RANK関数を利用すると、顧客ひとりひとりの商品別購入金額ランキングによって人気商品を分析できます。

●ゴールデン導線分析
⇒ROW_NUMBER関数を利用すると、「あるユーザーがページAにランディングし、次にページB→ページCと遷移した後にコンバージョンした」といった分析ができます。

▶ナビゲーション関数

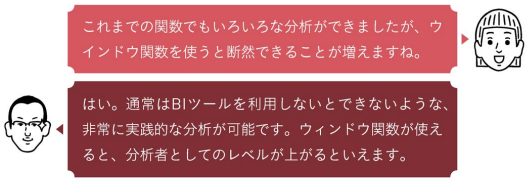
●CRM分析
⇒FIRST_VALUE関数を利用すると、「初回注文の購入金額が、その顧客にとっての複数回の注文の中で最大額だった顧客は、そうでない顧客に比べてLTVの増加ペースが早いのではないか？」といった詳細な顧客分析ができます。

●Web解析
⇒LEAD関数を使うと、Webサイトのログデータにおいてヒットのタイムスタンプしか記録されていない状態から、前後の差分によるページ滞在時間を取得できます。

▶集計分析関数

●累計の取得
⇒SUM関数を集計分析関数として利用することで、累計を取得できます。

●移動平均の取得
⇒AVG関数を集計分析関数として利用することで、移動平均を取得できます。



ウィンドウ関数の基本構文と各句の意味

ウィンドウ関数の構文を見ていきます。これまでに学んだ関数に比べると複雑ですが、以下の［8-1-5］を基本構文として覚えてください。

8-1-5 ウィンドウ関数の基本構文

```
1  関数名 OVER (
2  PARTITION BY パーティションとして定義するフィールド名
3  ORDER BY パーティション内での並べ替えの基準とするフィールド名
4  WINDOWフレーム
5  )
```

関数名に続く「OVER（オーバー）」は必須です。関数の種類によっては「PARTITION BY（パーティション バイ）」「ORDER BY（オーダー バイ）」とそれに続く指定、および「WINDOWフレーム（ウィンドウ）」は「必須」「省略が可能」、あるいは「使えない」ことがあります。

それぞれの句の詳細は次の通りです。

▶OVER

OVER句は、ウィンドウ関数の利用を宣言する句なので、必ず記述します。**どのウィンドウ関数を使うときでも指定する決まり文句**と考えてください。OVERの後ろは半角カッコ「()」で囲み、OVER句で指定する内容を記述します。

▶PARTITION BY

PARTITION BY句は、パーティションして定義するフィールド名を指定します。省略するとテーブル全体をパーティションとして扱います。

例えば、RANK関数について考えてみましょう。次ページの［8-1-6］で示すように、各レコードに付与されるランクは「ユーザー」「都道府県」など、グループ化する範囲によって異なります。この範囲＝パーティションを定義す

るのがPARTITION BY句です。

パーティションとしてユーザーIDを指定し、商品ID別の販売金額ランクを取り出した場合、ユーザー「A」について商品「3」のランクは「1」、商品「1」のランクは「2」といった数値が格納されます。ユーザー「B」に切り替われば、そのユーザーにとっての商品ID別のランクが取得されます。

8-1-6 パーティションの定義による処理の違い

ユーザーごと

ユーザーID	商品ID	販売金額	販売ランク
A	3	3000	1
A	1	2750	2
A	2	1000	3
B	1	3800	1
B	3	3300	2

パーティション＝ユーザー ID。ユーザー ID が異なるとランクがリセットされる

都道府県ごと

都道府県	ユーザーID	年齢	年齢ランク
東京	254	68	1
東京	154	65	2
大阪	56	72	1
大阪	91	69	2

パーティション＝都道府県。都道府県が異なるとランクがリセットされる

▶ORDER BY

ORDER BY句は、パーティション内での並べ替えを行うときに指定します。基準とするフィールド名に続いて、昇順とする「ASC」、降順とする「DESC」のいずれかを記述してください。ASCはデフォルトとなるため省略が可能です。

例えば、RANK関数はパーティション内の上から順にランクを付けますが、そもそもの並べ替えはORDER BY句で制御しています。ASCとDESCを切り替えると、次ページの［8-1-7］ように結果が変わります。

8-1-7 パーティション内の並べ替えによる処理の違い

販売金額の降順

ユーザーID	商品ID	販売金額	販売ランク
A	3	3000	1
A	1	2750	2
A	2	1000	3
B	1	3800	1
B	3	3300	2

パーティション＝ユーザーIDとし、販売金額の降順で並べ替えている

販売金額の昇順

ユーザーID	商品ID	販売金額	販売ランク
A	2	1000	1
A	1	2750	2
A	3	3000	3
B	3	3300	1
B	1	3800	2

パーティション＝ユーザーIDとし、販売金額の昇順で並べ替えている

▶WINDOWフレーム

WINDOWフレーム句は、PARTITION BY句で定義したパーティション内で、さらに「どのレコードを使うか？」を「フレーム」として指定します。

AVG関数を使って移動平均を求める例で、WINDOWフレーム句の理解を深めていきましょう。以下の［8-1-8］を見てください。このテーブルではPARTITION BY句で［year］、ORDER BY句で［year_month］を指定し、「当月の値」「1か月前の値」「2か月前の値」を対象とした3か月の移動平均を取得しようとしています。

8-1-8 移動平均を求めるテーブル

行番号	year	year_month	revenue	3カ月移動平均
1	2020	2020-01	2000	A
2	2020	2020-02	3000	B
3	2020	2020-03	4000	C
4	2020	2020-04	4000	D
5	2020	2020-05	6000	E
6	2020	2020-06	3000	F

このとき、[3か月移動平均] フィールドの「A」～「F」には、どのような値が入るかを考えてみましょう。

●A：2000
⇒過去のレコードがないため、行番号1の値である「2000」が入ります。

●B：2500
⇒当月の値（行番号2の「3000」）と、1か月前の値（行番号1の「2000」の平均値「2500」(3000＋2000 / 2）が入ります。

●C：3000
⇒当月の値（行番号3の「4000」）、1か月前の値（行番号2の「3000」）、2か月前の値（行番号1の「2000」）の平均値「3000」(4000＋3000＋2000 / 3）が入ります。

●D：3666.666…
⇒当月の値（行番号4の「4000」）、1か月前の値（行番号3の「4000」）、2か月前の値（行番号2の「3000」）の平均値「3666.666…」(4000＋4000＋2000 / 3）が入ります。

「E」「F」も同様に、当月（自分のレコード）の値、1か月前（1つ前のレコード）の値、2か月前（2つ前のレコード）の値の3つが計算対象となります。つまり、移動平均を求める場合、AVG関数はパーティション内のすべてのレコードの値を使わず、一部のみを使うことになります。

このようにパーティション内の全部のレコードを使わない場合、どのレコードを利用するかを指定する必要があり、その役割を担うのがWINDOWフレーム句です。ここで指定するレコードは「フレーム」と呼びます。

フレームの上限・下限は、具体的には次ページの［8-1-9］に示した「**ROWS**（ロウズ） **BETWEEN**（ビトゥイーン）」句の構文に従って記述します。「ROWS」に続き、BETWEENとANDの2つの演算子で上限と下限のレコードを指定する形です。

8-1-9 ROWS BETWEEN句の構文

```
1  ROWS BETWEEN
2  パーティション内で利用するレコード（フレーム）の上限
3  AND
4  パーティション内で利用するレコード（フレーム）の下限
```

上記の構文において、[パーティション内で利用するレコード（フレーム）の上限］および［下限］を指定するキーワードには、次のようなものがあります。パーティション内のレコードの上限・下限（上端・下端）や、現在の行から○行分上まで・下までを指定する方法です。

UNBOUNDED＝際限なく、PRECEDING＝前に続く、FOLLOWING＝後に続くと、英語での意味を考えると分かりやすいでしょう。

- UNBOUNDED PRECEDING　⇒パーティションの上限
- ○ PRECEDING　⇒現在の行から○行だけ上
- CURRENT ROW　⇒現在の行
- ○ FOLLOWING　⇒現在の行から○行だけ下
- UNBOUNDED FOLLOWING　⇒パーティションの下限

ここで先ほどの［8-1-8］のテーブルに戻り、行番号3の「C」の値に注目してみましょう。フレームの上限・下限を指定するキーワードがどの行を指すのかを表すと、次のようになります。

- UNBOUNDED PRECEDING　⇒行番号1
- 2 PRECEDING　⇒行番号1
- CURRENT ROW　⇒行番号3
- 2 FOLLOWING　⇒行番号5
- UNBOUNDED FOLLOWING　⇒行番号6

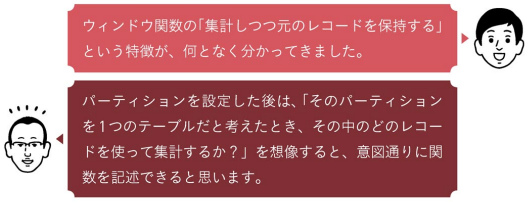
よって、当月を含む直近3か月間の移動平均を求めるのであれば、「2 PRECEDING」と「CURRENT ROW」をROWS BETWEEN句の構文に当てはめて、以下の［8-1-10］のように記述します。これをAVG関数のWINDOWフレーム句として指定すれば、意図通りの値を取得可能です。

8-1-10 前の2行～現在の行を表すROWS BETWEEN句

```
1  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

なお、WINDOWフレーム句は省略可能です。省略した場合、フレームの上限から現在のレコードまで、つまり「UNBOUNDED PRECEDING AND CURRENT ROW」を指定したことになります。

加えて、WINDOWフレーム句には、ROWSのほかに「RANGE」を利用した指定方法もあります。本書では理解しやすいROWSによる指定をメインに解説しますが、RANGEについても本節の末尾にあるSTEP UPで補足しています。あわせて参照してください。



指定できる「ウィンドウ」は関数によって異なる

ウィンドウ関数の構文にあるOVER句の中で、PARTITION BY、ORDER BY、およびWINDOWフレームで指定した内容を「ウィンドウ」と呼びます。このウィンドウは、すべてのウィンドウ関数で同じように指定できるわけではなく、利用する関数によって違いがあります。

詳細は以降で関数ごとに解説しますが、代表的なウィンドウ関数について、どのような違いがあるかを以下の表［8-1-11］にまとめました。おおまかなイメージを捉えておいてください。

8-1-11 ウィンドウ関数とウィンドウの対比

<table>
<tr><th>関数</th><th>PARTITION BY</th><th>ORDER BY</th><th>WINDOWフレーム</th></tr>
<tr><td>RANK</td><td rowspan="3">オプション</td><td>必須</td><td rowspan="3">利用不可</td></tr>
<tr><td>ROW_NUMBER</td><td>オプション</td></tr>
<tr><td>NTILE</td><td>必須</td></tr>
<tr><td>FIRST_VALUE</td><td rowspan="6">オプション</td><td rowspan="6">必須</td><td rowspan="3">オプション</td></tr>
<tr><td>LAST_VALUE</td></tr>
<tr><td>NTH_VALUE</td></tr>
<tr><td>LEAD</td><td rowspan="3">利用不可</td></tr>
<tr><td>PERCENTILE_CONT</td></tr>
<tr><td>LAG</td></tr>
<tr><td>SUM</td><td rowspan="2">オプション</td><td rowspan="2">オプション</td><td rowspan="2">オプション</td></tr>
<tr><td>AVG</td></tr>
</table>

STEP UP

WINDOWフレーム句のROWSとRANGEの違い

WINDOWフレーム句の指定方法には、本書で主に解説するROWS以外にも、RANGEを利用した方法があることを本節で述べました。

ROWSがOVER句のORDER BYで指定されたフレーム内の「行」を基準に関数を動作させるのに対して、RANGEはORDER BYで与えられたフレーム内の「値」を基準に関数を動作させます。

つまり、WINDOWフレーム句で「1 PRECEDING AND CURRENT ROW」が指定されていた場合も、ROWSでは文字通り「1行前と自分の行」がフレームとなるのに対し、RANGEでは「1つ前の値と自分の値」がフレームとなります。そのため、RANGEでフレームを指定する場合は、ORDER BYが数値で指定されている必要があります。

以下の［8-1-12］で示したSQL文と結果テーブルを見てください。仮想テーブルの1行目の日付が、2〜3行目の日付とは連続していないことが動作の違いになります。

8-1-12 ROWSとRANGEで移動平均を求める

```
WITH master AS
(SELECT date "2020-10-10" AS date, 1 AS qty
UNION ALL SELECT date "2020-11-25", 2
UNION ALL SELECT date "2020-11-26", 3 )

SELECT date, unix_date(date) AS unix_date, qty
, AVG(qty) OVER (
ORDER BY unix_date(date)
ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
) AS window_by_row
```

```
11    , AVG(qty) OVER (
12    ORDER BY UNIX_DATE(date)
13    RANGE BETWEEN 1 PRECEDING AND CURRENT ROW
14    ) AS window_by_range
15    FROM master
```

結果テーブル

行	date	unix_date	qty	window_by_rows	window_by_range
1	2020-10-10	18545	1	1.0	1.0
2	2020-11-25	18591	2	1.5	2.0
3	2020-11-26	18592	3	2.5	2.5

AVG 関数の WINDOW フレーム句を、ROWS と RANGE で指定した場合の値を比較している

［window_by_rows］フィールドは、SQL文の９行目の指定により、１レコード前と自分の行を指定した移動平均となっています。１レコード目は直前のレコードがないので「1.0」、２レコード目は（1＋2）÷2＝「1.5」、３レコード目は（2＋3）÷2＝「2.5」が計算内容です。

対して［window_by_range］フィールドは、SQL文の13行目の指定で『1 日前と自分のレコードの「日」』の移動平均が計算されます。１レコード目は直前の日がないので「1.0」、２レコード目も「１日前の日」がないので「2.0」、３ レコード目は11月26日に対する直前日＝11月25日が存在するので（2 ＋ 3）÷ 2 ＝「2.5」となります。

RANGEでフレームを指定する場合、OVER句のORDER BYは数値である必要があるので、UNIX_DATE(date)という関数で、日付型の［date］を「1970年１月１日からの経過日数」という数値に変換しています。

SECTION

8-2 ランキング・行番号の取得

ウィンドウ関数のうち「番号付け関数」を使った分析について、具体例とともに見ていきましょう。

どんな分析を想定していますか？

本節では、人気商品のランキングや、Webサイトにおける閲覧者のページ導線について分析してみます。

RANK関数による人気商品分析

テーブル全体での商品別の販売金額ランキングは、商品IDでグループ化し、SUM関数で販売金額を合計する単純な集計関数で取得できます。その結果、仮に商品ID「1」がランキング1位だったとしましょう。

このとき、全体として商品ID「1」がもっとも多く購入されたことは分かりますが、どの顧客も商品ID「1」をもっとも多く購入したかは、定かではありません。ごく一部の顧客が大量購入しただけで、ほかの顧客はそれほど商品ID「1」を購入していない可能性もあるからです。

一方、もっとも多く購入された商品を顧客ごとに求め、「何人の顧客がもっとも多く購入した商品として選んだか？」が分かれば、本当の人気商品が何であったかが分かるでしょう。このような分析をしたいときに使うのが、ウィンドウ関数（番号付け関数）の1つである**RANK（ランク）関数**です。

次ページの［8-2-1］では、100人の顧客がいるなかで、ランキング1位の商品が「ケーキ」の顧客は58人、「クッキー」は26人、「ゼリー」は16人いることが分かります。総額からではなく顧客ごとのランキングから、本当の人気商品が「ケーキ」であることが分かります。

8-2-1 人気商品ランキングの例

販売金額ランキング	商品	販売個数
1	ケーキ	58
1	クッキー	26
1	ゼリー	16

それでは、RANK関数の使い方を見ていきましょう。まずは以下の[8-2-2]に示した構文を確認してください。

RANK関数は、指定したパーティション内を並べ替えの基準・順序に従って「1、2、3、…」と値を付与します。1人の顧客が商品「A」「B」「C」「D」を、それぞれ「4000円」「2000円」「2000円」「1000円」と購入した場合のように、同一の値が存在する場合は「1、2、2、4」のようにランクをスキップします。

8-2-2 RANK関数の構文

```
1  RANK() OVER (
2  PARTITION BY パーティションとして定義するフィールド名
3  ORDER BY パーティション内での並べ替えの基準とするフィールド名
4  )
```

関数名の後の（）	何も記述しない
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	利用不可
戻り値のデータ型	整数型

PARTITION BY句のフィールド名に迷ったら「何ごとにランキングを付けたいのか？」を考えましょう。

続いて、RANK関数を使い、[8-2-1]と同様の結果テーブルを得るためのSQL文を考えてみましょう。以下の[8-2-3]には、RANK関数が7行目に記述されています。本章のSQL文はすべて演習用テーブルを対象としているので、みなさんもBigQueryで実行してみてください。

WITH句では、[sales]テーブルの顧客（user_id）ごとに、商品（product_id）別の購入金額（revenue）をSUM関数で集計しています。

また、今は顧客ごとにランキングを表示したいので、8行目のPARTITION BY句では［user_id］を指定しています。加えて、販売（購入）金額の大きい順に並べてランクを付与したいので、9行目のORDER BY句で［revenue］を指定し、オプションは「DESC」としました。

8-2-3 顧客別に商品ごとの販売金額ランキングを求める

```
1  WITH master AS (
2  SELECT user_id, product_id, SUM(revenue) AS sum_rev
3  FROM sample.sales
4  GROUP BY user_id, product_id
5  )
6  SELECT user_id, product_id, sum_rev
7  , RANK() OVER (
8  PARTITION BY user_id
9  ORDER BY sum_rev DESC
10 ) AS revenue_rank
11 FROM master
12 ORDER BY 1, 3 DESC
```

結果テーブル

行	user_id	product_id	sum_rev	revenue_rank
1	10059	1	4800	1
2	10059	2	2200	2
3	10059	14	1600	3
4	10060	14	1600	1
5	10060	9	960	2
6	10070	1	1600	1
7	10161	2	2200	1

[revenue_rank] フィールドで
ランキングの数値を取得できた

実際の結果テーブルは1000行以上ありますが、上記の画面に見える範囲では、3人の顧客別に［product_id］ごとの購入金額が高い順にランクが付与されています。[user_id] が「10059」の顧客であれば、[product_id] が「1」の商品の購入金額が1位となり、2位は商品「2」、3位は商品ID「14」という具合です。

ここに、さらに外側のクエリを足したのが［8-2-4］です（WITH句は［8-2-3］と同じなので省略）。外側というのは、具体的には1〜4行目、12行目〜14行目を指しています。

顧客にとって購入金額が1位（[revenue_rank] が「1」）の商品だけに絞り込み、[revenue_rank] と［product_id］でグループ化を行ったうえで顧客数、つまり、何人の顧客がその商品のランクを1位としたのかを「COUNT(DISTINCT user_id)」で取得しています。

結果、[product_id] が「1」の商品をランキング1位とした（もっとも多額に購入した）顧客が143人、「2」を1位とした顧客が92人、「15」を1位とした顧客が76人…と続き、人気商品を分析できました。

8-2-4 ランキング1位の商品の顧客数を求める

```
1  SELECT revenue_rank
2  , product_id
3  , COUNT(DISTINCT user_id) AS no_of_customer
4  FROM (
```

```
SELECT user_id, product_id, sum_rev
, RANK() OVER (
PARTITION BY user_id
ORDER BY sum_rev DESC
) AS revenue_rank
FROM master
)
WHERE revenue_rank = 1
GROUP BY revenue_rank, product_id
ORDER BY 3 DESC
```

結果テーブル

行	revenue_rank	product_id	no_of_customer
1	1	1	143
2	1	2	92
3	1	15	76
4	1	5	35
5	1	14	34
6	1	4	27
7	1	6	25
8	1	10	13
9	1	12	13
10	1	11	12
11	1	13	11

[no_of_customer] フィールドで、それぞれの［product_id］の商品をもっとも多額に購入した顧客数を取得できた

なお、2つの商品について同額を購入しているため、[revenue_rank] が「1」となる［product_id］を2つ持つユーザーが2人います。そのため、ユーザー数の合計である497人に対して、[no_of_customers］の合計は499人になっています。

ROW_NUMBER関数によるゴールデン導線分析

Webサイトのログデータにおいて、「あるユーザーがページAにランディングし、次にページB→ページCの順に遷移したあと、コンバージョンした」といったデータを集計し、サイト全体でコンバージョンに至りやすい導線がないかを分析することを「ゴールデン導線分析」と呼びます。

ゴールデン導線分析は、Webサイト全体に対して実施しても、あまり知見は得られません。しかし、特定のランディングページからであれば、有用な気付きを得られることがあります。

この分析を行うときには、ヒットカウント、つまり「セッションの中で何番目に見たページなのか？」について番号を振る必要があります。以下の［8-2-5］のテーブルでいえば［ヒットカウント］フィールドです。

8-2-5 Webサイトのログデータのヒットカウント

ユーザーID	セッションカウント	タイムスタンプ	ページ	ヒットカウント
A	1	2020-01-01 18:05:45	/page_a.html	1
A	1	2020-01-01 18:06:02	/page_b.html	2
A	1	2020-01-01 18:06:12	/page_c.html	3
B	2	2020-01-01 22:15:55	/page_c.html	1
B	2	2020-01-01 22:15:59	/page_a.html	2

このヒットカウントの取得に利用できるのが**ROW_NUMBER関数**（ロウ・ナンバー）です。ROW_NUMBER関数は、パーティション内で順番に「1、2、3、…」と整数値を取得します。

RANK関数とは違い、値の大きさは問わず、とにかく順番通りに数値を振る処理をします。よって、同じ値があってもスキップせずに、抜けのない整数を割り当てます。構文は次ページの［8-2-6］となります。

8-2-6 ROW_NUMBER関数の構文

```
1  ROW_NUMBER() OVER (
2  PARTITION BY パーティションとして定義するフィールド名
3  ORDER BY パーティション内での並べ替えの基準とするフィールド名
4  )
```

関数名の後の（）	何も記述しない
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	オプション：省略時は適当な順番が振られる
WINDOWフレーム句	利用不可
戻り値のデータ型	整数型

では、セッションごとのヒットカウントを取得するSQLの全体像を見ていきましょう。具体的には次ページの［8-2-7］で、演習用の［web_log］テーブルを対象としています。結果テーブルが大きいと分かりにくいので、［user_id］が「11779」のユーザーに絞っています。

3行目にあるPARTITION BY句の後ろに、［user_id］と［session_count］の2つのフィールドが指定されていることに注目してください。［user_id］と［session_count］の両方を指定することで、「セッションにおける」ヒットカウントを取得できます。

結果テーブルにある［session_count］が「1」「3」「4」は、1セッションで1ページしか閲覧されていない、いわゆる「直帰」なので、セッション中のヒットカウント（hit_count）は「1」だけが記録されています。一方、［session_count］が「2」のセッションは、1セッションで3ページ閲覧されており、［date_time］順にヒットカウントが「1」「2」「3」と記録されています。このセッションがコンバージョンしているとすれば、コンバージョンに至ったページの導線が明らかになりました。

8-2-7 セッションごとのヒットカウントを取得する

```
1  SELECT user_id, session_count, date_time, page
2  , ROW_NUMBER() OVER (
3  PARTITION BY user_id, session_count
4  ORDER BY date_time ) AS hit_count
5  FROM sample.web_log
6  WHERE user_id = 11779
7  ORDER BY 2
```

結果テーブル

行	user_id	session_count	date_time	page	hit_count
1	11779	1	2018-05-23T20:33:10	/prod/prod_id_12/	1
2	11779	2	2018-11-13T14:02:20	/	1
3	11779	2	2018-11-13T14:02:25	/special/diet/	2
4	11779	2	2018-11-13T14:02:54	/prod/prod_id_1/	3
5	11779	3	2018-12-17T16:45:51	/prod/prod_id_2/	1
6	11779	4	2019-03-13T11:54:40	/	1

[hit_count] フィールドでヒットカウントを取得できた

この分析はすごいですね！ Googleアナリティクスでもできるのでしょうか？

残念ながら、従来のGoogleアナリティクスではできません。最新バージョンであるGA4が、BigQueryにエクスポートする生データにSQLを記述して取得することになります。

STEP UP

デシル分析とNTILE関数

CRM分析の1つに「デシル分析」があります。顧客を販売金額の大きい順に並べたうえで、同じ顧客数となるように10等分し、バケット（10分割された顧客グループ）ごとの販売金額合計や全体に占める割合を分析するものです。

RANK関数やROW_NUMBER関数より多少利用頻度は落ちるものの、デシル分析を行うときに最適な関数があるので紹介します。それは、**NTILE関数**です。関数名の後のカッコ内には分割したいグループ数を記述するので、デシル分析では「NTILE(10)」と記述します。

以下のSQL文［8-2-8］では、[sales] テーブルを対象に、2017年に購入のあった顧客をデシルに分割しています。また、デシルに属する顧客ごとの購入金額の合計が、2017年の販売全体に対してどのような割合を占めているかを求めています。

17〜20行目がいちばん内側のクエリです。その中の18行目に「NTILE(10)」が利用されています。結果テーブルを見ると、購入金額トップ10%、つまり［rev_bucket］が「1」に属する顧客の購入金額の合計が、全体の販売金額の約26%を占めていることが分かります。

8-2-8 2017年の販売金額を対象にデシル分析をする

```
WITH master AS (
SELECT user_id, SUM(revenue) AS sum_rev
FROM sample.sales
WHERE FORMAT_DATETIME("%Y", date_time) = "2017"
GROUP BY user_id
)

```

```
8   SELECT *
9   , round(sum_rev_by_bucket / sum_rev_total, 2)
10  AS rev_percentage
11  FROM (
12  SELECT rev_bucket
13  , SUM(sum_rev) AS sum_rev_by_bucket
14  , (SELECT SUM(sum_rev) FROM master)
15  AS sum_rev_total
16  FROM (
17  SELECT user_id, sum_rev
18  , NTILE(10) OVER (ORDER BY sum_rev DESC)
19  AS rev_bucket
20  FROM master
21  )
22  GROUP BY rev_bucket
23  )
```

結果テーブル

行	rev_bucket	sum_rev_by_bucket	sum_rev_total	rev_percentage
1	1	202630	766560	0.26
2	2	131380	766560	0.17
3	3	102180	766560	0.13
4	4	78300	766560	0.1
5	5	67400	766560	0.09
6	6	52200	766560	0.07
7	7	43050	766560	0.06
8	8	34600	766560	0.05

[rev_percentage] フィールドで、デシルごとの販売金額に占める割合を取得できた

[rev_backet] が「3」までで販売全体の半分以上が発生していることが分かる

BIツールを使っても、ある程度は習熟していないとできないような分析が、SQLでできることになります。ウィンドウ関数がなぜ分析関数と呼ばれるのか、実感できるのではないでしょうか。

NTILE関数の構文は以下の［8-2-9］の通りです。

8-2-9 NTILE関数の構文

```
1  NTILE() OVER (
2  PARTITION BY パーティションとして定義するフィールド名
3  ORDER BY パーティション内での並べ替えの基準とするフィールド名
4  )
```

関数名の後の（）	レコードを均等な数で分割したいグループの数を整数で指定
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	利用不可
戻り値のデータ型	整数型

処理としては、ORDER BY句で指定した順番に並べたレコードが均等な数で分類されるように、「NTILE」の後ろのカッコ内に記述したグループ数に該当する整数を返します。例えば、100レコードあるテーブルに対して「NTILE(5)」とすると、最初の20レコードに「1」、次の20レコードに「2」と続き、最後の20レコードには「5」が戻り値として返されます。

SECTION

8-3 最初の値・別レコードの値の取得

ウィンドウ関数の「ナビゲーション関数」を使った分析のテクニックを覚えましょう。

「ナビゲーション」と聞くと、導いてもらえそうな気がしますね。

特定の日からの経過日数や前月と今月の販売金額の差など、元のテーブルにある値を処理して、有用なデータを取得できます。CRM分析の仮説の検証などに役立ちます。

初回が最大額の注文である顧客を求める

CRM分析をしていると、顧客をグループ化して分析する「コホート分析」において、独自で、さまざまなバリエーションのグループ（＝コホート）についての仮説を検証したくなります。

例えば、「初回注文の購入金額が、その顧客にとっての複数回の注文の中で最大額だった顧客は、そうでない顧客に比べてLTVの増加ペースが早いのではないか？」といった仮説です。

次ページの［8-3-1］のような購入実績のある２人の顧客について、具体的に考えてみましょう。[user_id] が「1」の顧客は、2,000円、3,500円、3,900円と徐々に購入金額（revenue）を高めています。一方、[use_id] が「2」の顧客は、初回購入時は4,200円、次に2,000円と、初回にもっとも高額の購入をしています。

このとき、『２人の顧客のうち、「2」の顧客のほうがLTVの増加ペースが早いのではないか？』という仮説が浮かんだとしましょう。

8-3-1 顧客の購入実績例

user_id	date	revenue
1	2020-01-10	2000
1	2020-02-25	3500
1	2020-03-03	3900
2	2020-02-05	4200
2	2020-02-28	2000

この仮説が正しいと証明できれば、「初回に最大額の購入をする顧客の特徴はどのようなものであり、それは再現できないだろうか？」と、分析から成果を導く思考につながります。

そこで「複数回購入した顧客のうち、初回が最大額の注文である顧客」を抽出するには、ナビゲーション関数である**FIRST_VALUE関数**（ファースト・バリュー）を利用します。構文は以下の［8-3-2］の通りで、パーティション内を並べ替えたうち、もっとも上のレコードの値を取得します。

8-3-2 FIRST_VALUE関数の構文

```
1  FIRST_VALUE(フィールド名) OVER (
2  PARTITION BY パーティションとして定義するフィールド名
3  ORDER BY パーティション内での並べ替えの基準とするフィールド名
4  WINDOWフレーム
5  )
```

関数名の後の（）	取得したい値を含むフィールド名を指定
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	オプション
戻り値のデータ型	関数名の後の()で指定したフィールドのデータ型

FIRST_VALUE関数ではWINDOWフレーム句を指定できますが、省略も可能です。ORDER BY句で時系列での並べ替えをしておけば、求める戻り値はパーティションのもっとも上のレコードにあるので、WINDOWフレーム句を省略した場合に適用される「パーティションの上限から現在のレコードまで」をフレームとしてよいからです。

今回のテーマである「複数回購入した顧客のうち、初回が最大額の注文である顧客」については、次の考え方で取得します。

1. 顧客ごとに初回の購入日を取得する
2. 顧客ごとに最大額の購入日を取得する
3. 1と2の日付が同一の顧客に絞り込む

初回の購入日は、FIRST_VALUE関数を利用して取得します。[user_id]でパーティションを定義し、購入日である［date_time］が小さい順に並べて、FIRST_VALUE関数でもっとも上のレコードを取得することで、顧客別の初回購入の日付を取得できます。

最大額の購入日は、FIRST_VALUE関数で指定するORDER BY句を工夫します。パーティションを［user_id］で作成し、購入金額である［revenue］を降順に並べ替えます。そのうえで、FIRST_VALLUE関数でもっとも上のレコードを取得することで、顧客別の最大の購入額の日付を取得します。

これらをFIRST_VALUE関数の構文に当てはめたのが、以下の［8-3-3］と次ページの［8-3-4］です。

8-3-3 顧客ごとの初回の購入日を取得する

```
FIRST_VALUE(date_time) OVER (
PARTITION BY user_id
ORDER BY date_time
)
```

8-3-4 顧客ごとの最大額の購入日を取得する

```
FIRST_VALUE(date_time) OVER (
PARTITION BY user_id
ORDER BY revenue DESC
)
```

初回の購入日と最大額の購入日が取得できれば、あとはWHERE句で「初回の購入日＝最大額の購入日」での絞り込みが可能になります。「複数回購入した顧客のうち、初回が最大額の注文である顧客」を求めるSQL文全体は、以下の［8-3-5］となります。

17〜21行目では、[order_id] を2つ以上持っている顧客、つまり複数回注文した顧客に絞り込んでいます。1回しか注文していない顧客は全員、初回購入日＝最大額購入日になってしまうからです。

[sales] テーブルを対象に実行すると、94人の顧客が結果に表示されました。これらをコホートとして分析すれば、「初回注文の購入金額が複数回の注文の中で最大額だった顧客は、そうでない顧客に比べてLTVの増加ペースが早いのではないか？」という当初の仮説を検証できます。

8-3-5 初回が最大額の注文である顧客を求める

```
WITH master AS (
SELECT user_id, FORMAT_DATETIME("%F", date_time)
AS purchase_date, SUM(revenue) AS sum_rev
FROM sample.sales
GROUP BY user_id, date_time
)

```

```
SELECT user_id FROM (
SELECT user_id
, FIRST_VALUE(purchase_date) OVER (
PARTITION BY user_id ORDER BY purchase_date
) AS first_purchase_date
, FIRST_VALUE(purchase_date) OVER (
PARTITION BY user_id ORDER BY sum_rev DESC
) AS biggest_purchase_date
FROM  master
WHERE user_id IN (
SELECT user_id
FROM master
GROUP BY user_id
HAVING COUNT(DISTINCT purchase_date) >1
) )
WHERE first_purchase_date = biggest_purchase_date
GROUP BY user_id
ORDER BY 1
```

結果テーブル

行	user_id
1	15686
2	15704
3	15706
4	15732

初回が最大額の注文である[user_id]の値を取得できた

初回と2回目の購入日との差を求める

ナビゲーション関数として続いて覚えたいのが、**NTH_VALUE関数**です。この関数ではパーティション内を並べ替えたうち、任意の順番の値を取得できます。利用シーンとしては、例えば「初回と2回目の購入日の日付の差、つまり初回の購入日から何日後に2回目の購入をしたのか？」といった分析に役立ちます。NTH_VALUE関数の構文は、次ページの［8-3-6］の通りです。

8-3-6 NTH_VALUE関数の構文

```
NTH_VALUE(フィールド名, 値の順番) OVER (
PARTITION BY パーティションとして定義するフィールド名
ORDER BY パーティション内での並べ替えの基準とするフィールド名
WINDOWフレーム
)
```

関数名の後の（）	取得したい値を含むフィールド名と、パーティション内の何番目の値を取得するかを整数で指定
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	オプション
戻り値のデータ型	関数名の後の()で指定したフィールドのデータ型

WINDOWフレーム句を省略すると「パーティションの上端から現在のレコード」までとなり、その範囲内で「NTH_VALUE」の後のカッコの［値の順番］で指定した値を返します。取得したい値があるレコードの順番に応じて、WINDOWフレーム句を指定してください。

例として［sales］テーブルを対象に、［user_id］が「19940」の顧客について、初回と2回目の購入日の差を「日数」で取得してみましょう。完成形のSQL文は以下の［8-3-7］の通りです。

8-3-7 初回と2回目の購入日の差を求める

```
WITH master AS (
SELECT user_id, order_id, MIN(date_time)
AS order_datetime
```

```
4   FROM sample.sales
5   WHERE user_id = 19940
6   GROUP BY user_id, order_id
7   )
8   SELECT user_id
9   , MAX(FORMAT_DATETIME("%F", first_purchase_datetime))
10  AS first_purchase_date
11  , MAX(FORMAT_DATETIME("%F", second_purchase_datetime))
12  AS second_purchase_date
13  , DATETIME_DIFF(MAX(second_purchase_datetime)
14  , MAX(first_purchase_datetime), DAY) AS day_interval
15  FROM (
16  SELECT user_id
17  , NTH_VALUE(order_datetime, 1)
18  OVER(PARTITION BY user_id ORDER BY order_datetime)
19  AS first_purchase_datetime
20  , NTH_VALUE(order_datetime, 2)
21  OVER(PARTITION BY user_id ORDER BY order_datetime)
22  AS second_purchase_datetime
23  FROM master
24  )
25  GROUP BY user_id
```

結果テーブル

行	user_id	first_purchase_date	second_purchase_date	day_interval
1	19940	2017-04-07	2017-05-20	43

初回・2回目の購入日と、
その差の日数を取得できた

1〜7行目のWITH句では、[sales] テーブルに対して [user_id] を「19940」の顧客に絞り込み、[user_id] [order_id] [date_time] を一意の値に加工しています。元の [sales] テーブルは注文（order_id）と商品（product_id）でレコードが分かれているため、そのままでは初回の注文日と2回目の注文日を取り出せないためです。

17行目の「NTH_VALUE(order_datetime, 1)」では、「OVER」に続くORDER BY句で [order_datetime] の小さい順に並べ替えたパーティション内の1レコード目の値、つまり、初回の購入日を取得しています。20行目の「NTH_VALUE(order_datetime, 2)」も同様に、パーティション内の2レコード目の値、つまり2回目の購入日を取得しています。

ここまでのSQL文（1〜7行目、16〜23行目）だけを実行すると、以下の [8-3-8] の結果テーブルが得られます。初回の購入日（first_purchase_datetime）と2回目の購入日（second_purchase_datetime）は取得できていますが、ウィンドウ関数はグループ化とともに利用する集計関数のようにレコードを1つにまとめないため、WITH句が生成した仮想テーブル（master）のレコード数がそのまま維持されています。

8-3-8 初回と2回目の購入日

行	user_id	first_purchase_datetime	second_purchase_datetime
1	19940	2017-04-07T15:57:14	*null*
2	19940	2017-04-07T15:57:14	2017-05-20T20:06:13
3	19940	2017-04-07T15:57:14	2017-05-20T20:06:13
4	19940	2017-04-07T15:57:14	2017-05-20T20:06:13
5	19940	2017-04-07T15:57:14	2017-05-20T20:06:13
6	19940	2017-04-07T15:57:14	2017-05-20T20:06:13
7	19940	2017-04-07T15:57:14	2017-05-20T20:06:13
8	19940	2017-04-07T15:57:14	2017-05-20T20:06:13

1〜7行目、16〜23行目のSQL文で初回・2回目の購入日が取得できる

そこで、外側のクエリでグループ化と集計、および［second_purchase_datetime］から［first_purchase_datetime］を引いて「日」で表示することにより、2つの購入日の差を日数で取得します。当然、グループ化は［user_id］で行います。［user_id］は1種類しかないため、1行にまとめられます。

［first_purchase_datetime］と［second_purchase_datetime］を1レコードにするには、MAX関数を利用し、購入日の引き算にはDATETIME_DIFF関数を利用しています。最終的な結果テーブルは［8-3-7］の通りです。

なお、4行目のWHERE句で「user_id = 19940」に絞り込まなければ、複数回の購入実績のあるすべての顧客について、同様のレコードを取得できます。初回の購入日から何日後に2回目の購入が多いかが分かり、2回目の購入促進施策を立案するヒントにできます。

「NTH」は「エヌ・ティー・エイチ」ではなく「エヌス」と呼ぶのが適切と思われます。「fourth」（フォース）や「fifth」（フィフス）などのように「N番目の」を意味しているからです。

最終購入日からの経過日数を求める

CRM分析からのアクションとして、「最終購入日から○日経過した顧客」に対して休眠防止のためのコミュニケーションをとりたいことがあります。そのようなケースで顧客の最終購入日を取得できるのが、ナビゲーション関数の**LAST_VALUE関数**（ラスト・バリュー）です。パーティション内を並べ替えたうち、最終レコードの値を取得できます。

最終購入日が取得できれば、CHAPTER 7で学んだDATE_DIFF関数やDATETIME_DIFF関数で、最終購入日から今日までの経過日数を取得できます。経過日数が取得できれば、WHERE句で容易に絞り込みができます。

LAST_VALUE関数の構文は次ページの［8-3-9］となります。

8-3-9 LAST_VALUE関数の構文

```
LAST_VALUE(フィールド名) OVER (
PARTITION BY パーティションとして定義するフィールド名
ORDER BY パーティション内での並べ替えの基準とするフィールド名
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
)
```

関数名の後の（）	取得したい値を含むフィールド名を指定
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	オプション
戻り値のデータ型	関数名の後の()で指定したフィールドのデータ型

関数の構造としてはFIRST_VALUE関数と同じですが、WINDOWフレーム句を上記の通りとしました。これは「パーティションの上端から下端まで」を指示する文言です。LAST_VALUE関数においてWINDOWフレーム句はオプションではありますが、省略した場合、「パーティションの上端から現在のレコード」が対象となり、パーティション全体の下端の値（ここでは最終購入日）を取得できません。上記の記述は必須と考えてください。

したがって、顧客ごとの最終購入日を取得するには、以下の［8-3-10］のように記述します。結果テーブルでは、1人の顧客が複数レコードにわたって同一の［last_purchase_date］の値を持ちます。

8-3-10 顧客ごとの最終購入日を取得する

```
SELECT user_id
, LAST_VALUE(date_time) OVER (
```

```
3   PARTITION BY user_id
4   ORDER BY date_time
5   ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
6   ) AS last_purchase_date
7   FROM sample.sales
```

結果テーブル

行	user_id	last_purchase_date
1	10059	2019-09-24T10:05:45
2	10059	2019-09-24T10:05:45
3	10059	2019-09-24T10:05:45
4	10060	2018-09-05T04:04:55

[last_purchase_date] で顧客ごとの最終購入日を取得できた

ここでの目的は「最終購入日から○日経過した顧客」を明らかにすることなので、さらにSQL文を追加していきます。仮に今日が2019年12月31日として、[sales] テーブルを対象に最終購入日から90日以上経過した顧客を取得するSQL文は、以下の［8-3-11］となります。

冒頭のWITH句には、顧客ごとの最終購入日を取得する前掲の［8-3-10］を指定しています。複数のレコードで取得される同一の［user_id］と［last_purchase_date］を顧客ごとにまとめるため、18行目のGROUP BY句で［user_id］をグループ化し、12行目でMAX関数を利用しています。

結果テーブルは次ページの通りです。なお、最終購入日から今日（SQLを実行する日）までの経過日数が90日以上の顧客を取得するには、14行目を「DATETIME_DIFF(CURRENT_DATETIME("Asia/Tokyo")」にします。

8-3-11 最終購入日から90日以上経過した顧客を取得する

```
1   WITH master AS (
2   SELECT user_id
3   , LAST_VALUE (date_time) OVER (
4   PARTITION BY user_id
```

```
  ORDER BY date_time
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
  ) AS last_purchase_date
  FROM sample.sales
  )

  SELECT user_id
  , MAX(last_purchase_date)
  AS last_purchase_date_by_customer
  , DATETIME_DIFF(DATETIME "2019-12-31 23:59:59"
  , MAX(last_purchase_date), DAY)
  AS days_from_last_purchase
  FROM master
  GROUP BY user_id
  HAVING days_from_last_purchase >= 90
  ORDER BY 3
```

結果テーブル

行	user_id	last_purchase_date_by_customer	days_from_last_purchase
1	18246	2019-10-01T12:29:00	91
2	12172	2019-09-30T14:08:08	92
3	11640	2019-09-29T23:51:56	93
4	12397	2019-09-29T15:25:22	93
5	17693	2019-09-27T14:34:40	95

最終購入日から 90 日以上経過した顧客と、その顧客の最終購入日時、経過日数を取得できた

Webページの滞在時間を取得する

Webサイトの利用ログには、ページビューが発生した日付・時刻が記録されているだけで、ページの滞在時間が記録されていないことがよくあります。例えば、次ページの［8-3-12］のようなイメージです。

8-3-12 Webサイトの利用ログの例

ユーザーID	ページ	タイムスタンプ
A	/page_a.html	2020-01-01 18:05:45
A	/page_b.html	2020-01-01 18:06:02
A	/page_c.html	2020-01-01 18:06:12
B	/page_c.html	2020-01-01 22:15:55
B	/page_a.html	2020-01-01 22:15:59

上記のテーブルで、現在のレコードの直後（1行後ろ）のタイムスタンプを取得し、DATETIME_DIFF関数を利用して、直後のタイムスタンプから現在のタイムスタンプを差し引けば、Webページの滞在時間を取得できます。この「現在のレコードの直後のレコードの値」を取得できるのが、ナビゲーション関数の**LEAD関数**です。LEAD関数の構文は以下の［8-3-13］の通りです。

8-3-13 LEAD関数の構文

```
1  LEAD(フィールド名) OVER (
2  PARTITION BY パーティションとして定義するフィールド名
3  ORDER BY パーティション内での並べ替えの基準とするフィールド名
4  )
```

関数名の後の（）	取得したい値を含むフィールド名を指定
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	オプション
戻り値のデータ型	関数名の後の()で指定したフィールドのデータ型

LEAD関数を利用して、演習用の［web_log］テーブルから各ページの滞在時間を取得するSQL文は次ページの［8-3-14］となります。3行目のLEAD

関数により、現在のレコードの直後のレコードの［date_time］が取得され、［lead_date_time］フィールドに格納されています。

さらに、11行目にあるDATETIME_DIFF関数で2つのタイムスタンプの差を秒（SECOND）として取得し、結果テーブルの［time_on_page］フィールドに格納しました。

8-3-14 Webページごとの滞在時間を取得する

```
1  WITH master AS (
2  SELECT *
3  , LEAD (date_time)
4  OVER (PARTITION BY cid, session_count
5  ORDER BY date_time) AS lead_date_time
6  FROM sample.web_log
7  )
8
9  SELECT
10 cid, session_count, page, date_time, lead_date_time
11 , DATETIME_DIFF(lead_date_time, date_time, SECOND)
12 AS time_on_page
13 FROM master
14 ORDER BY 1, 2, 4
```

結果テーブル

行	cid	session_count	page	date_time	lead_date_time	time_on_page
1	1000001988.1500862980	20	/prod/prod_id_14/	2018-08-14T17:36:52	2018-08-14T17:37:08	16
2	1000001988.1500862980	20	/prod/prod_id_14/	2018-08-14T17:37:08	2018-08-14T17:37:36	28
3	1000001988.1500862980	20	/prod/prod_id_4/	2018-08-14T17:37:36	2018-08-14T17:38:12	36
4	1000001988.1500862980	20	/prod/prod_id_2/	2018-08-14T17:38:12	2018-08-14T17:38:33	21
5	1000001988.1500862980	20	/prod/prod_id_2/	2018-08-14T17:38:33	2018-08-14T17:38:43	10

［date_time］と［lead_date_time］の差からページ滞在時間を取得できた

なお、[8-3-14] のSQL文に少し追記して以下の [8-3-15] とすると、各ページの平均滞在時間を取得できます。

DATETIME_DIFF関数で取得したタイムスタンプの差をAVG関数で平均し、[page] でグループ化します。同じく [web_log] テーブルを対象としているので、みなさんのBigQuery環境でもお試しください。

8-3-15 Webページごとの平均滞在時間を取得する

```
WITH master AS (
SELECT *
, LEAD (date_time)
OVER (PARTITION BY cid, session_count
ORDER BY date_time) AS lead_date_time
FROM sample.web_log
)

SELECT page
, ROUND (AVG(DATETIME_DIFF (
lead_date_time, date_time, SECOND)), 0)
AS avg_time_on_page
FROM master
GROUP BY page
ORDER BY 2 DESC
```

結果テーブル

行	page	avg_time_on_page
1	/prod/prod_id_10/?sys=abc123	215.0
2	/prod/prod_id_4/	204.0
3	/prod/prod_id_13/?sys=abc123	171.0
4	/prod/prod_id_11/?sys=abc123	162.0
5	/prod/prod_id_8/?sys=abc123	161.0

[avg_time_on_page] として各ページの平均滞在時間を取得できた

当月と前月の販売個数の差を求める

『先月の販売個数が「100」で今月が「120」だから、差は「20」、成長率は「20%」』といった前月比での分析は、日常的によく行われます。そのような分析にはLAG関数を使います。

LAG関数は、パーティション内のうち、現在のレコードの直前のレコード（1行前のレコード）の値を取得できます。構文は以下の［8-3-16］の通りで、構造としてはLEAD関数と同じです。

8-3-16 LAG関数の構文

```
1  LAG(フィールド名) OVER (
2  PARTITION BY パーティションとして定義するフィールド名
3  ORDER BY パーティション内での並べ替えの基準とするフィールド名
4  )
```

関数名の後の（）	取得したい値を含むフィールド名を指定
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	オプション
戻り値のデータ型	関数名の後の()で指定したフィールドのデータ型

ここでは演習用の［sales］テーブルを対象に、2017年における各月の販売金額を、前月の販売金額と比較してみましょう。完成したSQL文である次ページの［8-3-17］から読み解きます。

冒頭では、WITH句で2017年の月別の販売金額（revenue）を合計しています。11行目では、合計した販売金額（sum_rev）に対して、LAG関数で1行前の［sum_rev］を取得しています。

すると、結果テーブルの通り、当月の販売金額（sum_rev）と前月の販売

金額（previous_sum_rev）が同じレコードに記録されました。このテーブルから、差額や差の割合を計算するのはもう簡単ですね。ぜひみなさんで考えてみてください。

8-3-17 各月の販売金額を前月と比較する

```
WITH master AS (
SELECT FORMAT_DATETIME("%Y-%m", date_time)
AS year_month
, SUM(revenue) AS sum_rev
FROM sample.sales
WHERE FORMAT_DATETIME("%Y", date_time) = "2017"
GROUP BY year_month
)

SELECT year_month, sum_rev
, LAG(sum_rev) OVER (ORDER BY year_month)
AS previous_sum_rev
FROM master
ORDER BY 1
```

結果テーブル

行	year_month	sum_rev	previous_sum_rev
1	2017-01	25600	*null*
2	2017-02	89900	25600
3	2017-03	61900	89900
4	2017-04	92660	61900
5	2017-05	90100	92660
6	2017-06	47420	90100
7	2017-07	90620	47420
8	2017-08	50450	90620
9	2017-09	70460	50450

当月と前月の販売金額を並べて取得できた

STEP UP

五数要約を実現するPERCENTILE_CONT関数

Google アナリティクスなどを利用して、Webサイトの利用状況の解析を行っている人も多いと思います。Google アナリティクスはデータを集約する機能があるため、以下のような値を得ても、「データのバラつき」までは意識することが少ないかもしれません。

- キャンペーンAのコンバージョン率が「2.0%」
- キャンペーンBのコンバージョン率が「1.5%」

しかし、キャンペーンを1か月実施していれば、それぞれ「日別のコンバージョン率」が存在したはずです。30日間であれば、キャンペーンごとに30個のコンバージョン率があったことになります。日別の30個のコンバージョン率がすべて「2.0%」や「1.5% 」だったとは考えにくく、日によってバラついているのが現実だと思います。

そうしたデータのバラつきを明らかにする1つの方法として「五数要約」があります。バラつきのある指標について、次の5つの値で要約するものです。

- 最小値
- 第一四分位数
- 中央値
- 第三四分位数
- 最大値

さきほどのキャンペーンA、Bに対して五数要約を行うと、日別のバラつきはもちろん、「コンバージョン率が特定の日や曜日によって変動しているのではないか？」といった仮説も立てやすくなります。また、平均値の「2.0%」や「1.5%」を代表値としてみなすことの妥当性につ

いても、おのずと考えに至るはずです。

五数要約の指標は、PERCENTILE_CONT関数で取得できます。構文は以下の［8-3-18］の通りです。

8-3-18 PERCENTILE_CONT関数の構文

```
1  PERCENTILE_CONT(フィールド名, 分位数) OVER (
2  PARTITION BY パーティションとして定義するフィールド名
3  ORDER BY パーティション内での並べ替えの基準とするフィールド名
4  )
```

関数名の後の（）	最小値や中央値を取得したいフィールド名と、取得したい分位数を「0」から「1」までの値で指定
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	オプション
戻り値のデータ型	関数名の後の()で指定したフィールドのデータ型

構文中の「PERCENTILE_CONT」の後のカッコ内で指定する［分位数］には、「0」から「1」までの値を次ページの表［8-3-19］にしたがって入力します。例えば「0.25」を指定すると、小さいほうから並べたときに4分の1の位置にある値（第一四分位数）を取得できます。

8-3-19 PERCENTILE_CONT関数で取得できる指標

指定する値	取得できる指標	意味
0	最小値	パーティション内のもっとも小さい値
0.25	第一四分位数	パーティション内を値の小さい順に並べたとき、1/4の位置にある値
0.5	中央値	パーティション内を値の小さい順に並べたとき、ちょうど半分の位置にある値
0.75	第三四分位数	パーティション内を値の小さい順に並べたとき、3/4の位置にある値
1	最大値	パーティション内のもっとも大きい値

例えば、[sales] テーブルの販売金額（revenue）を商品ID（product_id）単位で五数要約するには、以下のSQL文［8-3-20］を実行します。結果テーブルを見ると、15個の［product_id］についてきれいに五数要約ができていることが分かります。

およそすべての商品について、最小値と中央値は近い値なものの、最大値と中央値は大きく離れているようです。「外れ値として、ごく少数のユーザーが大量に購入しているのではないか？」「平均値は、その外れ値に引っ張られて高めに出ているかもしれない」などの仮説が導けるでしょう。

8-3-20 商品ごとに販売金額を五数要約する

```
1  SELECT product_id
2  , MAX(minimum) AS minimum_by_product
3  , MAX(first) AS first_quartilem_by_product
4  , MAX(median) AS median_by_product
```

```
5   , MAX(third) AS third_quartile_by_product
6   , MAX(maximum) AS maximum_by_product
7   FROM (
8   SELECT product_id
9   , PERCENTILE_CONT(revenue, 0)
10  OVER(PARTITION BY product_id) AS minimum
11  , PERCENTILE_CONT(revenue, 0.25)
12  OVER(PARTITION BY product_id) AS first
13  , PERCENTILE_CONT(revenue, 0.5)
14  OVER(PARTITION BY product_id) AS median
15  , PERCENTILE_CONT(revenue, 0.75)
16  OVER(PARTITION BY product_id) AS third
17  , PERCENTILE_CONT(revenue, 1)
18  OVER(PARTITION BY product_id) AS maximum
19  FROM sample.sales
20  )
21  GROUP BY product_id
```

結果テーブル

行	product_id	minimum_by_product	first_quartilem_by_product	median_by_product	third_quartile_by_product	maximum_by_product
1	1	1600.0	2000.0	2000.0	2000.0	8000.0
2	2	1760.0	2200.0	2200.0	2200.0	8800.0
3	3	1440.0	1800.0	1800.0	1800.0	3600.0
4	4	1920.0	2400.0	2400.0	3360.0	9600.0
5	5	2240.0	2800.0	2800.0	2800.0	8400.0
6	6	3240.0	3600.0	3600.0	3600.0	10800.0
7	7	1080.0	1200.0	1200.0	1200.0	2400.0
8	8	1080.0	1200.0	1200.0	1200.0	3600.0
9	9	960.0	1200.0	1200.0	1200.0	2400.0
10	10	2000.0	2500.0	2500.0	2500.0	7500.0

商品ごとの販売金額について、
五数要約の指標を取得できた

SECTION

8-4 累計・移動平均の取得

本節では、すでに集計関数として学習済みであるSUM関数やAVG関数を、ウィンドウ関数として利用する方法を学びます。ウィンドウ関数として利用する場合は「集計分析関数」と呼ばれます。

集計関数として利用した場合とは、異なる戻り値を得ることができるということですね。

はい。累計や移動平均など、通常はBIツールを利用して求めるような値を関数で導くことができます。

累計を取得する

集計分析関数は外見上、CHAPTER 4で学んだ集計関数と同じ形をしています。しかし、OVER句を伴うことで、集計分析関数として「元のレコードの値を維持したまま」集計を行います。例えば、SUM関数では［8-4-1］の［acc_revenue］フィールドのような「累計」の取得が可能です。

8-4-1 販売金額の累計

user_id	date	revenue	acc_revenue
1	2020-01-10	2000	2000
1	2020-02-25	3500	5500
1	2020-03-03	3900	9400
2	2020-02-05	4200	4200
2	2020-02-28	2000	6200

代表的な集計分析関数には、SUM関数のほかにも、AVG関数、MAX関数、MIN関数、COUNT関数が挙げられます。それぞれの構文は共通で、以下の［8-4-2］の通りです。パーティション内のうち、WINDOWフレーム句で指定した範囲のレコードに対して集計関数で計算した値を取得します。

8-4-2 集計分析関数の構文

```
関数名(フィールド名) OVER (
PARTITION BY パーティションとして定義するフィールド名
ORDER BY パーティション内での並べ替えの基準とするフィールド名
WINDOWフレーム
)
```

関数名の後の（）	取得したい値を含むフィールド名を指定
PARTITION BY句	オプション：省略時はテーブル全体が対象
ORDER BY句	必須
WINDOWフレーム句	オプション
戻り値のデータ型	関数名の後の()で指定したフィールドのデータ型

この構文にSUM関数を当てはめ、［8-4-1］の累計を結果テーブルとして得るための集計分析関数は、次ページの［8-4-3］のようになります。

8-4-3 ユーザーごとの販売金額の累計を取得する

```
1  SUM(revenue) OVER (
2  PARTITION BY user_id
3  ORDER BY date
4  ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
5  )
```

すでに番号付け関数やナビゲーション関数を見てきたので、PARTITION BY句とORDER BY句については、問題なく理解できると思います。

WINDOWフレーム句では、パーティション内のどのレコードを利用するかを指示します。累計とは何かを考えると、「現在のレコードと過去のすべてのレコードを足し上げる」ことにほかならないので、WINDOWフレーム句は「パーティションの上限から現在のレコードまで」となるべきで、それを「ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW」で指定しています。

とはいえ、WINDOWフレーム句を省略すると、同じく「UNBOUNDED PRECEDINGAND CURRENT ROW」が指定されたことになるので、[8-4-3]では省略しても結果は同じになります。

累計を、より実践的に利用するため、演習用の［sales］テーブルに対して、四半期ごとの販売金額の累計を年別に取得するSQL文を記述すると、次ページの［8-4-4］となります。8〜11行目で［revenue］を四半期ごとに集計した仮想テーブルを作成し、外側のクエリの3つ目のフィールドで四半期ごとに集計された販売金額の累計を求めています。

3行目にあるPARTITION BY句で指定しているパーティションが年となっているので、結果テーブルの販売金額累計は、年が切り替わるとリセットされていることが見てとれます。

8-4-4 四半期ごとの販売金額の累計を年別に取得する

```
SELECT year_quarter, sum_rev
, SUM(sum_rev) OVER (
PARTITION BY FORMAT_DATETIME("%Y", year_quarter)
ORDER BY year_quarter
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
) AS accumulated_revenue
FROM (
SELECT
DATETIME_trunc(date_time, quarter) AS year_quarter
, SUM(revenue) AS sum_rev
FROM sample.sales
GROUP BY year_quarter
)
```

結果テーブル

行	year_quarter	sum_rev	accumulated_revenue
1	2017-01-01T00:00:00	177400	177400
2	2017-04-01T00:00:00	230180	407580
3	2017-07-01T00:00:00	211530	619110
4	2017-10-01T00:00:00	147450	766560
5	2018-01-01T00:00:00	284740	284740
6	2018-04-01T00:00:00	244070	528810
7	2018-07-01T00:00:00	259050	787860
8	2018-10-01T00:00:00	251430	1039290
9	2019-01-01T00:00:00	337640	337640
10	2019-04-01T00:00:00	325350	662990
11	2019-07-01T00:00:00	297260	960250
12	2019-10-01T00:00:00	247120	1207370

[accumulated_revenue]で四半期ごとの販売金額の累計を取得できた

移動平均を取得する

「元のレコードの値を保持したまま集計する」というウィンドウ関数の特徴をAVG関数で利用すると、移動平均を取得できます。いわゆる平均では、存在するレコードの値をすべて合計し、レコード数で割ります。一方、移動平均では、対象のレコードを「ずらしながら」平均を求めます。

移動平均を計算するときの範囲は分析の目的によって異なりますが、ここでは「3項移動平均」でイメージをつかみます。3項移動平均とは、1レコード前、自分のレコード、1レコード後の3つのレコードの値の平均です。以下の［8-4-5］では、年月（year_month）ごとの販売金額（revenue）について、3項移動平均を［moving_avg］フィールドで求めています。

留意してほしいのは、どの行に対しても、そのルールで計算するということです。[moving_avg] の2行目の「7000」は（3000＋6000＋12000）÷3で計算されていますが、3行目の「9000」は1行下に移動して（6000＋12000＋9000）÷3で計算されています。

8-4-5 月ごとの販売金額と移動平均

year_month	revenue	moving_avg
2020-01	3000	4500
2020-02	6000	7000
2020-03	12000	9000
2020-04	9000	10000
2020-05	9000	11000
2020-06	15000	12000

平均を取る範囲が移動していくため、移動平均と呼びます。英語では「Moving Average」と呼びます。

移動平均は、月ごとの販売金額の推移などを折れ線グラフとして可視化するとき、ジグザグ感の強い急な増減を、もう少し緩やかなトレンドとして表現したいときによく利用されます。

上記では「直前の行、現在の行、直後の行」の3レコードを平均の範囲としましたが、その範囲は自由に変えることができます。例えば、B2Bのビジネスをしているwebサイトの日別セッション数は、平日は多く、土日は少ない周期型がほとんどです。よって、直前の7日間を範囲とすることで、どの範囲にも週末の2曜日（土日）が含まれるような移動平均が描けます。

では、演習用の［sales］テーブルから、実際に移動平均を求めてみましょう。以下の［8-4-6］は、［product_id］が「1」の商品について、2017年の月ごとの販売金額の3項移動平均を求めるSQL文です。移動平均の範囲は「現在のレコードの1つ前、現在のレコード、現在のレコードの1つ後ろ」の3レコードとしています。

WITH句では、月（year_month）別、商品（product_id）別に販売金額（revenue）を合計し、年を2017年、product_idを「1」に絞り込みます。

14〜16行目で集計分析関数のAVG関数を利用しています。16行目の「ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING」が、今回のお題である「現在のレコードの1つ前、現在のレコード、現在のレコードの1つ後ろ」の移動平均の範囲を示しています。

8-4-6 特定商品の販売金額の移動平均を求める

```
1  WITH master AS (
2  SELECT
3  DATETIME_TRUNC(date_time, month) AS year_month
4  , product_id
5  , SUM(revenue) AS sum_rev
6  FROM sample.sales
7  WHERE FORMAT_DATETIME("%Y", date_time) = "2017"
8  AND product_id = 1
```

```
9   GROUP BY year_month, product_id
10  )
11  SELECT
12  FORMAT_DATETIME("%Y-%m", year_month) AS year_month
13  , sum_rev
14  , AVG(sum_rev) OVER (
15  ORDER BY year_month
16  ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
17  ) AS moving_avg_rev
18  FROM master
```

結果テーブル

行	year_month	sum_rev	moving_avg_rev
1	2017-01	8000	12000.0
2	2017-02	16000	10000.0
3	2017-03	6000	13200.0
4	2017-04	17600	13133.333333333334
5	2017-05	15800	17666.666666666668
6	2017-06	19600	24266.666666666668
7	2017-07	37400	23600.0
8	2017-08	13800	19733.333333333332
9	2017-09	8000	13266.666666666666
10	2017-10	18000	14466.666666666666

商品ID「1」の商品について、月ごとの販売金額の移動平均を取得できた

なお、移動平均を求めるときには、データがないときの処理に注意してください。例えば、結果テーブルの1行目（[year-month]が「2017-01」）には「現在のレコードの1つ前のレコード」が存在しません。

その場合、足りないデータは考慮せずに平均する仕様なので、現在のレコードの［sum_rev］と1レコード後ろのレコードの［sum_rev］を足して「2」で割り、(8000＋16000）÷2＝12000として移動平均を求めます。

移動平均では、パーティションの上限や下限の近くでデータが足りないことがありますので、どのような計算方法で結果が導かれるのかは認識しておいてください。

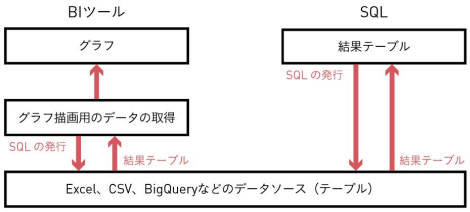
STEP UP

BIツールとSQL

本書を手に取ったみなさんは、当然ながら「ビジネスデータの分析」に興味がある、もしくは、データ分析を仕事にされているのではないかと思います。よって、TableauやGoogleデータポータルといったBIツールに、以前から触れている人も多いのではないでしょうか。

筆者も日常的にBIツールを利用していますが、ここでBIツールとSQLの関係性を紹介しましょう。以下の図［8-4-7］を見てください。

8-4-7 BIツールとSQLの関係

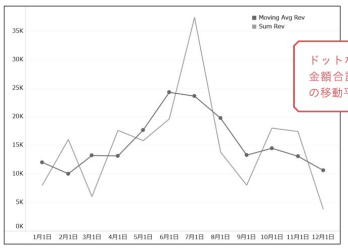


BIツールは「グラフなどで分かりやすくデータを視覚化」してくれるものですが、内部的では、SQLのような命令をデータに対して発行し、戻り値の結果テーブルを利用してグラフを描画しています。つまり、BIツールの内部でSQLが動いているといえます。一方、SQLはグラフではなく、データが羅列されたテーブルのみを得ます。

次ページのグラフ［8-4-8］は、先ほど［8-4-6］として示した移動平均の結果テーブルを元に、Tableauで視覚化したものです。ドットのないラインがオリジナル、つまり商品ID「1」の月ごとの販売金額合計の推移で、ドットありのラインが移動平均となっています。

移動平均では月ごとの変動幅が緩やかになり、大局的なトレンドが描かれていることが分かります。オリジナルとの違いは明らかであり、BIツールを利用すると、より直感的な理解が進むことは間違いありません。

8-4-8 Tableauで視覚化した移動平均



BIツールとSQLには［8-4-7］で示した関係性があるため、SQLを記述できなくても、BIツールがあればデータ分析ができるのは事実です。そのため、筆者が登壇したセミナーでは「BIツールが使えれば、SQLは覚える必要はないのではないか？」という質問を受講者から受けることがあります。

しかし、CHAPTER 1でも説明した通り、SQLが記述できて、さらにBIツールが使えることによって分析力・表現力が高まるので、**BIツールが使えてもなお、SQLを学ぶ意義は大きい**と筆者は考えています。

SQLが記述できれば、BIツールがなくても分析ができますし、BIツールについても、勘違いの少ない効率的な利用ができます。また、データの整形や前処理はSQLで、視覚化の部分はBIツールで、といった使い分けもできるようになり、データ分析者としての引き出しが大きく広がるはずです。

SECTION

8-5 確認ドリル

問題 026

[sales] テーブルから、年別の各月の販売金額（revenue）がどれだけバラついているか、「範囲」の比較で確認することにしました。「範囲」とは、最大値マイナス最小値で求めることができます。年ごとの月別の販売金額の範囲を、年の昇順に並べ替えて取得してください。結果テーブルは、年（year）と範囲（hani）の2カラムとします。

問題 027

[products] テーブルから、商品カテゴリ（product_category）ごとにコストが高い商品名（product_name）のトップ3を取得してください。結果テーブルは、商品カテゴリ（product_category）、商品名（product_name）、コスト（cost）、コストのランキング（cost_rank）の4カラムとしてください。[cost_rank] には、高い順に並べたとき、もっとものコストの高い商品に「1」を格納します。

問題 028

[sales] テーブルで月別・商品別に販売金額（revenue）を合計すると、特定の月における商品別の販売金額シェアが分かります。例えば、2017年1月の販売金額合計は25,600円、そのうち商品ID「1」は8,000円なので、その月における販売金額シェアは31.25%（8000÷32500）となります。

上記の通りに計算したとき、商品別の販売金額シェアが40%を超えた月（year_month）と商品ID（product_id）、月内の販売金額シェア（monthly_revenue_share）の3カラムを取得してください。結果テーブルは、該当商品の月内の販売金額シェアが高い順に並べ替えてください。

なお、結果テーブルで取得する月は「YYYY-MM形式」で整えます。販売金額シェアには「%」記号は不要で、小数での表示で構いません。

問題 029

どの顧客にも、初めて購入した商品があります。[sales] テーブルから商品ごとに、その商品がその顧客にとって初めて購入した商品であった顧客数（number_of_users）を取得してください。

初回の注文で2つ以上の商品を購入した場合は、商品ID（product_id）が小さいほうの商品を初めて購入した商品と見なします。結果テーブルは初回購入商品（first_purchase_product）と、その商品を初回に購入した顧客数（number_of_users）の2カラムとし、[number_of_users] が多い順に並べ替えて、トップ5に絞り込んでください。

問題 030

セッションの最初に閲覧されたWebページを「ランディングページ」、最後に閲覧されたWebページを「離脱ページ」と呼びます。[web_log] テーブルからランディングページと離脱ページの組み合わせ（landing_and_exit）と、セッション数（session）のトップ5を取得してください。セッション中に1つのページしか見ない（ランディングページと離脱ページが同一）の「直帰」のセッションは除きます。

なお、セッションはブラウザーを識別する［cid］と、何回目の訪問かを示す整数値の［session_count］の組み合わせで決まります。ランディングページと離脱ページの組み合わせ（landing_and_exit）は、それぞれのページの文字列を、矢印を表す「->」で連結した文字列としてください。

解答

解答 026

```
WITH master AS (
SELECT FORMAT_DATETIME("%m", date_time) AS month
, FORMAT_DATETIME("%Y", date_time) AS year
, SUM(revenue) AS sum_rev
FROM sample.sales
GROUP BY month, year)

SELECT year, MAX(max_rev) - MAX(min_rev) AS hani
FROM (
SELECT year
, PERCENTILE_CONT(sum_rev, 0)
OVER (PARTITION BY year) AS min_rev
, PERCENTILE_CONT(sum_rev, 1)
OVER (PARTITION BY year) AS max_rev
FROM master
)
GROUP BY year
ORDER BY year
```

結果テーブル

行	year	hani
1	2017	67060.0
2	2018	62320.0
3	2019	72180.0

まず、年別の各月の販売金額合計を、仮想テーブルとしてWITH句で作成しています。メインのクエリの内側では、年ごとの最小値と最大値を求めたいので、PERCENTILE_CONT関数の後ろのOVER句の中のPARTITION BY句に「year」を設定しています。

その後、メインのクエリの外側では［year］ごとにグループ化し、MAX関数を利用して1つの年が1レコードになるように整形しています。

解答 027

```
SELECT * FROM (
SELECT product_category, product_name, cost
, RANK() OVER (PARTITION BY product_category
ORDER BY cost DESC) AS cost_rank
FROM sample.products)
WHERE cost_rank <= 3
```

結果テーブル

行	product_category	product_name	cost	cost_rank
1	クッキー	クッキー詰め合わせ	600	1
2	クッキー	ガレット	300	2
3	クッキー	アイスボックスクッキー	300	2
4	ケーキ	ケーキ詰め合わせ	1000	1
5	ケーキ	ガトーショコラ	800	2
6	ケーキ	ショートケーキ	600	3
7	ゼリー	ゼリー詰め合わせ	1000	1
8	ゼリー	オレンジゼリー	300	2
9	ゼリー	グレープゼリー	300	2
10	ゼリー	洋ナシゼリー	300	2

RANK関数のパーティションは「○○ごとの」と考えます。よって、PARTITION BY句には［product_category］を指定します。［cost］を高い順に並べ替えるため、ORDER BY句のオプションは「DESC」です。

ただし、WHERE句でRANK関数の戻り値を利用してトップ3を実現しようと、以下のようなSQLを記述してもエラーになります。内部的にWHERE句がウィンドウ関数より前に実行されるため、WHERE句にウィンドウ関数の戻り値を記述しても認識しないためです。

```
SERECT product_category, product_name, cost
, RANK() OVER (PARTITION BY product_category
ORDER BY cost DESC) AS cost_rank
FROM sample.products
WHERE RANK() OVER (PARTITION BY product_category
ORDER BY cost DESC) <= 3
```

したがって、内側のクエリではWHERE句を記述せず、絞り込みをしないで全レコードを出力し、外側のクエリのWHERE句にて「cost_rank <= 3」と指定しています。

解答 028

```
WITH master AS (
SELECT FORMAT_DATETIME("%Y-%m", date_time)
AS year_month
, product_id
, SUM(revenue) AS sum_rev
FROM sample.sales
GROUP BY year_month, product_id
)

SELECT year_month, product_id, monthly_revenue_share
FROM (
```

```
12  SELECT
13  year_month, product_id
14  , sum_rev / SUM(sum_rev) OVER (PARTITION BY year_month
15  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
16  ) AS monthly_revenue_share
17  FROM master
18  )
19  WHERE monthly_revenue_share > 0.4
20  ORDER BY 3 DESC
```

結果テーブル

行	year_month	product_id	monthly_revenue_share
1	2017-11	1	0.4494962541978817
2	2017-06	1	0.41332770982707717
3	2017-07	1	0.41271242551313175
4	2018-12	1	0.4069494443574894

WITH句で月別・商品ID別の販売金額合計を仮想テーブルとして作成しています。メインのクエリ内側（12〜17行目）では、商品別の販売金額シェア（monthly_revenue_share）を求めるために、商品別の販売金額をその月の販売金額合計で割った結果を取得しています。月の販売金額を合計するために、PARTITION BY句の後ろに「year_month」がきていることに注目してください。

また、外側のクエリでは、月内の販売金額シェアが40%を超える商品だけに絞り込み、シェアの高い順に並べ替えて表示しています。

解答 029

```
SELECT first_purchase_product
, COUNT(DISTINCT user_id) AS number_of_users
FROM (
SELECT user_id
, MAX(first_product) AS first_purchase_product
FROM (
SELECT user_id, product_id
, FIRST_VALUE(product_id) OVER (PARTITION BY user_id
ORDER BY date_time, product_id) AS first_product
FROM sample.sales
)
GROUP BY user_id
)
GROUP BY first_purchase_product
ORDER BY 2 DESC
LIMIT 5
```

結果テーブル

行	first_purchase_product	number_of_users
1	1	182
2	2	84
3	15	43
4	14	34
5	5	33

いちばん内側のクエリ（7〜10行目）のポイントは、「顧客ごとに購入日時が同じ場合、商品ID（product_id）が小さい商品を初回に購入したと見なす」処理です。

［sales］テーブルには1人の顧客（user_id）について、同一の日時（date_time）を持つレコードが複数存在する場合があります。それは1つの注文（order_id）で、複数の商品が同時に購入されたことを表します。

その場合、もっとも商品IDの小さい商品をパーティションのいちばん上に並べ替える必要があります（9行目）。OVER句にあるORDER BY句に［date_time］と［product_id］を記述することにより、まず購入日時の古い順（小さい順）に並べ、さらに同一パーティション内に同一の購入日時があれば、商品IDを小さい順に並べる処理をしています。

1つ外側のクエリでは、1人の顧客に対して複数レコードが存在する状態を解消するために、［user_id］をグループ化し、MAX関数で［first_product］を集計しています。［first_product］は1ユーザーに対して1種類の値しか存在しないので、MAX関数、MIN関数のどちらでも同じ戻り値となりますが、ここではMAX関数を利用しています。

また、最終的な結果テーブルがより分かりやすくなるよう、フィールド名を［first_purchase_product］に変更しました。もっとも外側のクエリでは［first_purchase_product］ごとに、その商品を最初に購入した顧客が何人いるのかを集計し、顧客数の多い順に並べ替えて、5人に絞り込んでいます。

解答 030

```
1  WITH master AS (
2  SELECT cid, session_count
3  , FIRST_VALUE(page)
4  OVER (PARTITION BY cid, session_count
5  ORDER BY date_time) AS l_page
6  , LAST_VALUE(page)
7  OVER (PARTITION BY cid, session_count
8  ORDER BY date_time
9  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
10 ) AS e_page
```

```
11  FROM sample.web_log
12  )
13
14  SELECT
15  concat(l_page, " -> ", e_page) AS landing_and_exit
16  , COUNT(*) AS session
17  FROM (
18  SELECT cid, session_count, MAX(l_page) AS l_page,
19  MAX(e_page) AS e_page
20  FROM master
21  GROUP BY cid, session_count
22  HAVING COUNT(*) > 1
23  ORDER BY 1, 2
24  )
25  GROUP BY l_page, e_page
26  ORDER BY 2 DESC
27  LIMIT 5
```

結果テーブル

行	landing_and_exit	session
1	/ -> /special/diet/	33
2	/ -> /prod/prod_id_1/	19
3	/ -> /	19
4	/ -> /cart/	16
5	/ -> /thank_you/	14

WITH句では、ランディングページと離脱ページを取得しています。セッションは［cid］と［session_count］の組み合わせで決まるので、その2つのフィールドをパーティションにしたうえで、パーティション中のレコードを［date_time］の昇順に並べ替えます。

パーティションの上端の［page］の値をFIRST_VALUE関数で取り出してランディングページ（l_page）とし、パーティションの下端の［page］の値をLAST_VALUE関数で取り出して離脱ページ（e_page）として取得します。

WINDOWフレーム句を省略した場合、パーティションの上端から現在のレコードまでを示す「ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW」が適用されるため、ランディングページを取り出すためのFIRST_VALUE関数では省略しています。

一方、離脱ページを取り出すためのLAST_VALUE関数では、パーティションの下端の値が必要です。WINDOWフレーム句として「ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING」を記述して、明示的にLAST_VALUE関数がパーティションの下端の値を取得できるようにしています。

メインのクエリでは、内側のクエリで［cid］と［session_count］をグループ化し、MAX関数で［l_page］と［e_page］を集計しています。

そのうえで、［cid］と［session_count］でグループ化したときのレコード数（ページビュー数）が「1より大きい」条件で絞り込むことにより、直帰を除外しています。

もっとも外側のクエリは、文字列を連結するCONCAT関数で「(ランディングページ) → (出口ページ)」という文字列を生成して、レコード数を集計しています。すでに内側のクエリで［cid］と［session_count］でグループ化されており、各レコードは「セッション」を表すため、レコード数をカウントすれば、セッション数を取り出せるというわけです。

以上で、学習の本編は終わりです。ここまで学んだみなさんは、実務でビジネスデータを分析するうえで必要となる知識を、ほぼすべて習得されていると思います。あとは練習あるのみです。次章では、たくさん手を動かして学んだことを試してください。お疲れさまでした！

PRACTICE

演習ドリル

P-0　演習ドリルへの取り組み方
P-1　第1群
P-2　第2群
P-3　第3群
P-4　第4群
P-5　第5群

本章は、これまでに学んだSQLのスキルを駆使して、200問の問題を解いていただく演習ドリルとなります。頭では分かったつもりでも、いざ実務になると手が動かない。そのようなことを避けるため、章末の確認ドリルとの合計で230問というボリュームになっています。力試しも兼ねて、奮って取り組んでください。

SECTION

P-0 演習ドリルへの取り組み方

8つの章にまたがる学習、お疲れさまでした。しかし、SQLの習得は、読んだり聞いたりするだけでは完結しません。そのためには……

「手を動かして実践する」ことが大事なんですよね。

はい。その実践となる演習ドリルへの取り組み方について、最初にお伝えしたいことをまとめます。

解答の参照方法について

演習ドリルの解答は、本書の紙面ではなく、**インプレスのWebメディア「できるネット」に掲載**しています。スマートフォンの場合、以下のQRコードを読み取ってサポートページにアクセスし、各問題の番号に張られたリンクから解答のページを表示していただくと便利かと思います。

PCの場合、もしくはスマートフォンでのURL入力が苦にならない場合、以下の「問題031の解答ページ」の例のように、末尾に問題番号が付いたURLを入力することで該当する解答のページに直接アクセスできます。みなさんの環境に応じて、参照しやすい方法を選んでください。

本書のサポートページ
https://dekiru.net/dc_sql

問題031の解答ページ
https://dekiru.net/dc_sql_031

解答となるSQL文について

できるネットの解答ページには、各章末の確認ドリルの解答と同様に、実行すべきSQL文と、その結果テーブルを掲載しています。このうち、SQL文については、**筆者が記述した「正解を得る解答例の1つ」であって、「唯一の解答」や「ベストの解答」ではない可能性がある**ことに留意してください。

正解が含まれる結果テーブルを表示するためのSQL文は、1つとは限りません。問題によっては、複数の記述方法で同じ結果が得られる場合があります。厳密に言えば、結果テーブルが返ってくる時間が短い、SQL文そのものが短い、誰が見ても読み解きやすいSQL文になっているなど、重視するポイントによってSQL文の優劣に差が出る場合があります。

こうしたポイントは、エンジニアであれば当然気にする点ではありますが、私たちの目的が「自分自身の業務のためのデータ分析」であることを考えると、それほどこだわる必要がありません。よって、どのような記述方法でも、同じ正解にたどり着ければよいことにしましょう。

みなさんが記述したSQL文が、解答ページにあるSQL文とは異なっていたとしても、同じ正解を含む結果テーブルを表示できたのであればOKです。自信をもって次の問題に進んでください。

問題の「群」と「HINT」について

本章の演習ドリルは以下の5つの群に分かれており、それぞれの群は「どの章で学んだ知識で解けるか？」という観点でリンクしています。例えば、第1群はCHAPTER 3と7の知識で解けますが、第5群はすべての章で学んだ知識を総動員しないと解くことができません。

- 第1群　⇒CHAPTER 3（SQLの基本構文）、7（さまざまな関数）
- 第2群　⇒CHAPTER 3、7＋4（グループ化とデータの集計）
- 第3群　⇒CHAPTER 3、4、7＋5（複数テーブルの結合と集合演算）
- 第4群　⇒CHAPTER 3、4、5、7＋6（サブクエリ）
- 第5群　⇒CHAPTER 3、4、5、6、7＋8（ウィンドウ関数）

また、各問題には「HINT」として、関連する「CHAPTER - SECTION」の番号を記載してあります。**どの章・節で学んだことが出題されているのか、群とHINTからたどれる**ようにしていますので、参考にしてください。章ごとの復習を終えるたびに、それに該当する群の問題を解いていくのも、本書の良い使い方かと思います。

なお、CHAPTER 3〜8の章末には確認ドリルとして各5問、合計30問がありましたので、本章の演習ドリルは問題番号「031」から始まります。問題は「易しい順」に並べているため、一部、似たような内容の問題が連続する箇所がありますが、ご容赦ください。

解答が分からなかったときは

最後に、もし正解の結果テーブルにたどり着けなかった場合は、解答ページにあるSQL文を分析してみてください。「どのような仮想テーブルを使っているのか」「どのようなグルーピングを行った後に別のテーブルとJOINしているのか」など、筆者が記述したSQL文を読み解くことによって、正解を求めるための道筋や考え方を学ぶことができるはずです。

他人が記述したSQL文を分析するうえでは、いくつかのコツがあります。まず、WITH句が使われている場合は、そこから分析して、どのような仮想テーブルを取り出しているのかを捉えてください。WITH句を、例えば［master］という名前で記述した後、「SELECT * FROM master」をメインのクエリで実行して、想定通りの仮想テーブルを作り出せているかを確認するのも有効です。

そのうえで、SQL文がサブクエリによって入れ子になっている場合は、内側のSELECT文から読み解いていきます。加えて、問題に付記したHINTを参照して、それぞれの内容を復習してもらえればと思います。

演習ドリルは、マーケティングや営業、商品企画などの業務におけるデータ分析でみなさんが直面しそうな、より実践的なシチュエーションを想定して作成しています。業務の合間に少しずつチャレンジしてみてください。

SECTION

P-1 第1群

問題 031 HINT ➡ 3-1, 7-1

［customers］テーブルから『「山」で始まって「子」で終わる』名前（name）の顧客のレコードを全カラム取得してください。

問題 032 HINT ➡ 3-2, 3-3

［customers］テーブルから、顧客をランダムに3人選択し、すべてのカラムを取得してください。

問題 033 HINT ➡ 3-1, 7-5

［sales］テーブルから、［quantity］と［revenue］の相関係数を求めてください。結果テーブルは［corr］というカラム名としてください。

問題 034 HINT ➡ 3-1, 3-2

［products］テーブルから、原価（cost）の標準偏差を求めてください。ただし、［products］テーブルはサンプルではなく、全数（母集団）と考えます。

問題 035 HINT ➡ 3-1, 4-2

［sales］テーブルから、もっとも古い日付・時刻（date_time）を取得してください。カラム名は［oldest_datetime］とします。

問題 036 HINT ➡ 3-2, 7-2

［sales］テーブルから、もっとも高額な［revenue］が記録されているレコードの［order_id］［user_id］［revenue］を取得してください。

問題 037 HINT 3-1, 7-4

［sales］テーブルから、商品ID（product_id）が「5」で、販売金額（revenue）が8,000円を超えるレコードのすべてのカラムを取得し、［order_id］の小さい順に表示してください。

問題 038 HINT 3-1, 3-2

［web_log］テーブルから、メディア（media）として「cpc」が最初に記録された［date_time］を取得してください。

問題 039 HINT 3-1, 3-2

［sales］テーブルで、初めて値引き販売が記録された［date_time］を取得してください。

問題 040 HINT 3-1, 3-2

［customers］テーブルから、最高齢の顧客の誕生日（birthday）を取得してください。

問題 041 HINT 3-1, 3-2

［product］テーブルから、「ゼリー」のカテゴリを除いたレコードの［cost］の分散を求めてください。なお、データは全数（母集団）であり、サンプルではありません。結果テーブルのカラム名は［var_cost］とします。

問題 042 HINT 3-1, 3-2

［sales］テーブルから、「3」でも「7」でも割り切れる［order_id］で注文をした顧客のうち、もっとも小さい［user_id］を取得してください。

問題 043 HINT 3-3, 7-5

［sales］テーブルに記録されているもっとも新しい日付（date_time）を取得してください。結果テーブルのカラム名は［newest_date］とし、YYYY-MM-DD形式で表示してください。

問題 044 HINT ➡ 3-2, 3-3

［customers］テーブルから、『名前（name）が「子」で終わる』もっとも年齢が高い顧客のレコードを全カラム取得してください。

問題 045 HINT ➡ 3-2, 7-1

［customers］テーブルのうち、誕生日（birthday）に会員登録をした顧客が1人だけいます。その顧客のレコードを全カラム取得してください。

問題 046 HINT ➡ 3-1, 7-4

［customers］テーブルから、都道府県（prefecture）が「福岡」「佐賀」「長崎」のいずれかに該当するプレミアム顧客のリストを全カラム取得してください。結果テーブルは［user_id］の小さい順に並べ替えて表示します。

問題 047 HINT ➡ 3-1, 3-3

［sales］テーブルから、2017年2月1日から2月10日までの期間で、もっとも販売金額（revenue）が大きいレコードを全カラム取得してください。

問題 048 HINT ➡ 3-1, 3-2

［customers］テーブルから、姓が「山」で終わる男性顧客の顧客ID（user_id）と氏名（name）を取得してください。

問題 049 HINT ➡ 3-2, 7-3

［customers］テーブルに登録されている顧客の中に1人だけ、誕生日（birthday）は「null」だが、会員登録日（register_date）は判明している人がいます。その顧客の名前（name）、誕生日（birthday）、会員登録日（register_date）を取得してください。

問題 050 HINT ➡ 3-2, 3-3

［products］テーブルのうち、カテゴリ（product_category）は「クッキー」だが、商品名（product_name）が「クッキー」で終わらないレコードを全カラム取得してください。

問題 051 HINT ➡ 3-3, 7-2

［customers］テーブルから、もっとも若い顧客の「現在の年齢」を求めてください。結果テーブルのカラム名は［current_age］とします。

問題 052 HINT ➡ 3-2, 7-4

［customers］テーブルに記録されている最年少の顧客は、2019年12月31日時点で何歳か求めてください。結果テーブルのカラム名は［youngest_age］とします。

問題 053 HINT ➡ 3-2, 7-2

［products］テーブルから、商品カテゴリ（product_category）と商品名（product_name）をハイフンでつないだ文字列（例：ケーキ-レアチーズケーキ）を表示する［cat_prod］カラムと［cost］カラムを取得してください。結果テーブルは［cost］の高い順に3レコードだけ表示します。

問題 054 HINT ➡ 3-2, 7-4

［sales］テーブルから、商品ID（product_id）が「7」「9」「11」のいずれかで、販売個数（quantity）が「2」以上、かつ、値引販売されているレコードを取得してください。結果テーブルは［order_id］の小さい順に並べ替えて、全カラムを取得します。

問題 055 HINT ➡ 3-2, 3-3

［customers］テーブルから、月曜日に生まれた、もっとも年齢の高い顧客について、顧客ID（user_id）、氏名（name）、誕生日（birthday）を取得してください。

問題 056 HINT ➡ 3-2, 7-4

［customers］テーブルから、「川」で始まる顧客名（user_name）と、誕生日（birthday）を取得してください。結果テーブルは年齢の高い順に並べ替え、上位3人に絞り込んで表示してください。

問題 057 HINT ➡ 3-2, 7-4

［sales］テーブルから、1月1日に販売された注文をすべて取得してください。全カラムを取得し、並べ替えは［order_id］の小さい順とします。

問題 058 HINT ➡ 3-2, 7-2

［customers］テーブルから、氏名（name）の後ろにその顧客が男性ならば「様」、女性ならば「さま」という文字列を付与したカラム［name_sama］を取得してください。結果テーブルは登録日（register_date）の新しい順に、5人の顧客に絞り込んで表示してください。

問題 059 HINT ➡ 3-2, 7-2

［products］テーブルから、商品名（product_name）の末尾が「ゼリー」で終わっている場合、「ゼリー」を「ジュレ」に変更して、［modified_name］というカラム名で取得してください。結果テーブルは、元の商品名である［product_name］と、［modified_name］の2カラムとします。

問題 060 HINT ➡ 3-2, 7-2

［customers］テーブルから、姓も名も3文字の顧客のリストを取得してください。結果テーブルは［user_id］と［name］の2カラムとします。

問題 061 HINT ➡ 3-2, 7-3

［web_log］テーブルの［page］カラムに記録されたWebページのうち、クエリパラメータ「?sys=abc123」が付与されているものがあります。分析に不向きなことがあるため、クエリパラメータを削除した文字列をクエリパラメータ除外ページ（page_without_parameter）として取得してください。結果テーブルは、元のWebページ名を格納する［page］と、クエリパラメータを削除した［page_without_parameter］の2カラムとします。元の［page］がクエリパラメータ付きであったWebページに絞り込み、［date_time］の小さい順に3レコードだけ表示してください。

問題 062 HINT 3-2, 4-2

［web_log］テーブルの［page］カラムに格納されている「/prod/prod_id_［数字］/」の［数字］部分は、商品ID（product_id）を示しています。［page］から［数字］だけを抜き出し、［product_id］カラムとして取得してください。結果テーブルは［page］と［product_id］の2カラムとし、［date_time］の小さい順に並べ替えて、5レコードに絞り込んで表示します。

問題 063 HINT 3-2, 7-2

［products］テーブルから、商品カテゴリ（product_category）と商品名（product_name）をハイフンでつないだ文字列（例：「ケーキ-レアチーズケーキ」）を［cat_prod］カラムとして、また、その文字列の長さを［length］というカラム名で取得してください。結果テーブルは［length］の長い順に3レコードに絞り込んで表示します。

問題 064 HINT 3-2, 7-4

［customers］テーブルから［name］［prefecture］［register_date］、そして登録日から10年後の日付を［aniversary10］カラムとして取得してください。ただし、都道府県（prefecture）が「千葉」の男性で、登録日（register_date）が「null」でない顧客に絞り込んでください。並べ替えは登録日の古い順とします。

問題 065 HINT 3-3, 7-4

［customers］テーブルから、会員登録したときの年齢がもっとも若い顧客について、顧客ID（user_id）、氏名（name）、誕生日（birthday）、会員登録日（register_date）、会員登録時年齢（register_age）を取得してください。

SECTION

P-2 第2群

問題 066　HINT ➡ 7-2, 7-3

［web_log］テーブルから、Webサイトのページビュー数を求めてください。

問題 067　HINT ➡ 3-2, 4-2

［products］テーブルから、原価（cost）の平均金額を求めてください。カラム名は［avg_cost］とします。

問題 068　HINT ➡ 3-2, 4-2

［products］テーブルから、原価（cost）の最高金額（最大値）を求めてください。カラム名は［max_cost］とします。

問題 069　HINT ➡ 3-2, 4-2

［sales］テーブルから、最大の販売金額（revenue）を求めてください。カラム名は［max_revenue］とします。

問題 070　HINT ➡ 3-2, 4-2

［sales］テーブル全体の販売金額（revenue）の合計を求めてください。

問題 071　HINT ➡ 3-2, 4-2

［sales］テーブル全体の販売個数（quantity）の合計を求めてください。カラム名は［sum_quantity］とします。

問題 072　HINT ➡ 3-2, 4-2

［web_log］テーブル全体のユーザー数を求めてください。ユーザーは固有の［cid］で判定します。カラム名は［users］とします。

問題 073 HINT ➡ 3-2, 4-2

［sales］テーブル全体のユニークな注文数を求めてください。カラム名は［orders］とします。

問題 074 HINT ➡ 3-2, 4-2

［sales］テーブル全体で、何人の顧客に対して販売しているかを求めてください。カラム名は［users］とします。

問題 075 HINT ➡ 3-2, 4-2

［web_log］テーブルから［date_time］の最大値を教えてください。

問題 076 HINT ➡ 3-2, 4-2

［sales］テーブル全体で、何種類の商品（product_id）が販売されているかを求めてください。カラム名は［items］とします。

問題 077 HINT ➡ 3-2, 4-2

［sales］テーブルに記録されている販売金額（revenue）の金額が税込だったとします。テーブル全体の［revenue］の合計が税抜でいくらになるか、求めてください。消費税率は期間全体で10%とします。結果テーブルのカラム名は［revenue_without_tax］としてください。

問題 078 HINT ➡ 3-2, 4-2

［web_log］テーブルに記録されているユニークな［media］の個数を求めてください。結果テーブルは［number_of_media］というカラム名で取得します。

問題 079 HINT ➡ 3-2, 4-2

［sales］テーブルから、商品ID（product_id）が「1」の商品の販売金額（revenue）の合計を求めてください。結果テーブルは［sum_revenue_1］というカラム名とします。

問題 080 HINT 3-2, 4-2

［web_log］テーブルから［session_count］が「1」(新規訪問）のページビュー数の合計を求めてください。結果テーブルのカラム名は［pv_new_visit］とします。

問題 081 HINT 3-2, 4-2

［products］テーブルにユニークな商品カテゴリ（product_category）がいくつあるか求めてください。結果テーブルのカラム名は［categories］とします。

問題 082 HINT 3-2, 4-2

［products］テーブルから、原価（cost）が300円以下の商品の個数を求めてください。結果テーブルのカラム名は［items］とします。

問題 083 HINT 3-2, 4-2

［sales］テーブルから、販売金額（revenue）合計を全顧客数で割って、顧客の平均販売金額を求めてください。結果テーブルのカラム名は［avg_rev_by_user］とします。

問題 084 HINT 3-2, 4-2

［web_log］テーブルの最大の［date_time］から最小の［date_time］を差し引いて、期間を日単位で表示してください。結果テーブルのカラム名は［days］とします。

問題 085 HINT 3-2, 4-2

［web_log］テーブルから、デバイス別のページビュー数を求めて下さい。結果テーブルは［device］［pageviews］の2カラムとし、［pageviews］の降順に並べ替えて表示してください。

問題 086 HINT 3-2, 4-2

［customers］テーブルに都道府県（prefecture）が「東京」以外の顧客は何人いるか調べてください。結果テーブルのカラム名は［users］とします。

問題 087 HINT 3-3, 4-2

［customers］テーブルで、会員登録日（register_date）が「null」の顧客の人数を求めてください。結果テーブルのカラム名は［users］とします。

問題 088 HINT 4-2, 7-4

［web_log］テーブルに、何日の固有な日が記録されているかを調べてください。結果テーブルのカラム名を［unique_days］とします。

問題 089 HINT 4-1, 4-2

［web_log］テーブルから、ページ（page）別のページビュー数を取得し、ページビュー数の多い順に、トップ5のページ（page）を表示してください。パラメータ「?sys=abc123」が付いているページと、付いていないページは区別して集計します。結果テーブルは［page］と［pageviews］の2カラムとします。

問題 090 HINT 4-2, 7-2

［web_log］テーブルに記録された「/prod/prod_id_8/」のページビュー数を求めてください。クエリパラメータ「?sys=abc123」の有無にかかわらずカウントします。結果テーブルのカラム名は［pageviews_8］とします。

問題 091 HINT 3-2, 4-2

［sales］テーブルから、商品ID（product_id）が「14」もしくは「15」の商品の販売個数（quantity）の合計を求めてください。結果テーブルのカラム名は［sum_qty_14_15］とします。

問題 092 HINT 3-2, 4-2

［sales］テーブルから、販売金額（revenue）の合計を注文数で割って、注文あたりの平均販売金額を求めてください。結果テーブルのカラム名は[avg_rev_by_order]とします。[avg_rev_by_order］は十の位で切り捨て、百円単位に丸めます。

問題 093 HINT 4-2, 7-4

［sales］テーブルから、もっとも高額な注文［order_id］とその注文の販売金額（revenue）の合計（sum_rev）を取得してください。結果テーブルのカラムは［order_id］と［sum_rev］の2カラムとします。

問題 094 HINT 3-2, 4-2

［sales］テーブルに記録されている販売金額（revenue）が税抜だったとします。テーブル全体の［revenue］が税込でいくらになるか求めてください。消費税率は、2019年10月1日以前は8%、以降は10%とします。カラム名は［revenue_with_tax］とします。

問題 095 HINT 3-2, 4-2

［sales］テーブルに記録された2019年の販売金額（revenue）の合計を求めてください。結果テーブルのカラム名は［sum_revenue_2019］とします。

問題 096 HINT 3-2, 4-2

［products］テーブルから、原価（cost）別に商品の個数を取得してください。結果テーブルは［cost］と商品数［number_of_products］の2カラムとします。なお、原価（cost）が500円未満の商品に絞り込んで表示してください。

問題 097 HINT 4-2, 7-3

［customers］テーブルの誕生日（birthday）も、会員登録日（register_date）も「null」の顧客は何人いるか求めてください。結果テーブルのカラム名は［users］とします。

問題 098 HINT 4-2, 7-4

［web_log］テーブルに記録されたWebページ（page）が、「/prod/prod_id_［数字］/」に該当するページビュー数を求めてください。「数字」は「1」から「15」まで存在します。結果テーブルのカラム名は［pageviews_prod］とします。なお、ページには「/prod/prod_id/15/?sys=abc123のように、末尾に「?sys=123」が存在している場合があります。そのページはカウントしないでください。

問題 099 HINT 3-2, 4-2

［sales］テーブルから「年」を問わず、1月と12月に販売された注文の個数を取得してください。結果テーブルのカラム名は［orders］とします。

問題 100 HINT 4-2, 7-1

［customers］テーブルから、1980年代生まれの顧客の数を求めてください。カラム名は［users］とします。

問題 101 HINT 4-2, 7-4

［customers］テーブルから、都道府県別の顧客数を求めてください。結果テーブルは都道府県（prefecture）、顧客の数（users）の2カラムとします。絞り込みは顧客数の多い順に3レコードとしてください。

問題 102 HINT 4-2, 4-4

［customers］テーブルに記録されている、2019年第1四半期に会員登録を行った顧客数を求めてください。結果テーブルのカラム名は［users］とします。なお、第1四半期は1〜3月とします。

問題 103 HINT 4-2, 7-4

［sales］テーブルから、販売金額（revenue）の合計をユニークな「日」（date_time）の数で割り、日別の平均売上金額を求めてください。結果テーブルのカラム名は［avg_rev_by_day］とします。［avg_rev_by_day］は十の位で切り上げして百円単位に丸めて表示してください。

問題 104 HINT 4-1, 7-3

［sales］テーブルを対象に、販売された時間を「年」と「午前または午後」でグループ化したときの販売金額を表示してください。結果テーブルは、「2017-AM」から「2019-PM」までの6つの値を持つ［sales_time］と、それらのグループごとの販売金額の合計（sum_rev）の2カラムとしてください。

問題 105 HINT 4-1, 4-2

［customers］テーブルから、都道府県（prefecture）が千葉県の女性顧客の平均年齢を求めてください。年齢は2019年12月31日時点の満年齢とします。結果テーブルのカラム名は［avg_age］とします。

問題 106 HINT 4-1, 4-2

［sales］テーブルにて、商品ID（product_id）が「1」、個数（quantity）が「1」で値引販売されたレコードは記録ミスだということが判明しました。実際には、それらのレコードは定価である1個あたり2,000円で販売されていました。販売金額（revenue）を修正し、テーブル全体の販売金額の合計を取得してください。結果テーブルのカラム名は［modified_sum_rev］とします。

問題 107 HINT 4-1, 4-2

［products］テーブルから、商品カテゴリ（product_category）ごとの原価（cost）の平均を求めてください。平均原価（avg_cost）は整数型（INT64型）に変換し、金額の大きい順に表示してください。

問題 108 HINT 4-1, 4-3

［sales］テーブルから、2019年に販売された商品ID（product_id）が「5」の商品の販売金額（revenue）の合計はいくらか求めてください。カラム名は［sum_revenue_5_2019］とします。

問題 109 HINT 4-1, 7-1

［sales］テーブルから、年ごとの平均の販売個数を求めてください。結果テーブルは［year］と［avg_qty］の2カラムとします。［avg_qty］は四捨五入で小数点以下第2位まで表示してください。

問題 110 HINT 4-1, 4-2

［sales］テーブルから、販売した年別の顧客数を求めてください。結果テーブルは、年（year）と顧客数（users）の2カラムとして、年の昇順に並べ替えます。

問題 111 HINT 4-1, 7-3

［web_log］テーブルのうち、日別のページビュー数でトップ3を求めてください。結果テーブルは、日付（date）とページビュー（pageviews）の2カラムとしてください。

問題 112 HINT 4-1, 7-4

［web_log］テーブルで、1日の時間を「00」から「23」までに分けたとき、ページビュー数の多い順にトップ3を求めてください。結果テーブルは、時間（hour）とページビュー数（pageviews）の2カラムとしてください。

問題 113 HINT 4-1, 7-4

［web_log］テーブルから、メディア（media）別のページビュー数を取得してください。ただし、対象となるWebページ（page）は、「/prod/」ディレクトリ配下を除きます。結果テーブルは［media］と［pageviews］の2カラムとし、［pageviews］の降順で表示してください。

問題 114 HINT 4-1, 7-4

［web_log］テーブルから、デバイス（divice）がモバイル（mobile）で、サイトを初回訪問したユーザー数（users）を求めてください。［users］は固有な［cid］の数のカウントで求めます。結果テーブルは［session_count］と［users］の2カラムとします。

問題 115 HINT 4-1, 7-4

［customers］テーブルから、都道府県（prefecture）別に会員登録日（register_date）が「null」の顧客（users）をカウントし、該当する顧客の多い順にトップ3の都道府県に絞り込んで表示してください。結果テーブルは［prefecture］と、［users］の2カラムとします。

問題 116 HINT 4-1, 4-2

［customers］テーブルで、都道府県別のプレミアム顧客数を調べてください。結果テーブルは、都道府県（prefecture）とプレミアム顧客数（premium_users）の2カラムとします。結果テーブルはプレミアム顧客数の多い順に3レコードに絞り込んで表示してください。

問題 117 HINT 4-1, 4-2

［customers］テーブルから、名字別に顧客が何人いるのかを求めてください。結果テーブルは、名字（last_name）と、顧客数（users）の2カラムとします。ただし、女性（gender = 2）に絞り込み、顧客数が3人以上いる名字に絞り込んで表示してください。

問題 118 HINT 4-1, 4-2

［web_log］テーブルから、月別のセッションでトップ3を求めてください。結果テーブルは、月（month）とセッション（session）の2カラムとします。ただし、月については、2017年1月、2018年1月、2019年1月のように、同じ月であれば年を問わず合算してください。月は「January」「February」など完全な月の名前で表示してください。

問題 119 HINT 4-1, 7-4

［sales］テーブルから、顧客あたり平均注文数を求めてください。結果テーブルは、顧客数（users）、注文数（orders）、平均注文数（avg_orders）の3カラムとします。なお、［avg_orders］は四捨五入して小数点以下第2位まで表示してください。

問題 120 HINT 4-1, 4-2

［customers］テーブルから、都道府県別に「プレミアム顧客、もしくは男性顧客」の数を調べてください。結果テーブルは、都道府県（prefecture）と該当する顧客数（premium_or_male_users）の2カラムとします。［premium_or_male_users］の多い順に3レコードに絞り込んで表示してください。

問題 121 HINT 4-1, 4-3

［web_log］テーブルから、1週間を曜日（Monday〜Sunday）と時間（00〜23）で分けたときのページビュー（pageviews）を求めてください。結果テーブルは、曜日（day_of_week）、時間（hour）、ページビュー数（pageviews）の3カラムとしてください。なお、ページビュー数の合計が「60」以上の曜日と時間の組み合わせに絞り込んで表示してください。

問題 122 HINT ➡ 4-1, 4-4

［products］テーブルから、原価（cost）を「300以下」「301-600」「601-900」「901以上」の4つのコストカテゴリ（cost_category）に分類したうえで、商品点数を取り出してください。結果テーブルは、コストカテゴリ（cost_category）と商品点数（number_of_products）の2カラムとします。［cost_category］の安い順に並べ替えて表示してください。

問題 123 HINT ➡ 4-1, 7-4

［customers］テーブルで、［一都三県］、［一都三県以外］を値に持つ［pref_category］ごとに年齢のばらつきを知りたいとします。［pref_category］ごとに年齢（age）の平均（ave_age）と標準偏差（stddev_age）を求めてください。年齢は2019年12月31日時点での満年齢とし、データはサンプルではなく全数として扱います。結果テーブルは［pref_category］［ave_age］［stddev_age］の3カラムとします。

SECTION

P-3 第3群

問題 124 HINT ▶ 4-1, 5-1

［sales］と［customers］テーブルから、顧客別に販売した金額（revenue）の合計を求めてください。結果テーブルは、顧客の氏名（name）と販売金額の合計（sum_rev）の2カラムとします。[sum_rev］の大きい順に3名の顧客に絞り込んでください。

問題 125 HINT ▶ 4-1, 5-2

［web_log］テーブルに記録されたセッションはいくつあるか求めてください。[cid］と［session_count］のユニークな組み合わせがユニークなセッションです。結果テーブルのカラムは［session］としてください。

問題 126 HINT ▶ 4-2, 5-6

［sales］と［customers］テーブルから、女性客に販売した商品点数（quantity）の合計（sum_qty）を求めてください。結果テーブルのカラムは［sum_qty］としてください。

問題 127 HINT ▶ 4-2, 5-6

［sales］と［customers］テーブルから、都道府県別の販売個数（quantity）の合計（sum_qty）が小さい順に3つの都道府県を求めてください。結果テーブルは、都道府県（prefecture）と［sum_qty］の2カラムとします。

問題 128 HINT ▶ 4-2, 5-6

［sales］と［products］テーブルから、商品カテゴリ（product_category）別に販売個数（quantity）の合計（sum_qty）と販売金額（revenue）の合計（sum_rev）を取得してください。販売金額合計が大きい順に並べ替えて表示してください。

問題 129 HINT ➡ 5-2, 5-6

［sales］と［customers］テーブルから、販売金額（revenue）の合計（sum_rev）がもっとも多い顧客が居住する都道府県（prefecture）と販売金額合計（sum_rev）を求めてください。

問題 130 HINT ➡ 4-4, 5-1

［sales］テーブルから、顧客（user_id）別の販売金額の合計値について、最大額（max_sum_rev）、最小額（min_sum_rev）、平均額（avg_sum_rev）を求めてください。

問題 131 HINT ➡ 4-2, 5-2

［customers］テーブルから、2019年12月31日時点での年齢（age）ごとに顧客数（users）を求めてください。結果テーブルは男性に絞り込み、顧客が多い順にトップ3を表示してください。

問題 132 HINT ➡ 5-6, 7-4

［web_log］テーブルから［user_id］別のページビュー数を取得し、ページビュー数の多い順に3レコード取得してください。結果テーブルは［user_id］と［pageviews］の2カラムとします。なお、1人のユーザーは、会員登録前にはブラウザーを識別する［cid］しか持っておらず、会員登録後に［user_id］が発行されます。したがって、例えば［cid］として「XYZ」を持つが、［user_id］を持っていないレコードを考えると、その［cid］（XYZ）が後日［user_id］として「123」を持った場合、［cid］が「XYZ」の［pageviews］も［user_id］が「123」のユーザーの［pageviews］として合算します。WITH句を使って回答してください。

問題 133 HINT ➡ 5-2, 5-6

［customers］テーブルから、東京都に住む顧客に対して、販売金額（revenue）を基準として、もっとも販売した商品のトップ3を求めてください。結果テーブルは商品名（product_name）と販売金額の合計（sum_rev）の2カラムとし、［sum_rev］の降順で表示してください。

問題 134 HINT ➡ 5-6, 7-4

［sales］と［customers］テーブルから、もっとも販売金額（revenue）の合計（sum_rev）が多い男性顧客の年齢を求めてください。結果テーブルは、年齢（age）と購入額（sum_rev）の2カラムとしてください。年齢は2019年12月31日時点とします。

問題 135 HINT ➡ 4-2, 5-6

［web_log］テーブルから「カート完遂率」を求めてください。カート完遂率は、カートに到達したセッションのうち、コンバージョンまで至ったセッションの割合で表します。カートに到達したことは、セッション中に「/cart/」ページを閲覧したことで分かります。コンバージョンまで至ったことは、セッション中に「/thank_you/」ページを閲覧したことで分かります。結果テーブルは1行1列の1つの値となります。WITH句を使って回答してください。

問題 136 HINT ➡ 4-4, 5-2

［sales］テーブルから、ユニークな注文数（count_orders）、販売金額の合計（sum_rev）、注文別の販売金額の平均（average_order_value）を取得してください。

問題 137 HINT ➡ 5-2, 5-6

［sales］テーブルから、2019年に販売実績のない顧客を取得し、［user_id］の小さい順に10人の顧客をリストとして表示してください。

問題 138 HINT ➡ 5-6, 7-4

［web_log］テーブルから、曜日ごとのページビュー数を求めてください。結果テーブルは、曜日（day_of_week）とページビュー数（sum_pv）の2カラムとします。並べ替えは、日曜日（Sunday）から土曜日（Saturday）の順にして表示してください。

問題 139 HINT ➡ 5-6, 7-4

［sales］と［customers］テーブルから、都道府県（prefecture）別の平均単位（avg_unit_price）を求めてください。平均単価は、販売金額合計を販売数量合計で割って求めます。結果テーブルは［prefecture］と［avg_unit_price］の2カラムとし、金額の高い順に3つの［prefecture］を表示してください。

問題 140 HINT ➡ 4-4, 5-2

［sales］と［customers］テーブルから、東京、埼玉、千葉、神奈川を「東京圏」、大阪、京都、奈良、兵庫を「大阪圏」として、それぞれの圏（area）ごとに販売金額（revenue）の合計（sum_rev）を求めてください。結果テーブルは、圏（area）と販売金額合計（sum_rev）の2カラムとしてください。

問題 141 HINT ➡ 5-2, 7-4

［sales］と［customers］テーブルから、年齢（age）別の販売金額（revenue）の合計（sum_rev）を求め、［sum_rev］の大きい順に3レコードを取得してください。結果テーブルは年齢（age）と販売金額合計（sum_rev）の2カラムとしてください。年齢は2019年12月31日時点とします。

問題 142 HINT ➡ 5-6, 4-2

［sales］と［customers］テーブルから、東京に居住する顧客を「性別」と「プレミアム顧客か否か」の2軸で4つに分類したときの、それぞれの販売金額（revenue）の合計（sum_rev）を取得してください。結果テーブルは［gender］［is_premium］［sum_rev］の3カラムとします。順序は［gender］を優先して昇順、次に［is_premium］を降順（true、falseの順）に表示します。

問題 143 HINT ➡ 4-4, 5-2

［sales］と［customers］テーブルから、プレミアム顧客（premium_users）と、それ以外の顧客（standard_users）ごとに、顧客別の販売金額（revenue）合計の最大値（max_revenue_by_customer）を求めてください。結果テーブルは、顧客区分を表す［user_category］と［max_revenue_by_customer］の2カラムとします。

問題 144 HINT ➡ 4-2, 5-6

［sales］と［customers］テーブルから、50歳以上の顧客に対して販売した商品（product_id）別の販売金額（revenue）合計（sum_rev）トップ3を求めてください。結果テーブルは［product_id］と［sum_rev］の2カラムとしてください。年齢は2019年12月31日時点での満年齢とします。

問題 145 HINT → 5-6, 7-1

［sales］と［customers］テーブルから、男女別の注文回数の合計を求めてください。注文回数はユニークな［order_id］の個数とします。ただし、男性（gender = 1）と女性（gender = 2）は数値ではなく「男性」「女性」と表示してください。結果テーブルは性別（gender_cat）、注文回数（sum_orders）の2カラムとし、［sum_orders］の降順で表示してください。

問題 146 HINT → 5-1, 5-6

［sales］と［customers］テーブルで、会員登録日（register_date）を四半期ごとに分け、販売金額（revenue）の合計（sum_rev）を取得し、金額の大きい順にトップ3を表示してください。ただし、会員登録日が「null」の場合は「1916-4」と分類してください。

問題 147 HINT → 5-2, 5-6

［sales］と［customers］テーブルから、男女顧客別に販売金額（revenue）の平均を求めてください。平均販売金額（avg_order_value）は「販売金額の合計」を「注文回数（ユニークな［order_id］の個数）」で割ったものとします。結果テーブルは、性別（gender）、avg_order_valueの2カラムとします。ただし、男性（gender = 1）と女性（gender = 2）は数値ではなく「男性」「女性」と表示してください。

問題 148 HINT → 5-5, 5-6

［sales］と［customers］テーブルから、男性顧客と女性顧客で、顧客あたりの平均注文件数を取得してください。ただし、男性（gender = 1）と女性（gender = 2）は数値ではなく「男性」「女性」と表示してください。

問題 149 HINT → 3-2, 5-6

［sales］テーブルで、商品ID（product_id）が「1」の商品の平均単価（avg_unit_price_1）、テーブル全体の平均単価（avg_unit_price_all）、［avg_unit_price_1］と［avg_unit_price_all］の差（diff_percent）を取得してください。結果テーブルは［avg_unit_price_1］［avg_unit_price_all］［diff_percent］の3カラムとします。

問題 150 HINT ➡ 5-5, 5-6

［customers］テーブルから、関東地方を「東京」と「南関東」(千葉、埼玉、神奈川)、「北関東」(栃木、群馬、茨城）の3つに分けたとき、それぞれの地域（region）における2019年12月31日時点での平均年齢を求めてください。平均年齢（avg_age）は小数点以下第1位まで求め、平均年齢の若い順に表示します。

問題 151 HINT ➡ 4-2, 5-6

［sales］テーブルと［web_log］テーブルから、［product_id］別に販売個数（quantity）の合計（sum_qty)、販売金額（revenue）の合計（sum_rev)、ページビュー（pageviews）を取得して［pageviews］の多い順に3レコードに絞り込んで表示してください。［web_log］テーブルの［page］カラムにある「/prod/prod_id_［数字］」は、［product_id］が［数字］の商品について閲覧されたと考えてください。

問題 152 HINT ➡ 4-2, 5-6

［sales］テーブルから、2019年の各四半期の販売金額（revenue）の合計について、前年の四半期からの成長率を取得してください。第1四半期は1月から始まります。結果テーブルは、四半期を表す数字を格納した［quarter］、販売金額成長率を示す［growth_rate］の2カラムとし、［quarter］の昇順に表示してください。

問題 153 HINT ➡ 5-6, 7-4

［sales］テーブルから、2017年、2018年、2019年の3年すべてに注文履歴がある顧客の名前（name）を取得してください。結果テーブルには年齢の若い順に3人の顧客を表示してください。

問題 154 HINT ➡ 4-2, 5-6

［sales］テーブルから「販売金額（revenue）の合計が、2017年には10万円以上、2018年には20万円以上であったが、2019年には20万円以下」に該当する商品ID（product_id）を求めてください。

問題 155 HINT 5-2, 5-6

［sales］テーブルと［customers］テーブルから、注文別の販売金額の平均が、2017年は2,000円以下だったのに、2018年、2019年には6,000円以上となっている都道府県（prefecture）をすべて表示してください。

問題 156 HINT 4-4, 5-6

［sales］テーブルから、顧客別の年別の販売金額（revenue）の合計（sum_rev）を求め、2017年、2018年、2019年と販売金額が伸びている顧客について、販売総額（ttl_sum_rev）の大きい順に3人を絞り込んで表示してください。結果テーブルは、［user_id］、2017年の販売金額（sum_rev_17）、2018年の販売金額（sum_rev_18）、2019年の販売金額（sum_rev_19）、販売総額（ttl_sum_rev）の5カラムとしてください。

SECTION

P-4 第4群

問題 157 HINT ➡ 5-6, 6-1

［sales］テーブルから、商品ID（product_id）が「7」の商品を販売した顧客が何人いるか求めてください。カラム名は［users］とします。

問題 158 HINT ➡ 4-2, 6-1

［sales］テーブルに記録されている1つの注文番号（order_id）の中に、値引品と定価品が混在している注文はいくつあるか求めてください。結果テーブルのカラム名は［orders］とします。

問題 159 HINT ➡ 4-2, 6-1

［customers］テーブルから、2019年12月31日時点の顧客の平均年齢を求めてください。結果は小数点以下第1位まで表示してください。カラム名は［avg_age］とします。サブクエリを使って記述してください。

問題 160 HINT ➡ 5-6, 6-1

［customers］テーブルから、誕生日（birthday）が同じ顧客を取得してください。結果テーブルは、顧客名（name）と誕生日（birthday）の2カラムとして［birthday］の昇順に表示します。

問題 161 HINT ➡ 3-2, 6-1

［sales］テーブルで、各レコードに記録されている販売金額（revenue）が全体の販売金額に占める割合（revenue_percentage）を求めてください。結果テーブルは［order_id］［product_id］［revenue］［revenue_percentage］（販売金額の割合）の3カラムとし、［revenue］の多い順にトップ3を表示してください。

問題 162 HINT 5-6, 6-1

［sales］テーブルにて、2017年の販売金額（revenue）の散らばり度合いと、テーブル全体の販売金額の散らばり度合いを比較します。2017年の［revenue］の標準偏差（stddev_2017）と、テーブル全体の［revenue］の標準偏差（revenue_stddev）を求めてください。ただし、データはサンプルではなく全数（母集団）と考えます。

問題 163 HINT 4-2, 6-1

［sales］テーブルで、定価で販売されたレコードの平均の販売金額（revenue）よりも高額な販売金額（revenue）が記録されている、割引なしの定価で販売されたレコードはいくつあるか求めてください。カラム名は［orders_over_avg_rev］とします。

問題 164 HINT 6-1, 7-4

［sales］テーブルを注文ID（order_id）単位で見たときに、値引品を含まない注文の販売金額（revenue）の合計はいくらか求めてください。カラム名は［sum_rev］とします。

問題 165 HINT 3-2, 6-1

［sales］テーブルで、テーブル全体の販売金額（revenue）の平均よりも高額な売上のあるレコードについて、数量（quantity）の合計を求めてください。ただし、割引なしの定価で販売されたレコードだけを対象とします。カラム名は［sum_qty_over_avg_rev］とします。

問題 166 HINT 4-2, 6-1

［sales］テーブルから、値引品しか販売していない顧客は何人いるか求めてください。カラム名は［users］とします。

問題 167 HINT 4-2, 6-1

［sales］テーブルで、複数種類の商品を販売した顧客が何人いるかを求めてください。複数種類の商品の販売は、1回の注文でも、複数回の注文でも構いません。結果テーブルのカラム名は［users］とします。

問題 168 HINT 4-2, 6-1

［sales］テーブルから、1回の注文（order_id）で、複数の商品（product_id）を購入した履歴のある顧客が何人いるか調べてください。結果テーブルのカラム名は［users］とします。

問題 169 HINT 5-6, 6-1

［customers］テーブルから、都道府県別に顧客の平均年齢（2019年12月31日時点）を求めてください。結果テーブルは、都道府県（prefecture）と平均年齢（avg_age）の2カラムとしてください。なお、平均年齢は四捨五入して小数点第1位までとし、高い順にトップ3まで表示します。

問題 170 HINT 4-3, 6-1

［web_log］テーブルから、コンバージョン数を求めてください。コンバージョン数は、セッション中に「/thank_you/」ページの閲覧があったセッションの数です。カラム名は［cv］としてください。

問題 171 HINT 4-2, 6-1

［customers］テーブルで、テーブル全体の平均年齢よりも年齢が高い顧客の平均年齢を求めてください。年齢は2019年12月31日時点とします。

問題 172 HINT 5-6, 6-1

［customers］テーブルから、3人以上に使われているポピュラーな女性顧客の下の名前を取得してください。結果テーブルは、下の名前（first_name）と、その名前の顧客の数（users）の2カラムとしてください。

問題 173 HINT 4-3, 6-1

［sales］テーブルと［products］テーブルから、都道府県別に販売金額（revenue）の合計額の、販売金額全体に対する割合を求めてください。結果テーブルは、都道府県（prefecture）、販売金額合計（sum_rev）、販売金額の割合（percent_sum_rev）の3カラムとします。［sum_rev］の多い順に並べ替えて、トップ3の都道府県に絞り込んでください。

問題 174 HINT 6-1, 7-4

［sales］テーブルから、商品ID（product_id）が「1」と「2」の両方の商品を販売した顧客は何人いるか求めてください。異なった注文ID（order_id）での販売であっても構いません。結果テーブルのカラムは［users］としてください。

問題 175 HINT 4-2, 6-1

［sales］テーブルにて、注文ID（order_id）別の販売金額（revenue）の合計が、注文ID別の販売金額の合計の最大値に対する割合（percentage_against_max_revenue）を求めてください。

結果テーブルは、[order_id]、注文ID別の販売金額合計（sum_rev）、[percentage_against_max_revenue］の3カラムとします。[sum_rev］の大きい順にトップ5に絞り込んで表示してください。1レコード目の[percentage_against_max_revenue]は「1.0」となるはずです。

問題 176 HINT 4-2, 6-1

［sales］テーブルから、1回の注文で商品ID（product_id）が「1」と「2」の商品を同時に販売した顧客は何人いるかを求めてください。結果テーブルのカラムは［users］としてください。

問題 177 HINT 4-2, 6-1

［web_log］テーブルから、2回以上コンバージョンした顧客の顧客ID（user_id）とコンバージョン数（cv_by_user）を求めてください。

コンバージョンは、セッション中に「/thank_you/」ページを閲覧したか表示したかどうかで判断します。結果テーブルは［user_id］と［cv_by_user］の2カラムとし、[user_id］の昇順で表示してください。

問題 178 HINT 6-1, 7-4

［sales］テーブルと［customers］テーブルから、1つの注文（order_id）で、1つの商品（product）の販売が2レコード記録されている顧客を求め、その顧客のID（user_id）と、氏名（name）を表示してください。2人以上の顧客がいるときは、顧客ID（user_id）の小さい順に表示します。

問題 179 HINT 5-6, 6-1

［web_log］テーブルからサイト全体のコンバージョン率を求めてください。コンバージョン数は、セッション中に「/thank_you/」ページの閲覧があったセッションの数です。コンバージョン率は、コンバージョン数をセッション数で割って求めます。

結果テーブルはコンバージョン数（cv）、セッション（session）、コンバージョン率（cvr）の3カラムとします。コンバージョン率は小数点以下第4位まで求めてください。

問題 180 HINT 4-4, 6-1

［customers］テーブルから、顧客が会員登録したときの年齢（register_age）を取得し、その年齢で会員登録した顧客数（users）を求めてください。結果テーブルは［register_age］と顧客数（users）の2カラムとし、［users］の降順でトップ3を表示してください。誕生日（birthday）、もしくは会員登録日（register_date）が「null」の顧客は無視します。年齢は2019年12月31日時点での満年齢とします。

問題 181 HINT 6-1, 7-4

［web_log］テーブルから、サイト全体の直帰率（bounce_rate）を求めてください。直帰率は、1ページビューしか閲覧されなかったセッション数を、全体のセッションの数で割って求めます。結果テーブルは、小数点以下第3位を四捨五入して、小数点以下第2位まで求めてください。

問題 182 HINT 5-6, 6-1

［customers］テーブルから、都道府県（prefecture）別のプレミアム顧客（is_premium = true）の比率を求めてください。結果テーブルは、都道府県名（prefecture）、プレミアム顧客の数（premium_users）、顧客総数（users）、プレミアム顧客の比率（premium_rate）の4カラムとします。［premium_users］の多い順に3つの都道府県に絞り込んで表示してください。

問題 183 HINT 4-2, 6-1

［web_log］テーブルから、ページビュー／セッション（pv_per_session）を取得してください。［pv_per_session］は、サイト全体のページビュー数をサイト全体のセッションで割って求めます。結果テーブルは「pv_per_session」を小数点以下第2位で四捨五入し、小数点以下第1位まで表示してください。

問題 184 HINT 4-4, 6-1

［customers］テーブルで、テーブル全体の平均年齢よりも年齢が高い顧客を会員登録日（register_date）の古い順に3人取得してください。年齢は2019年12月31日時点とします。結果テーブルは、会員ID（user_id）と年齢（age）の2カラムとしてください。

問題 185 HINT 6-1, 7-1

［web_log］から、デバイス（device）別のコンバージョン率を求めてください。コンバージョン率は、セッション中に「/thank_you/」ページの閲覧があったセッション数を全体のセッション数で割って求めます。結果テーブルは、デバイス（device）、コンバージョン数（cv）、セッション（session）、コンバージョン率（cvr）の4カラムで取得してください。［cvr］は小数点以下第四位まで求めてください。

問題 186 HINT 6-1, 7-4

［sales］テーブルと［products］テーブルから、利益（profit）の合計額と利益率（profit_rate）を求めてください。利益は販売金額（revenue）の合計から仕入額（cost×quantity）の合計を差し引いて計算します。結果テーブルは利益の合計（sum_profit）と利益率（profit_rate）の2カラムとしてください。

問題 187 HINT 5-6, 6-1

［sales］テーブルから、商品ID（product_id）が「1」と「2」の商品は販売したが、「15」の商品は販売していない顧客は何人いるか、求めてください。結果テーブルは［users］としてください。

問題 188 HINT 5-6, 6-1

［web_log］テーブルを対象に、コンバージョン数別のユーザー数をまとめてください。コンバージョン数は、セッション中に「/thank_you/」ページの閲覧があったセッションの数です。

結果テーブルは、コンバージョン回数（cv_count）、ユーザー数（users）の2カラムとしてください。ユーザーはユニークな［cid］でカウントします。並べ替えはコンバージョン回数の多い順とします。

問題 189 HINT ▶ 5-6, 6-1

［sales］テーブルと［products］テーブルから、顧客別の販売金額（revenue）の合計を求め、性別（gender）ごとにそれらの散らばり度合いを標準偏差（std_dev）で計算してください。性別の名称（gender_name）は「男性」「女性」の2つに分類します。標準偏差は、四捨五入して整数で取り出します。結果テーブルは、性別の名称（gender_name）、標準偏差（std_dev）の2カラムとします。

問題 190 HINT ▶ 4-2, 6-1

［sales］テーブルと［products］テーブルから、もっとも利益が出た注文ID（order_id）と、その利益額（profit_by_order）を調べてください。結果テーブルは、注文ID（order_id）、利益額（profit_by_order）の2カラムとします。

問題 191 HINT ▶ 4-4, 6-1

［sales］テーブルと［customers］テーブルから、顧客ごとの販売金額合計を求め、性別（gender_name）ごとの最大値を計算してください。[gender_name] は「男性」「女性」の2つに分類します。結果テーブルは、性別（gender_name）と顧客別販売金額の合計の最大額（max_sum_rev）とします。

問題 192 HINT ▶ 5-6, 6-1

［web_log］テーブルから、Webサイト全体の平均セッション時間（avg_session_duration）を求めてください。平均セッション時間は、セッション時間（セッションの最初と最後のヒットスタンプの差）をセッションの数で割って求めます。

例えば、30秒、10秒、20秒のセッションがあれば、合計60秒÷3セッション＝20秒／セッションとなります。[avg_session_duration] は秒単位で小数点以下第1位まで表示してください。

問題 193 HINT ▶ 5-6, 6-1

［sales］テーブルと［web_log］テーブルから、[sales] テーブルに販売記録のある顧客（user_id）のうち、何人がwebサイトを利用していたかの割合を［web_usage_rate］として取得してください。

問題 194 HINT 4-4, 6-1

［sales］テーブルと［products］テーブルから、商品カテゴリ（product_category）ごとに利益（profit）と利益率（profit_rate）を取得してください。結果テーブルは利益率の高い順に並べ替えて表示してください。利益は販売金額（revenue）の合計から仕入額（cost×quantity）の合計を差し引いて計算します。

問題 195 HINT 4-2, 6-1

［sales］テーブルと［products］テーブルから、もっとも利益の少ない顧客を求めてください。結果テーブルは、顧客ID（user_id）、利益（profit）、利益率（profit_rate）の3カラムとしてください。利益は販売金額（revenue）の合計から仕入額（cost×quantity）の合計を差し引いて計算します。

問題 196 HINT 5-6, 6-1

［sales］テーブルから、商品ID（product_id）が「2」の商品を販売した後に、同じ年のうちに商品IDが「15」の商品を販売した顧客の顧客ID（user_id）を求めてください。結果テーブルは［user_id］の昇順に表示してください。

問題 197 HINT 6-1, 7-4

［web_log］テーブルで「/special/diet/」ページを閲覧したセッションと、閲覧していないセッションでそれぞれのコンバージョン率を取得してください。コンバージョン率は、全セッションのうち、コンバージョンしたセッションの比率です。コンバージョンしたセッションは、セッション中に「/thank_you/」ページを閲覧したセッションとします。結果テーブルは、「/special/diet/」ページを閲覧したセッションかどうかのブール型のフラグ（view_diet_page_session）とコンバージョン率（cvr）の2カラムとしてください。

問題 198 HINT 5-6, 6-1

［sales］テーブル、［products］テーブルと［customers］テーブルから、女性顧客に対して販売した商品別の販売金額（revenue）の合計のトップ3を、商品名（product_name）で取得してください。また、それら商品別の販売金額の合計が性別を問わない販売金額全体の何%にあたるかも表示してください。

結果テーブルは、商品名（product_name）、女性顧客に対する販売金額合計（sum_rev_female）、販売金額比率（percent_rev_female）の3カラムとしてください。［percent_rev_female］の値は小数のままで構いません。

問題 199 HINT 4-2, 5-6

［sales］テーブルから、値引き額の総額を求めてください。結果テーブルのカラム名は［sum_discount］としてください。

問題 200 HINT 4-4, 6-1

［sales］テーブルと［products］テーブルから、商品別の利益率を求めてください。結果テーブルは、商品名（product_name）、平均販売単価（avg_rev）、仕入れ単価（unit_cost）、販売個数の合計（sum_qty）、販売金額の合計（sum_rev）、仕入額の合計（sum_cost）、利益の合計（sum_profit）、平均利益率（avg_profit_rate）の8カラムとし、平均利益率の高い順にトップ3を表示してください。

問題 201 HINT 4-2, 7-2

［sales］テーブルから顧客ごとのバスケット分析をします。ある顧客に商品ID（product_id）「1」「2」「3」を販売した場合、「1-2」「1-3」「2-3」の組み合わせで購入した顧客がそれぞれ1名いることとします。その際、同一の注文であったか、別の注文であったかは問いません。

商品の組み合わせ（prod_combi）は、［product_id］を「-」（ハイフン）でつないで表します。結果テーブルは、商品の組み合わせ（prod_combi）と、その組み合わせで販売した顧客の数（users）の2カラムとし、［users］の多い順にトップ3だけ表示してください。［prod_combi］が「1-2」と「2-1」は同じ組み合わせなため、前者（小さい数字-大きい数字の組み合わせ）だけを表示するものとします。

問題 202 HINT 4-2, 5-6

目標の年間販売金額の合計が2017年80万円、2018年100万円、2019年120万円として、[sales] テーブルから年別の販売金額の達成率（achivement_rate）を取得してください。目標販売金額はテーブルを作るのではなく、WITH句で作成してください。

結果テーブルは、年（year）、目標販売金額（target）、販売実績（result_rev）、達成率（achievement_rate）とし、[year] の昇順に並べ替えて表示してください。

問題 203 HINT 5-2, 5-6

[sales] テーブルから、2017年の四半期ごとの販売金額（revenue）の合計（sum_rev）と累計（accum_sum_rev）を求めてください。結果テーブルは、四半期（quarter）を「2017-1」、「2017-2」と表現してください。結果テーブルは [quarter] [sum_rev] [accum_sum_rev] の3カラムとしてください。ただし、ウィンドウ関数は使ってはいけません。

問題 204 HINT 5-2, 5-6

[web_log] テーブルから、ランディングページ別のセッション数を求めてください。ランディングページとは、セッションが開始したページのことです。結果テーブルは、ランディングページ（landing_page）、セッション（sessions）の2カラムとし、[sessions] の降順にトップ3に絞り込んで表示してください。

問題 205 HINT 5-2, 5-6

[sales] テーブルから、顧客ID（user_id）別に注文回数（total_orders）、販売金額（revenue）の合計（total_revenue）、1回の注文での最大額（biggest_order）、販売したことのある商品の種類数（items）を取得してください。

結果テーブルは注文回数が7回以上の顧客に絞り込んで [user_id] の昇順に並べ替えて表示してください。

問題 206 HINT 5-2, 5-6

[sales] テーブルと [web_log] テーブルから、Webサイトを利用したユーザー（[web_log] テーブルに [user_id] が記録されているユーザー）と、利用していないユーザーを「Web利用あり顧客」と「Web利用なし顧客」のグループに分けて、それぞれ販売金

額（revenue）の合計と顧客別平均販売金額（avg_rev_by_user）を求めてください。結果テーブルは、Web利用有無をグループ化した［web_usage］、販売金額合計（sum_rev）、［avg_rev_by_user］の3カラムとし、［sum_rev］の降順に表示してください。

問題 207 HINT ➡ 5-6, 7-4

［web_log］テーブルから、初回の訪問日から、初回コンバージョン日までの平均日数を求めてください。ただし、ユーザーは［cid］で識別します。初回の訪問が記録されていないユーザー、およびコンバージョンしていないユーザーは無視してください。結果テーブルのカラム名は［avg_cv_leadtime_day］とし、日数は一の位で四捨五入して表示してください。

問題 208 HINT ➡ 4-2, 5-6

［web_log］テーブルから、月別のセッションとコンバージョン数の相関係数を求めてください。結果テーブルは［correlation］とします。

問題 209 HINT ➡ 5-6, 7-4

［sales］テーブルから、2018年の月ごとの販売金額（revenue）の合計を前年同月と比較した場合の増加率を取得してください。結果テーブルは、月（month）を「January、February」といった英語表記とし、2017年の販売金額（sum_rev_2017）、2018年の販売金額（sum_rev_2018）、前年同月比（change）の4カラムとしてください。

なお、増加率トップ3に絞り込み、［change］は小数点以下第2位まで表示してください。

問題 210 HINT ➡ 5-2, 5-6

［web_log］テーブルから、直近でコンバージョンしたユーザーの初回の訪問日から初回のコンバージョン日までの経過日数を求めてください。

結果テーブルはユーザーを識別する［cid］、初回訪問日（first_visit_day）、初めてコンバージョンした日（first_cv_day）、初回訪問日から初回コンバージョン日までの経過日数（cv_leadtime_days）の4カラムとしてください。ただし、初回の訪問が記録されているユーザーのみを対象とします。

SECTION

P-5 第5群

問題 211 HINT 3-3, 8-3

［customers］テーブルから、「東京」「大阪」「京都」のそれぞれの都府における、もっとも高齢の顧客名（name）を取得してください。結果テーブルは都道府県（prefecture）と顧客名（most_elder_custoemr）の2カラムとします。

問題 212 HINT 4-4, 8-3

［sales］テーブルから、年ごとにもっとも販売金額（revenue）の合計が大きい時間を求めてください。結果テーブルは、年（year）、時間（hour）、販売金額合計（sum_rev）の3カラムとし、年の昇順で並べ替えて表示してください。時間は「00」から「23」のいずれかの値となります。

問題 213 HINT 6-1, 8-3

［sales］テーブルから、2018年と2019年において、それぞれの年で販売金額（revenue）の合計が大きかった商品（product_id）のトップ3を取得してください。結果テーブルは、年（year）、商品ID（product_id）、ランク（rev_rank）の3カラムとします。

問題 214 HINT 6-1, 8-2

［customers］テーブルから、男女別に平均年齢（avg_age）と中央値（median_age）を求めてください。結果テーブルは［gender］［avg_age］［median_age］の3カラムとしてください。［gender］の値は数値ではなく「男性」「女性」と表示します。

問題 215 HINT 5-6, 8-2

［web_log］テーブルから、2018年第二四半期（2018年4月〜6月）について、ユーザー別のページビュー数を見ると「外れ値」が1つだけあります。そのページビュー数をWebサイトにもたらしたユーザーの［cid］を求めてください。結果テーブルは、1行1列の1つの値（cid）です。

問題 216 HINT 5-6, 8-3

［customers］テーブルから、男女別に、それぞれ2番目に若い顧客の名前（name）、誕生日（birthday）を取得してください。結果テーブルは［gender］［name］［birthday］の3カラムとします。［gender］の値は数値ではなく「男性」「女性」と表示してください。

問題 217 HINT 7-4, 8-3

［sales］テーブルから、2017年の第1四半期（1月〜3月）に初めて販売した顧客のグループ（コホート）の、2017年、2018年、2019年のそれぞれの第1四半期の販売金額（revenue）合計を求めてください。

結果テーブルは、販売が発生した四半期（year_quarter）と販売金額合計（sum_rev）の2カラムとし、［year_quarter］の昇順で表示してください。

問題 218 HINT 5-6, 8-3

［sales］テーブルから、2017年の月別販売について、1月を基準としたときの変化の割合がプラス方向に大きい月のトップ3を取得してください。

結果テーブルは、yyyy-MM年月（year_month）、1月からの変化率（change_from_jan）の2カラムとします。［change_from_jan］は四捨五入で小数点以下第2位まで表示してください。

問題 219 HINT 5-6, 8-3

［sales］テーブルの顧客（user_id）を販売金額（revenue）の合計が高い順に並べたときに、それら100人の顧客が全顧客数の何%にあたるか、および、それら100人目までの顧客の販売金額の累計が、全体の販売金額の何%にあたるかを取得してください。

結果テーブルは、100人の顧客が全顧客数に占める割合（user_accum_percentage）、それら100人の顧客への販売金額が全体の販売金額に占める割合（rev_accum_percentage）の2カラムとし、ともに四捨五入で小数点以下第3位まで表示してください。

問題 220 HINT 5-6, 8-2

［sales］テーブルから、1人の顧客に対する2種類の商品（product_id）の販売金額（revenue）が等しくなった記録を求めてください。ただし、商品別の販売金額の合計は2,000円以上とします。具体的には、同一の顧客に［product_id］の「1」も「2」も3,000円販売した、といった場合です。同一の注文でなくても構いません。

結果テーブルは、顧客ID（user_id）、1つ目の商品ID（priduct_id）、1つ目の商品の販売金額（sum_rev）、2つ目の商品ID（product_id2）、2つ目の商品の販売金額（sum_rev2）の5カラムとし、[user_id] の昇順で表示してください。

問題 221　HINT ➡ 5-2, 8-2

[sales] テーブル、[customers] テーブルと [products] テーブルから、性別（gender）ごとに販売金額（revenue）の合計が高かった商品（product_name）トップ3を求め、それぞれの商品の販売金額合計（sum_rev）とともに求めてください。

結果テーブルは、性別（gender）、商品名（product_name）、販売金額合計（sum_rev）、性別における販売金額ランク（rank_in_gender）の4カラムとし、[gender] [sum_rev] の降順に並べ替えます。[gender] の値は数値ではなく「男性」「女性」と表示してください。

問題 222　HINT ➡ 5-6, 8-3

[sales] テーブルから、2017年の月ごとの販売金額（revenue）の合計について、前月からの増加率を「（当月の販売金額－前月の販売金額）÷前月の販売金額」として取得してください。ただし、1月は前月の販売金額がないため除きます。

結果テーブルは、年月（yyyy-MM）、前月の販売金額（previous_month_sum_rev）、当月の販売金額（sum_rev）、増加率（change_from_previous_month）の4カラムとし、結果テーブルは、2月、3月、4月の3か月分だけに絞り込んで表示してください。増加率は四捨五入で小数点以下第3位まで表示してください。

問題 223　HINT ➡ 5-6, 8-4

[sales] テーブルの顧客（user_id）を、販売金額（revenue）の高い順に並べて、トップ20%、次の20%、その次の20%というように、5つのグループに分割します。「1」～「5」の値を持つこのカラムを「rev_bucket」と名付けます。

[rev_bucket] ごとの販売金額（revenue）の合計を [sum_rev_by_bucket] カラムに格納します。[rev_bucket] ごとの [sum_rev_by_bucket] の累計額を [accum_sum_rev_by_bucket]、さらに [acum_sum_rev_by_bucket] が販売総額全体に占める割合を [accum_rev_percentage] として求めてください。

結果テーブルは [rev_bucket] [sum_rev_by_bucket] [accum_sum_rev_by_bucket] [accum_rev_percentage] の4カラム、5レコードとなります。[rev_bucket] の小さい順に並べ替えて表示してください。

問題 224 HINT 7-3, 8-3

［sales］テーブルでRFM分析を行います。顧客別の、もっとも新しい注文日（date_time）を「Recency」、注文（order_id）回数の合計を「Frequency」、販売金額（revenue）の合計を「Monetary」とします。また、RFMの3つの指標それぞれで顧客を5つのグループに分けて「1」～「5」の値を付与します（Recencyが新しいほうが「1」、古いほうが「5」、Frequencyが高いほうが「1」、低いほうが「5」、Monetaryが大きいほうが「1」、小さいほうが「5」）。3つのスコアを合計したものを［total_score］とします。

［total_score］ごとに顧客数をカウントし、RFMのすべてのスコアが最上位である［total_score］が「3」の顧客は何人いるか求めてください。結果テーブルのカラム名は［users］とします。

問題 225 HINT 5-6, 8-3

［sales］テーブルから、３回以上注文した顧客のうち、2回目の注文と3回目の注文の間隔を取得してください。結果テーブルは、顧客ID（user_id）、2回目と3日目の注文の間隔を「日」単位で表した整数（interval_between_2_and_3）の2カラムとし、短い順にトップ3顧客に絞込んで表示してください。

問題 226 HINT 5-2, 8-2

［sales］テーブルから、2017年の販売金額デシルがバケット1（販売金額合計が上位10%の顧客）、2018年デシルでもバケット1に属する顧客が1人だけいます。その顧客の顧客ID［user_id］を取得してください。

問題 227 HINT 5-6, 8-2

［web_log］テーブルから、ランディングページごとの直帰率を求めてください。直帰とは、1セッションで1ページビューしか発生しなかったセッションのことです。直帰率はランディングページごとに「直帰数の合計÷セッションの合計」で求められます。

結果テーブルはランディングページ（landing_page）、セッション（session）、直帰数（bounce）、直帰率（bounce_rate）の4カラムとし、［bounce_rate］の高い順にトップ3に絞り込んで表示してください。

なお、pageには、末尾に「?sys=abc123」が存在する場合としない場合がありますが、両者は同じページですので、「?sys=abc123」が存在しないページにマージしてください。例えば「/prod/prod_id_1/?sys=abc123」は「/prod/prod_id_1/」にマージします。

問題 228 HINT ▶ 7-4, 8-3

［sales］テーブルから、複数の注文をした顧客について、注文別の最小額（min_rev）、最大額（max_rev）、金額の幅（rev_range）を求めてください。結果テーブルは［user_id］［min_rev］［max_rev］［rev_range］の4カラムとし、［rev_range］の大きい順にトップ3を表示してください。

問題 229 HINT ▶ 5-6, 8-4

［web_log］テーブルから、トップページにランディングしたセッションについて、セッション中の2ページ目、3ページ目として、どのページに、何セッションが遷移したかを取得してください。直帰した場合は、セッション中の2ページ目は存在しないため「null」となりますが、「null」も含めてください。

結果テーブルは、ランディングページ（landing_page）、セッション中の2ページ目（second_page）、セッション（session）の3カラムとし、［session］の多い順にトップ5を表示してください。

問題 230 HINT ▶ 7-4, 8-3

［sales］テーブル、［customers］テーブルと［products］テーブルから、2017年から2019年の各年において、もっとも販売金額（revenue）の合計が大きかった顧客と、その年にその顧客に対して販売した金額がもっとも大きかった商品を取得してください。

結果テーブルは、年（year）、顧客ID（user_id）、顧客名（name）、商品ID（best_product_by_user）、商品名（best_product_nameby_user）、商品別販売金額合計（sum_rev_by_user_product）の6カラムとし、［year］の昇順に並べ替えて表示してください。

おわりに

最後までお読みいただき、ありがとうございました。お疲れさまでした。

読了したみなさんであれば同意していただけると思いますが、本書はエンジニアではない、一般のビジネスパーソンの方々に、しっかりとしたSQLの知識・スキルを身につけていただけるように設計しました。

そのために、テーブルを更新する、テーブルにレコードを挿入・削除するといった、通常エンジニア側に求められるSQLのテクニックについては、大胆に、まったく触れていません。そのぶん、全編を通して「ビジネスデータの分析」をするためのテクニックに解説を集中しています。

その「選択と集中」により、著者としては以下の5点を実現することができたのではないかと考えています。

1. Google BigQueryという、多くのビジネスパーソンにとって、実はハードルの低いデータベースの利用開始方法をていねいに解説したこと。
2. まったく前提知識がなくSQLを学ぶ読者にも抵抗なく読み進めてもらえる、もっとも難易度の低いテクニックから解説したこと。
3. 実際の業務でも利用できるよう、難易度が高い一方で、分析の幅が大きく広がるウィンドウ関数や、UNNEST関数について触れられたこと。
4. 「SQLは手を動かさなければ身につかない」という信念のもと、類書と比較してももっとも多い230問のドリルを用意し、最大で1,000レコード以上ある4つのテーブルを対象に手を動かして学習できる環境を提供したこと。
5. 230問のドリルのうち、巻末の演習ドリル200問の解答は「できるネット」上にあり、書籍とWebのハイブリッド型で学習環境を提供できたこと。

上記の意図がどのように受け止められたか、読者のみなさんの評価を待ちたいと思います。私にとって、本書が4冊目の書籍となります。本書も含め、すべての書籍で編集者としてインプレスの小渕隆和さんに伴走していただきました。本当にありがとうございました。

2021年1月　木田和廣

INDEX

A

ABS関数 …… 310
AND …… 104
AS …… 078, 096
ASC …… 087
AVG関数 …… 142, 416

B

BETWEEN …… 111
BigQuery …… 034
　アカウント …… 040
　下部パネル …… 059
　クエリエディタ …… 059, 061
　コンソール …… 059
　サンドボックス …… 035
　データセット …… 039, 050
　テーブル …… 052, 069
　ナビゲーションパネル …… 059
　ビュー …… 067, 069
　プロジェクト …… 041, 060
　無料枠 …… 045
　料金 …… 044, 118
BIツール …… 018, 419
BOOL型 …… 030

C

CASE …… 155
CAST関数 …… 314
CEIL関数 …… 308
COALESCE関数 …… 207
CONCAT関数 …… 317
CORR関数 …… 353
COUNT関数 …… 138
CROSS JOIN …… 202
CSV …… 046, 058
CURRENT_DATETIME関数 …… 336
CURRENT_DATE関数 …… 336

D

DATE_ADD関数 …… 337
DATE_DIFF関数 …… 341
DATE_TRUNC関数 …… 342
DATETIME_ADD関数 …… 337
DATETIME_DIFF関数 …… 341
DATETIME_TRUNC関数 …… 342
DATETIME型 …… 031
DATETIME関数 …… 339
DATE型 …… 031
DATE関数 …… 339
DESC …… 087

E

ELSE …… 159
END …… 159
EXCEPT …… 090, 210
EXTRACT関数 …… 344

F

FIRST_VALUE関数 …… 392
FLOAT64型 …… 030
FLOOR関数 …… 308
FORMAT_DATETIME関数 …… 346
FORMAT_DATE関数 …… 346
FROM …… 075, 120
FULL OUTER JOIN …… 186

G

Google Cloud Platform …… 035
Google Cloud Storage …… 058
Googleアナリティクス …… 013
Googleアナリティクス4 …… 015, 288
　UNNEST関数 …… 292
　日別のテーブルをまとめて指定 …… 295
Googleデータポータル …… 071, 286
GROUP BY …… 132, 135

H

HAVING ····· 149, 151

I

IF ····· 155
IFNULL関数 ····· 205
IN ····· 107
INNER JOIN ····· 179
INSTR関数 ····· 321
INT64型 ····· 030
INTERSECT ····· 218
IS NOT NULL ····· 117
IS NOT TRUE ····· 115
IS NULL ····· 115
IS TRUE ····· 114

J

JOIN ····· 178

L

LAG関数 ····· 406
LAST_VALUE関数 ····· 399
LEAD関数 ····· 403
LEFT OUTER JOIN ····· 183
LEFT関数 ····· 320
LENGTH関数 ····· 325
LIKE ····· 109
LIMIT ····· 099

M

MAX関数 ····· 142, 367
MIN関数 ····· 142
MOD関数 ····· 312

N

NOT BETWEEN ····· 113
NOT IN ····· 109
NOT LIKE ····· 111
NTH_VALUE関数 ····· 395
NTILE関数 ····· 388
null ····· 029
  置換 ····· 205
  判定 ····· 115
NULLABLE ····· 052
NUMERIC型 ····· 030

O

OFFSET ····· 100
ON ····· 181
OR ····· 104
ORDER BY ····· 086, 371
OVER ····· 371

P

PARTITION BY ····· 371
PERCENTILE_CONT関数 ····· 408

R

RAND関数 ····· 313
RANGE BETWEEN ····· 378
RANK関数 ····· 380
RDB ····· 023
RECORD型 ····· 290
REGEXP_CONTAINS関数 ····· 328
REGEXP_EXTRACT関数 ····· 330
REGEXP_REPLACE関数 ····· 331
REPLACE関数 ····· 324
RIGHT OUTER JOIN ····· 186
RIGHT関数 ····· 320
ROUND関数 ····· 308
ROW_NUMBER関数 ····· 385
ROWS BETWEEN ····· 374

S

SELECT ····· 075, 085
SELF JOIN ····· 200
SQL ····· 023, 074
  FROMを省略 ····· 120
  内側から読み解く ····· 261

各句が実行される順序 …… 153
末尾の「;」 …… 082
学ぶ意義 …… 012
用語 …… 027
STDDEV_POP関数 …… 147, 352
STRING型 …… 030
STRPOS関数 …… 323
SUBSTR関数 …… 318
SUM関数 …… 142, 412

T

Tableau …… 017, 419
THEN …… 159
TIMESTAMP型 …… 031
TIME型 …… 031

U

UNION …… 178, 210
UNNEST関数 …… 288, 292
USING …… 182

V

VAR_POP関数 …… 351

W

WHEN …… 159
WHERE …… 101, 152
WINDOWフレーム …… 371, 373
WITH …… 227

ア

一時テーブル …… 064
移動平均 …… 416
インデント …… 261
ウィンドウ関数 …… 366
演習用ファイル …… 010, 046

カ

仮想テーブル …… 225
WITH句 …… 227
サブクエリ …… 246
ビュー …… 232
カラム …… 027
関数
移動平均を取得 …… 416
行番号を取得 …… 385
均等な数に分割 …… 388
現在の日付・時刻を取得 …… 336
五数要約の指標を取得 …… 408
最後の値を取得 …… 399
最初の値を取得 …… 391
数値を丸める …… 308
正規表現で合致する文字列を取得 …… 330
正規表現で合致する文字列を置換 …… 331
正規表現で文字列の有無を判定 …… 328
絶対値を取得 …… 310
相関係数を取得 …… 353
直後の値を取得 …… 403
直前の値を取得 …… 406
データ型を変換 …… 314
任意の順番の値を取得 …… 395
日付・時刻型オブジェクトを作成 …… 339
日付・時刻に一定値を加算 …… 337
日付・時刻の差分を取得 …… 341
日付・時刻の表示形式を変更 …… 346
日付・時刻を丸める …… 342
日付の属性を取得 …… 344
標準偏差を取得 …… 352
分散を取得 …… 351
文字列の一部を取得 …… 318
文字列の出現位置を取得 …… 321
文字列の長さを取得 …… 325
文字列や値を連結 …… 317
文字列を置換 …… 324
ランクを取得 …… 380
乱数を発生 …… 313
累計を取得 …… 412
割り算の余りを取得 …… 312

行 …… 028
絞り込み …… 098
クエリ …… 029
一部を実行 …… 063
データ量 …… 045, 119
保存 …… 066
履歴 …… 072
クエリエディタ …… 061
グループ化 …… 128, 154
結果テーブル …… 064
視覚化 …… 071, 286
スキーマの確認 …… 065
コホート分析 …… 391
コメント …… 080

サ

サブクエリ …… 246, 264
算術演算子 …… 094
集計関数 …… 136
集計分析関数 …… 370, 412
条件式 …… 103, 107
ジョブ …… 056, 072
スキーマ …… 028
正規表現 …… 327

タ

データ型 …… 028, 030
データセット …… 039, 050
テーブル …… 027, 174
作成 …… 052
縦持ち・横持ち変換 …… 266, 270
プレビュー …… 098
テーブルの結合 …… 179
3つ以上のテーブル …… 193, 199
同一のテーブル …… 200
非等値結合 …… 208
テーブルの集合演算 …… 210, 222
デシル分析 …… 388
統計集計関数 …… 147, 350

ナ・ハ

ナビゲーション関数 …… 370, 391
パーティション …… 366
バスケット分析 …… 208
番号付け関数 …… 369, 380
ビュー …… 029, 067
フィールド …… 027
プロジェクト …… 039
作成 …… 041
選択 …… 060
分析 …… 021
Webページの滞在時間 …… 402
ゴールデン導線分析 …… 385
コホート分析 …… 391
最終購入日からの経過日数 …… 399
作業記録とカレンダーの利用 …… 280
初回が最大額の注文 …… 391
初回と2回目の購入日の差 …… 395
前月との販売個数の差 …… 406
注文履歴からのユーザーの分類 …… 273
デシル分析 …… 388
特定商品の有無別で比較 …… 276
人気商品分析 …… 380
バスケット分析 …… 208
偏差値 …… 355

マ・ラ

メタ文字 …… 327
リレーショナルデータベース …… 023, 174
累計 …… 412
レコード …… 028, 128
列 …… 027
作成 …… 091
取得 …… 084
除外して取得 …… 090
すべて取得 …… 089
並べ替え …… 086
別名 …… 096

本書のご感想をぜひお寄せください

https://book.impress.co.jp/books/1119101189

読者登録サービス CLUB IMPRESS

アンケート回答者の中から、抽選で商品券（1万円分）や図書カード（1,000円分）などを毎月プレゼント。当選は賞品の発送をもって代えさせていただきます。

STAFF LIST

カバー・本文デザイン	松本 歩（細山田デザイン事務所）
帯・本文イラスト	山内麗資
編集・DTP	今井 孝
デザイン制作室	今津幸弘（imazu@impress.co.jp）
	鈴木 薫（suzu-kao@impress.co.jp）
編集協力	佐川莉央（sagawa-r@impress.co.jp）
編集長	小渕隆和（obuchi@impress.co.jp）

■商品に関する問い合わせ先

インプレスブックスのお問い合わせフォームより入力してください。
https://book.impress.co.jp/info/
上記フォームがご利用頂けない場合のメールでの問い合わせ先
info@impress.co.jp

●本書の内容に関するご質問は、お問い合わせフォーム、メールまたは封書にて書名・ISBN・お名前・電話番号と該当するページや具体的な質問内容、お使いの動作環境などを明記のうえ、お問い合わせください。
●電話やFAX等でのご質問には対応しておりません。なお、本書の範囲を超える質問に関しましてはお答えできませんのでご了承ください。
●インプレスブックス（https://book.impress.co.jp/）では、本書を含めインプレスの出版物に関するサポート情報などを提供しておりますのでそちらもご覧ください。
●該当書籍の奥付に記載されている初版発行日から3年が経過した場合、もしくは該当書籍で紹介している製品やサービスについて提供会社によるサポートが終了した場合は、ご質問にお答えしかねる場合があります。
●本書の利用によって生じる直接的あるいは間接的被害について、著者ならびに弊社では一切の責任を負いかねます。あらかじめご了承ください。

■落丁・乱丁本などのお問い合わせ先

TEL：03-6837-5016
FAX：03-6837-5023
service@impress.co.jp
（受付時間 10:00-12:00／13:00-17:30、土日・祝祭日を除く）
●古書店で購入されたものについてはお取り替えできません。

■書店／販売店の窓口

株式会社インプレス 受注センター
TEL：048-449-8040
FAX：048-449-8041

株式会社インプレス 出版営業部
TEL：03-6837-4635

集中演習（しゅうちゅうえんしゅう）　SQL入門（エスキューエルにゅうもん）
Google（グーグル） BigQuery（ビッグクエリ）ではじめるビジネスデータ分析（ぶんせき）
（できるDigital（デジタル） Camp（キャンプ））

2021年2月21日　初版発行

著　者　木田和廣（キダカズヒロ）
発行人　小川 亨
編集人　高橋隆志
発行所　株式会社インプレス
　　〒101-0051　東京都千代田区神田神保町一丁目105番地
　　ホームページ　https://book.impress.co.jp/
印刷所　音羽印刷株式会社

本書は著作権法上の保護を受けています。本書の一部あるいは全部について（ソフトウェア及びプログラムを含む）、株式会社インプレスから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

Copyright ©2021 Kazuhiro Kida. All rights reserved.
ISBN978-4-295-01074-6 C3055
Printed in Japan