データ加工から売上・ユーザー把握、
レポーティング等々の各種データ分析まで

ビッグデータ分析・活用のためのSQLレシピ

加嵜長門、田宮直人［著］

「ビッグデータ時代」のSQL活用術・レシピ集！

PostgreSQL、Apache Hive、Amazon Redshift、
Google BigQuery、SparkSQL対応。

マイナビ

ビッグデータ 分析・活用のための SQLレシピ

CONTENTS

Chapter 1
ビッグデータ時代に求められる分析力とは

Chapter 2
本書で扱うツールとデータ群

Chapter 3
データ加工のためのSQL

Chapter 4
売上を把握するためのデータ抽出

Chapter 5
ユーザーを把握するためのデータ抽出

Chapter 6
Webサイトでの行動を把握するための
データ抽出

Chapter 7
データ活用の精度を高めるための分析術

Chapter 8
データを武器にするための分析術

Chapter 9
知識に留めず行動を起こす

データ加工から売上・ユーザー把握、
レポーティング等々の各種データ分析まで

ビッグデータ分析・活用のためのSQLレシピ

加嵜長門、田宮直人［著］

マイナビ

本書のサポートサイト

本書のサンプルデータ、補足情報、訂正情報などを掲載します。適宜ご参照ください。
http://book.mynavi.jp/supportsite/detail/9784839961268.html

●本書は2017年2月段階での情報に基づいて執筆されています。
本書に登場する製品やソフトウェア、サービスのバージョン、画面、機能、URL、製品のスペックなどの情報は、すべてその原稿執筆時点でのものです。
執筆以降に変更されている可能性がありますので、ご了承ください。

●本書に記載された内容は、情報の提供のみを目的としております。
したがって、本書を用いての運用はすべてお客様自身の責任と判断において行ってください。

●本書の制作にあたっては正確な記述につとめましたが、
著者や出版社のいずれも、本書の内容に関してなんらかの保証をするものではなく、
内容に関するいかなる運用結果についてもいっさいの責任を負いません。あらかじめご了承ください。

●本書中の会社名や商品名は、該当する各社の商標または登録商標です。
本書中ではTMおよびRマークは省略させていただいております。

はじめに

本書は、執筆陣が普段の業務で実際に作成しているレポートやSQLのコードをより汎用化して、レシピ集としてまとめたものです。SQLは歴史の長い言語ですが、標準仕様にはウィンドウ関数やCTEをはじめ、新たな概念がいくつも取り込まれ、ハードウェアやミドルウェアの進化と共に活用の場をどんどん広げつつあります。しかし、残念ながら、新しいSQLを活用するノウハウは、十分に浸透していないように感じます。

昨今では、SQLの機能を解説する記事や書籍は数多く存在しますが、実際の業務で記述するクエリとは大きな隔たりがあり、最終的には自らコードを考えることがほとんどです。周辺の現場を見渡しても、SQLといえばWebサービスのバックエンドとして、RDBの問い合わせに利用するだけで、分析用途のSQLを効率的に記述するノウハウは皆無でした。
同様に、データ分析の手法も分析を通じてサービスを改善する事例は頻出するものの、肝心の手段に関しては触れられておらず、やりたいけれどもやり方が分からず途方に暮れる経験があります。

最新のSQLの記述から分析手法、そしてサービス改善手法を1冊にまとめることが、大きな価値になるのではないだろうかと考えたことが、本書執筆のきっかけです。日常的な業務をこなしながらの執筆だったため、本書の原稿を参照しながらSQLを作成したり、分析やサービスの改善方法を質問された際には、本書の原稿を前に説明するほど、実務に即したものにまとめあげられたと自負しています。

本書を通じて、より多くの方がビッグデータ活用のための分析力・SQL力を身に付ける手助けとなれば幸いです。

2017年 春
加嵜 長門、田宮 直人

本書の構成

本書は、4部で構成されています。
Chapter 1と2は、データ分析の世界に踏み込む前に読んでいただきたい、いわば導入部分です。本書が提供する価値を説明し、取り扱うデータやミドルウェアを記載しています。

Chapter 3で基礎的なSQLの記述やデータ加工の手法を解説し、続くChapter 4～6では、実践的なコード例と共に、さまざまな場面で活用できるデータ抽出を解説しています。

Chapter 7ではデータ加工やクレンジングなど、データ活用の精度を高めることに主眼を置き、Chapter 8ではデータを武器として戦う分析術を豊富なコード例と共に解説します。

最後のChapter 9では、本書を知識だけに留めず、データ活用の事例やヒントを紹介することで、実践へと踏み出す1歩を手助けします。

本書の読み方

どこから読み始めていただいても構いません。目次を参照して興味が湧いた章から読み進めてください。それが最善の選択です。

本書では、データの集計からレポートの作成、改善の提案、APIの作成など、さまざまなシーンを想定して、具体的かつ実践的に説明しています。また、執筆陣の豊富な経験に裏打ちされた、本書の解説と豊富なコード例が、さまざまな課題を解決する糸口になるはずです。
以下に、本書の読み進め方として3パターンを提案しますが、まずは興味を持った章から読み始めてください。

■ 冒頭から順番に読む

難易度が低く、容易に習得できる内容から、徐々に難しく高度な内容を取り扱えるように、順を追って解説を進めています。
特に知識を網羅的に吸収したいと考えている方におすすめします。

■ 気になるところから読む

分析したい対象、改善したい事象、作成したいレポートなど、今現在抱えている課題に関連する項目を、目次や索引から探し出して、本書を読み進めていただいても大丈夫です。

本書は、基本的どこから読み始めても理解できるように、各項は単独で完結する内容にまとめています。また、各項で遡って確認すべきことは、関連項目として別項のインデックスを用意しています。不明な点があれば、関連項目を辿ってみてください。

■ 最後のChapterから読む

最終章には、冒頭でビッグデータを有効に活用するまでの段階を整理し、データ活用に事例、ヒント、検討すべき内容をまとめています。
所属する組織のステータスに応じてやるべきことがまとめられているため、必要に応じて本書記載の項目を参照可能です。

SQLに関する注意事項

記載のSQLは、執筆時にサンプルデータを用いて実行を確認しています。
万が一、既にビッグデータ解析基盤を導入済みで、コード例を手元の環境で実行すると、保有全データに対する操作が行われるため、システムに負荷を掛けたり、ミドルウェアや環境次第では、高額な料金が発生する可能性もあります。実行するSQLは適切なパーティションに対して実行する、取得件数を制限するなどの配慮をお願いします。

本書で扱う各ミドルウェアとバージョン

■ PostgreSQL

本書執筆時でのPostgreSQLの最新版は、バージョン9.6.2です。本書では、バージョン9.6.0で動作を確認しています（2017年3月）。
バージョン9.6.xのリリースノートは、下記公式ドキュメントを参照してください。
https://www.postgresql.org/docs/9.6/static/release-9-6.html

■ Apache Hive

Apache Hiveの最新版はバージョン2.1.1ですが、ほとんどのHadoopディストリビューションは最新版のHiveをサポートされていません。代表的なHadoopディストリビューションでは、「CDH」（Cloudera Distribution Including Apache Hadoop）の最新版（5.10）がHive 1.1に対応、「HDP」（Hortonworks Data Platform）の最新版（2.5）はHive 1.2.1に対応しています。
本書では、CDH 5.7.4に対応するHive 1.1で動作を確認しています。

■ Amazon Redshift

Amazon Redshiftのバージョン（クラスターバージョン）は長らく1.0のままですが、細かな改定が随時リリースされています。
本書では、2017年2月16日アナウンスのバージョン1.0.1187で動作を確認しています。
最新リリース情報は、下記の公式フォーラムを確認してください。
https://forums.aws.amazon.com/forum.jspa?forumID=155&start=0

■ Google BigQuery

Google BigQueryもクラウドサービスであり、新機能やバグ修正が随時リリースされています。本書では、2017年2月1日リリースのバージョンで動作を確認しています。
最新リリース情報は、下記の公式リリースノートを参照してください。
https://cloud.google.com/bigquery/docs/release-notes

■ SparkSQL

本書執筆時のApache Sparkの最新版は、バージョン2.1.0です。本書では、バージョン2.0.0で動作を確認しています。
バージョン2.0.0のリリースノートは、下記の公式ドキュメントを参照してください。
http://spark.apache.org/releases/spark-release-2-0-0.html

CONTENTS

Chapter 1 ビッグデータ時代に求められる分析力とは 001

1 データを取り巻く環境の変化 002
2 さまざまな課題 004
1 分析担当者の課題 004
2 エンジニアの課題 005
3 職種を越える横断的な分析力を身に付ける 006

Chapter 2 本書で扱うツールとデータ群 009

1 システム 010
1 PostgreSQL 010
2 Apache Hive 010
3 Amazon Redshift 012
4 Google BigQuery 013
5 SparkSQL 013
2 データ 015
1 データの種類 015
2 業務データ 017
3 ログデータ 019
4 2つのデータを利用することで生まれる価値 021
コラム 「1%の当選確率なら100回やれば1回当たる？」 024

Chapter 3 データ加工のためのSQL 025

1 1つの値に対する操作 026
1 コード値をラベルに置き換える 027
2 URLから要素を取り出す 028
3 文字列を配列に分解する 030
4 日付やタイムスタンプを扱う 031
5 欠損値をデフォルト値に置き換える 035
2 複数の値に対する操作 036
1 文字列を連結する 037
2 複数の値を比較する 038
3 2つの値の比率を計算する 042

4 2つの値の距離を計算する 045
5 日付・時刻を計算する 047
6 IPアドレスを扱う 053
3 1つのテーブルに対する操作 057
1 グループの特徴を捉える 058
2 グループの中での順序を扱う 061
3 縦持ちのデータを横持ちに変換する 068
4 横持ちのデータを縦持ちに変換する 070
4 複数のテーブルに対する操作 078
1 複数のテーブルを縦に並べる 079
2 複数のテーブルを横に並べる 080
3 条件のフラグを0と1で表現する 083
4 計算したテーブルに名前を付けて再利用する 085
5 擬似的なテーブルを作成する 088

Chapter 4 売上を把握するためのデータ抽出 093

1 時系列に沿ってデータを集約する 094
1 日別の売上を集計する 095
2 移動平均を用いて日別の推移を見る 096
3 当月売上の累計を求める 098
4 月別売上の昨対比を求める 102
5 Zチャートで業績の推移を見る 104
6 売上を把握するための大事なポイント 109
2 多面的な軸を使ってデータを集約する 112
1 カテゴリ別の売上と小計を計算する 113
2 ABC分析で売れ筋を判別する 115
3 ファンチャートで商品の売れ行きの伸び率を見る 119
4 ヒストグラムで購入価格帯を集計する 122

Chapter 5 ユーザーを把握するためのデータ抽出 131

1 ユーザー全体の特徴・傾向を見つける 132
1 ユーザーのアクション数を集計する 133
2 年齢別区分を集計する 139

3 年齢別区分ごとの特徴を抽出する 143
4 ユーザーの訪問頻度を集計する 145
5 ベン図でユーザーのアクションを集計する 148
6 デシル分析でユーザーを10段階のグループに分ける 155
7 RFM分析でユーザーを3つの視点でグループに分ける 159
2 ユーザー全体の時系列による状態変化を見付ける 169
1 登録数の推移と傾向を見る 170
2 継続率と定着率を算出する 174
3 継続と定着に影響するアクションを集計する 189
4 アクション回数に応じた定着率を集計する 193
5 利用日数に応じた定着率を集計する 199
6 ユーザーの残存率を集計する 203
7 訪問頻度からユーザーの属性を定義し集計する 207
8 訪問種別を定義して成長指数を集計する 218
9 指標を改善する手順を身に付ける 225
3 時系列に沿ってユーザー個別のアクションを分析する 227
1 ユーザーのアクション間隔を集計する 228
2 カート追加後に購入されているかを把握する 233
3 登録からの売上を経過日数別に集計する 237

Chapter 6 Webサイトでの行動を把握するためのデータ抽出 243

1 サイト全体の特徴・傾向を見つける 244
1 日次の訪問者数・訪問回数・ページビューを集計する 245
2 ページ毎の訪問者数・訪問回数・ページビューを集計する 247
3 流入元別に訪問回数やCVRを集計する 251
4 アクセスされる曜日、時間帯を把握する 256
コラム 『デバイスによるユーザー行動の違いを意識しよう』 259
2 サイト内のユーザー行動を把握する 260
1 入口ページと出口ページを把握する 262
2 離脱率と直帰率を計算する 266
3 成果に結び付くページを把握する 269
4 ページの価値を算出する 273
5 検索条件毎のユーザー行動を可視化する 279
6 フォールアウトレポートを用いてユーザーの回遊を可視化する 285
7 サイト内のユーザーフローを把握する 291

8 ページの読了率を集計する 298
9 ユーザー行動の全体像を可視化する 301
3 エントリーフォームを最適化する 304
1 エラー率を集計する 305
2 入力〜確認〜完了までの遷移率を集計する 307
3 フォーム直帰率を集計する 310
4 エラーが発生している項目と内容を集計する 312
コラム 『印刷したレポートを折ってみる！？』 314

Chapter 7 データ活用の精度を高めるための分析術 315
1 データ加工による新たな切り口を作る 316
1 IPアドレスから国・地域を補完する 316
2 都道府県に隣接都道府県情報を付与する 321
3 土日・祝日を判断する 325
4 1日の集計範囲を変更する 328
2 異常値を検出する 330
1 データの分布を計算する 331
2 クローラーを除外する 333
3 データの妥当性を確認する 337
4 特定IPアドレスからのアクセスを除外する 339
3 データの重複を検出する 347
1 マスタデータの重複を検出する 347
2 ログの重複を検出する 350
4 複数のデータセットを比較する 357
1 データの差分を抽出する 358
2 2つのランキングの類似度を計算する 362

Chapter 8 データを武器にするための分析術 367
1 検索機能を評価する 368
1 NoMatch率とそのワードを集計する 370
2 再検索率とそのワードを集計する 373
3 再検索ワードを分類して集計する 377
4 検索離脱率とそのワードを集計する 380

5 検索ワードに関する指標の集計を効率化する 383
6 検索結果の網羅性を指標化する 384
7 検索結果の妥当性を指標化する 390
8 検索結果の順位を考慮した指標を計算する 393
2 データマイニング 399
1 アソシエーション分析 400
3 レコメンド 406
1 レコメンドシステムを広義に捉える 407
2 このアイテムに興味がある人はこんなアイテムも見ています 410
3 あなたにオススメの商品 417
4 レコメンドシステムを改善するポイント 421
5 表示時におけるポイント 424
6 レコメンドに関する指標を知る 425
4 スコアの計算 427
1 複数の値をバランスよく組み合わせてスコアを計算する 427
2 値の範囲が異なる指標を正規化して比較可能な状態にする 434
3 各データの偏差値を計算する 438
4 巨大な数値の指標を直観的に分かりやすく加工する 442
5 独自のスコアリング方法を定義してランキングを作成する 447

Chapter 9 知識に留めず行動を起こす 451
1 データ活用の現場 452
1 データの活用方法を考える 454
2 データに関わる登場人物を知る 457
3 ログフォーマットを考える 459
4 データを活用しやすい状態に整える 463
5 データ分析のプロセスを習得する 466
6 分析のはじめの一歩を踏み出す 470
7 相手の職種・役職に応じたレポートを作成する 472
8 さらなるデータ活用のスキルを磨く 475
9 ビッグデータ時代のデータ分析者に向けて 479

索引 480
謝辞 484

Chapter 1

ビッグデータ時代に求められる分析力とは

データを取り巻く環境が時代と共にどのように変化し、
ビッグデータの登場でどのような課題が
浮き彫りになっているのか執筆陣の経験から紐解き、
ビッグデータ時代を生き抜くための
スキルと分析力とは何かを定義します。

1-1 データを取り巻く環境の変化

アクセス解析ツールの登場

昨今では、「ビッグデータ」や「データサイエンティスト」などの単語をよく見掛けますが、2010年頃までは大きな話題になることはなく、「アクセス解析」や「Webアナリスト」が注目されていました。

著者のデータ関連の業務を振り返ると、データらしいデータに触れた経験は、2000年頃に自らのWebサーバに設置したアクセス解析プログラムがはじめてです。Webサイトへのアクセスでサーバ上にログファイルを出力し、ページビューの推移を集計したり、ログファイルに記録された時刻やURLを閲覧する程度のものでした。

当時、さまざまなアクセス解析ツールが登場しましたが、特に影響が大きかったのは2005年11月に登場した「Google Analytics」です。登場後は計測用のタグをWebサイトに埋め込み、外部でのデータの蓄積・集計が主流となりました。

Google Analyticsでは訪問回数やページビューのみならず、流入キーワードや離脱率、直帰率など、それまで把握できなかった多種多様な指標が提供され、それまでは、ユーザー情報や売上など確実なデータに基づく戦略しか立案できませんでしたが、多くの指標が提供されたことで、Web上のユーザー行動も重要な要素になりました。

もちろん、当初は各種指標の評価方法が分からず手探りでしたが、徐々にアクセス解析ツールによる改善事例やベストプラクティスの発表、意見交換や情報共有が盛んに行われ、専門的な知識を持つWebアナリストと呼ばれる職種が注目を浴びたのもこの時期です。

データを活用する動きはさらに活発になり、Google Analytics以外のアクセス解析ツールも多数登場し、広告分野でもさまざまな仕組みが開発されますが問題が発生します。多種多彩なツールの開発が進み、その利用を推進した結果として、アクセス解析ツールや広告管理ツール、自社専用の集計ツールなど、それぞれの元にデータが点在する状態となることです。

そのため、各データを横断して分析することは困難でそれぞれのツールで分析して、緩やかにそれぞれの関連性を見る程度に留まざるを得ませんでした。結果的に、ユーザーデータや購買データなどの業務データに加え、アクセスログなども一箇所に保存して、横断的な分析を可能にする環境を望む声が高まります。そうして、「ビッグデータ」が脚光を浴びることになったのです。

ビッグデータの登場

ビッグデータに関して、ウィキペディア日本語版（Wikipedia）には下記の通り記載されています。

> ビッグデータ（英: big data）とは、市販されているデータベース管理ツールや従来のデータ処理アプリケーションで処理することが困難なほど巨大で複雑なデータ集合の集積物を表す用語である。その技術的な課題には**収集、取捨選択、保管、検索、共有、転送、解析、可視化**が含まれる。大規模データ集合の傾向をつかむことは、関連データの1集合の分析から得られる付加的情報を、別の同じデータ量を持つ小規模データ集合と比較することにより行われ、「**ビジネスの傾向の発見、研究の品質決定、疾病予防、法的引用のリンク 、犯罪防止、リアルタイムの道路交通状況判断**」との相関の発見が可能になる

https://ja.wikipedia.org/wiki/ビッグデータ

上記の通り、データを収集し集計し可視化することで、ビッグデータはさまざまな分野で活用されています。もちろん、ビッグデータが注目される以前から行われていた部分もありますが、下記に挙げる技術的進歩の後押しでより注目されています。

- 記憶容量の大容量化
- ハードウェアの低廉化とクラウド技術の登場
- 分散処理を担当するミドルウェアの登場

Web界隈では分散処理を担当するミドルウェアとしてApache Hadoop、クラウドサービスではAmazon RedshiftやGoogle BigQueryなどが登場し、従来のアクセスログに加えて、自社の持つデータ、その他のツールで取得したデータをすべて一元管理し、従来は触れることが困難だった分析が可能になっています。そして、下記に挙げる新たな用途が見出されています。

- ユーザー個々の行動把握
- ユーザーの趣味嗜好の類推（レコメンド技術）
- 広告配信におけるターゲティング
- 予測モデリング
- 機械学習

アクセス解析の領域を越え、データそのものを武器に戦う企業が増加しています。新たな取り組みから得られた事例やベストプラクティスが共有され、同様の活用に期待を馳せ、従来のアクセス解析ツールから脱却し、自社でビッグデータ分析基盤を保有する動きが活発になっているのが昨今です。

ただし、現場に求められるスキルや役割も多様化しています。本章では、この新たな動きを取り込む中で、エンジニアと分析担当者の混乱や問題点を解説します。

1-2 さまざまな課題

昨今の流れに後押しされて、仮にある会社でビッグデータの分析基盤を導入することが決定したとしましょう。前述の通り、ビッグデータの分析基盤を導入する場合は、まずはアクセスログや自社の持つデータを1箇所に保存することになります。

続いて、データにアクセスする手段は、ビッグデータを管理するミドルウェアの付属ツールを使用します。例えば、HadoopであればHue、Amazon RedshiftであればAginityなどが該当します。これらのツールを用いて、取得したいデータの条件や形式をSQLで表現し取得するのが、ビッグデータに触れる最初のステップになります。

1-2-1 分析担当者の課題

アクセス解析の技術を保有する分析担当者が、この状態でビッグデータの分析基盤を取り扱うには、さまざまな問題が発生します。

1. 今までは指標はツールが提供していたが、自ら指標を決める必要がある
2. データを取り出す手段がSQLでの抽出となり、SQLの習得が求められる
3. SQLを習得できていない状況では、そもそも何を集計できるのか見当が付かない
4. 求められるレポート作成に十分なデータやツールが揃っていてもSQLが未熟ゆえに作成できない
5. レポートでの施策検討よりもSQLの試行錯誤に時間を取られ、施策検討の質が担保できなくなる

従来は、管理画面でのマウス操作やテキスト入力などでデータを抽出していた部分が、SQLと呼ばれる言語に置き換わり、不慣れなために今までとは違う苦労を背負うことになります。求められるスキルの不一致に気付かず業務に取り組んだり推進すると、業務の停滞を招いたり、ダウンロードしたデータをExcelを利用して手動でレポートを作成するなど、以前より悪い状況に陥っていたりすることが稀に見受けられます。

もちろん、昨今ではBIツール（TableauやKibanaなど）も充実しているため、SQLが未熟であってもデータの可視化を手伝ってくれます。しかし、SQLに苦戦するマーケティング担当者に配慮して、BIツールを導入してもやはり問題は発生します。

1 従来はできなかった複雑な集計・分析を可能にするため、アクセス解析ツールからの脱却を目指したが、BIツールではそこまでの水準を満たせない
2 自社のDBを格納する数多くのテーブルやリレーションシップの把握が追いつかない
3 BIツールにロードする初期データの準備に、SQLが必要なケースが出てくる

上記の問題が発生するのは、ビッグデータを取り扱うために求められるスキルと従来の分析担当者の持つスキルが一致していないにも関わらず、与えられている業務は変わっていないことが原因です。また、この原因に対して学習のコストや時間を十分に与えられないまま、実務での利用と従来の水準以上のレポートを求められる状況が、こうした事態を招きます。

ビッグデータはマーケティング担当者の課題を解決する手段ではあるものの、ビッグデータ自体がそれを解決してくれる訳ではありません。

1-2-2 エンジニアの課題

ビッグデータは従来の分析担当者が持つスキルでは扱えない印象が既にあるかもしれません。そのため、SQLを利用するのであれば、分析担当者の業務を一部、サービス開発に従事するエンジニアに移譲することを検討するかもしれません。しかし、実際に任せてみるとうまくいきません。

1 レポーティングの手法・分析のノウハウがないと、単純な集計しか提供できず、従来のアクセス解析ツールの域を脱却できない
2 分析担当者や経営層がどんなレポートを望んでいるのか、どんな課題を解決したいのか分からない
3 分析に特化したSQLの存在を知らず、本来は可能なデータ抽出に苦労するケースがある

上記もやはり従来のエンジニア業務と分析・レポート業務で、そのスキルが一致しないことが原因です。例えば、サービス開発を担当するエンジニアは、条件に合致するレコードを取得するSQLや応答速度に優れたSQLを主に使用しています。また、SQLが複雑にならないようにテーブルに保存するデータ形式を工夫し、シンプルなデータ処理を心掛けています。

一方、分析担当者は大量データの加工や一括で集計するSQLを主に使用します。サービス開発を担当するエンジニアが使用する機会が少ない関数も、分析担当者は数多く使用します。この通り、エンジニアがすべて担当するには、従来とは全く異なるスキルを習得する必要があります。

そこで、エンジニアのみで担当するには無理があると判断して、分析担当者とエンジニアの二人三脚で推進しても、下記に挙げる問題が発生します。

1 分析担当者の求めるデータを抽出しても、後日似通った依頼のたびにSQLを作る作業に追われる
2 分析担当者も何が可能か分からないため曖昧な依頼となるため、最終的なアウトプットを擦り合わせる過程で何度も手戻りが発生する

最終的なアウトプットや他でも流用可能な状態にデータを整備することで、上述の事態は回避できますが、お互いの作業を理解しないまま作業を進めると、分析担当者は何度も類似する依頼を出し、エンジニアは他に使い回しができない汎用性の低いデータ抽出を実行することになります。

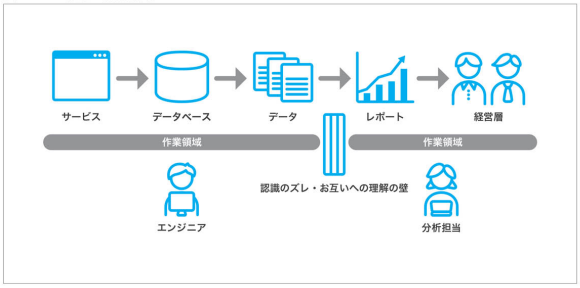
また、エンジニアが兼務しているケースでは、分析担当者は待ち時間が長くなり、従来のアクセス解析ツールの方が業務効率が高いと感じることもあり、アクセス解析ツールにアクセスするようでは元も子もありません。

エンジニアは従来の業務とは異なる分析・レポートに関するスキルと手法を早期に身に付け、依頼に素早く対応できるように学習する必要があります。ビッグデータを取り扱うには、エンジニアにとっても新しいスキルが必要なのです。

1-2-3 職種を越える横断的な分析力を身に付ける

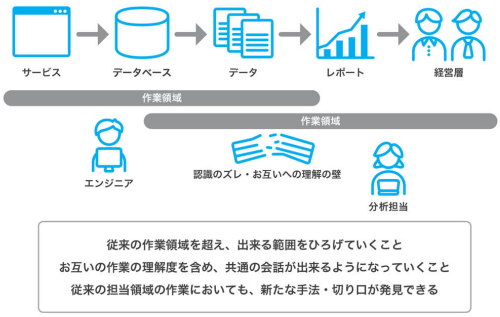
前述の通り、ビッグデータ分析基盤の導入で、エンジニアや分析担当者にさまざまな問題が降りかかるのは、お互いの持つスキルや作業範囲が異なることが原因と理解できます。もしかしたら、ビッグデータ分析基盤を導入していなくとも、相互の作業領域を十分に理解していないため、ギクシャクしている職場もあるのではないでしょうか。

図1.2.3.1：従来の作業領域



データを取り巻く環境の変化に合わせて、分析担当者もエンジニアも新たなスキルを身に付けていく必要があります。また、両者には「**従来の作業領域を超え、担当できる範囲を広げる**」ことが求められます。

図 1.2.3.2：従来の作業領域を超え、できる範囲を広げる



新しい知識を得ることで、従来の担当領域でも新しい手法・切り口を発見できるようになります。エンジニアはデータを活用し、商品の並び順の最適化やレコメンドなどの仕組みを自社サービスに導入可能になり、分析担当者はアクセス解析から脱却して、過去データの集計のみならず、未来のデータ予測が可能になるなど、それぞれ担当できる業務の幅が広がります。

本書で実現したいこと

本書では、分析担当者とエンジニアを対象に、下記の情報を提示します。

- データ加工の手法
- 分析に使用するSQL
- レポーティング・分析の手法

分析担当者であれば、アクセス解析ツールが提供する指標やフィルタがなくても、それ以上のことを自らで実践でき、より高度な人材へ成長できるように、また、エンジニアであれば、分析業務を十分に理解して、分析担当者や経営層に対して、適切な情報提供やレポート提出、そして助言できる、エンジニアの枠を超えた存在になることを目指してください。

図1.2.3.3：分析担当者に対して、本書が提供する知識、メリット

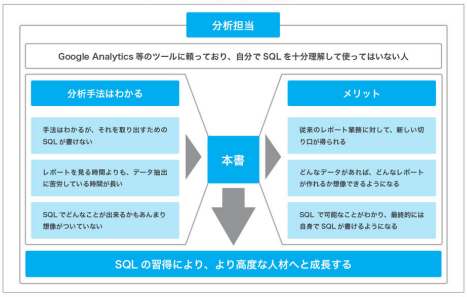
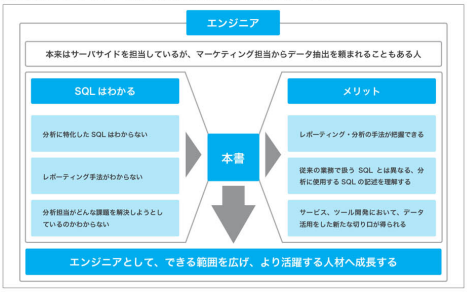


図1.2.3.4：エンジニアに対して、本書が提供する知識、メリット



本書は、分析担当者とエンジニア双方に新しい知識を提供し、相互間の通訳的な役割となり、双方がビッグデータを扱う上で必要な知識を習得し、業務を遂行できるスキルを身に付けることを目的としています。最終的には、分析担当者とエンジニアの枠を越えて、自らデータを取り扱って施策を考案し、事業を良い方向に導く専門的な立場の人材となることを望んでいます。

過去によく耳にしたWebアナリストでもなく、昨今話題のデータサイエンティストともまた違う、オールマイティかつ必要不可欠な人材として、新たな道を切り開いていく手助けとなることを願います。

Chapter 2

本書で扱うツールとデータ群

本書で扱うビッグデータ分析基盤システムの概要と、
扱うデータの種類を解説します。
代表的なビッグデータ処理基盤システムを解説し、
メリットとデメリットを紹介しながら最適なツールの選択をサポートします。
また、取り扱うデータの種類とその性質を解説することで、
データ分析作業の手戻りやミスリードを防ぐことを目指します。

2-1 システム

昨今はビッグデータ活用ツールやサービスが数多く提供され、どこから手を付ければいいのか分からない状況です。本節では、代表的なビッグデータ処理のシステムを解説し、取り扱うデータ量や処理内容、コスト面での制約などを考慮して、最適なツールの選択をサポートします。

なお、本書で紹介するシステムは、基本的に互換性のあるSQLインターフェイスを備えているため、どのツールを選択しても、他ツールに移行するハードルはそれほど高くありません。

2-1-1 PostgreSQL

PostgreSQLは、オープンソースのRDB（Relational Database）です。さまざまなプラットフォームに対応し、GUIのインストーラを用いて簡単に手元のパソコンにインストールして利用できます。MySQLなど他のオープンソースRDBと比較すると、PostgreSQLは標準SQLへの準拠率が高く、ウィンドウ関数やCTE（WITH句）など、分析用途に必須となる構文が数多く実装されています。また、本書で紹介するミドルウェアの中でも歴史がもっとも古く、PostgreSQL特有の拡張機能が多く提供されているところも利点です。

本書で紹介するSQLのクエリは、基本的にPostgreSQL 9.6で動作するクエリを紹介します。小規模なデータの分析や、SQLの学習を目的に利用するのであれば、ローカルで比較的軽量に動作するPostgreSQLを使用することをおすすめします。

2-1-2 Apache Hive

PostgreSQLなどのRDBで取り扱いが破綻する大量データを処理する上で、最もボトルネックとなる点は何でしょうか。数テラバイト～数ペタバイトまでのデータを保持する場合、通常は安価なディスク上に保存します。

しかし、ディスクからデータを読み込む際の転送速度は、一般的なCPUで処理できる速度に比べて遥かに遅くなります。例えば、転送速度180MB/sのハードディスクから、5TBのデータを読み込むには、(5×1,024×1,024／180)＝58,254秒（＝16時間以上）が掛かります。

ディスクI/Oがボトルネックとなるケースで、高速にデータを処理するためのアーキテクチャとして、分散ファイルシステムが考案されています。分散ファイルシステムでは、巨大なデータを細かく分割し

て複数のディスクに分散して保存し、各ディスクから同時にデータを読み込むことで、高速な大量データ処理を実現します。例えば、先ほどの5TBのデータを100台のディスクに分散して保存し、同時に読み込めば、読み込みに要する時間は100分の1になります。

Apache Hive（以下Hive）は、HDFSと呼ばれる分散ファイルシステム上のデータを、SQLライクなインターフェイスで簡単に処理するためのシステムです。ただし、分散ファイルシステム上のデータを同時に読み込むことでディスクI/Oのボトルネックを解消しても、バラバラに配置されたデータを並列で処理できなければ意味がありません。そこで、分散ファイルシステム上のデータを並列処理する手法として、MapReduceと呼ばれるアルゴリズムが考案されました。

このHDFSとMapReduceのアーキテクチャを実装したシステムが初期のApache Hadoopです。HiveはこのHadoopエコシステムの一部であり、HiveQLと呼ばれるSQLライクな問い合わせ言語で記述されたクエリを、自動的にMapReduceのジョブに変換し、簡単に並列分散処理を実現できます。

メリットとデメリット

前述のPostgreSQLに代表されるRDBMSと比較すると、Hiveの特徴の1つとして、Hiveはあくまでもファイルベースのシステムであることが挙げられます。ファイルベースのシステムであるため、特定レコードだけの更新や削除は難しく、インデックスも基本的に存在しないため、クエリ発行時にはファイル全件を走査することになります（ただし、最近ではHiveの利用に特化して、効率的にデータにアクセスするファイルフォーマットやサブシステムなども登場しています）。

Hiveの利点として、クエリ実行時に動的にデータの定義を与えられる点があります。例えば、「Japan,Tokyo,Minato-ku」のデータに対して、「国」「都道府県」「市区町村」の3カラムのデータとして取り扱うだけではなく、1つの文字列として扱うことが可能です。これは、「具体的に何に使うか分からないが、とりあえず保持しておきたいデータ」をHDFS上に蓄積し、必要になったタイミングで動的にスキーマを定義できることを意味します。

また、データ分析のための豊富なUDF（User-Defined Function）も活用でき、SQLだけでは実現が難しい文字列処理なども簡単に実現できます。独自のUDFをJavaで実装して導入できるので、Javaで可能な処理の多くはHiveで実現できることになります。

一方、Hiveのデメリットは、あくまでもスループットを高めるためのアーキテクチャであるため、レイテンシの低い処理を求められるケースには向いていないことです。クエリ実行時に、HiveQLで記述された処理を一旦Javaコードに変換し、生成されたjarを各計算ノードに配置して処理を開始する、煩雑な手順を踏むため、簡単なクエリでも結果を得るのに数十秒待たされることも多く、場合によっては、数時間以上を要することも少なくありません。

近年では、実行エンジンをMapReduce以外のSparkやTezで実行したり、データフォーマットを最適化するなど、低レイテンシな処理を実現する方法もありますが、そのチューニングにはある程度専門的な知識が必要となるため、ハードルが高いのが現状です。

2-1-3 Amazon Redshift

Amazon Redshift（以下Redshift）は、Amazon Web Service（以下AWS）で提供されている分散並列RDBです。分散環境で並列処理される点では前述のHiveと似ていますが、Hiveがファイルベースのバッチ処理をSQLライクなインターフェースで実装できるシステムであるのに対して、Redshiftはれっきとしたリレーショナルデータベースです。したがって、行単位での更新や削除も可能で、トランザクションもサポートされています。また、Redshiftの接続インターフェースはPostgreSQL互換のため、PostgreSQL用のODBC／JDBCドライバや、psqlクライアントでそのままRedshiftに接続することが可能です。

メリットとデメリット

Redshiftは通常のRDBでは扱えない大量データに対して、インタラクティブにクエリを発行したい場合に効果的です。HiveでMapReduceを用いた分散処理を実行する場合、どんなに軽いジョブでも10秒以上は掛かりますが、Redshiftでは最短で数ミリ秒で結果を返します。

また、AWS上のクラウドサービスであるため、比較的小さなデータから開始でき、扱うデータが多くなっても簡単にスケールアウトできる点も大きなメリットです。ただし、PostgreSQLやHiveがオープンソースで無料であるのに対して、Redshiftは利用時間に応じて料金が発生します。最小構成で構築した場合でも、1年間利用し続ければ100万円以上のコストが掛かるため、基本的には商用利用が前提となります。

一方、Redshiftのデメリットとして、パフォーマンスチューニングや金額を抑えた運用を行いたい場合には、最適なノード数やノードのスペック見積り、インスタンスの起動・終了の管理などが必要となり、ある程度の専門的な知識が必要となる点が挙げられます。Redshiftを大規模に活用することを検討するのであれば、Redshiftを専門的に管理する人材が必要になります。

また、PostgreSQLとのアーキテクチャ的な違いとして、Redshiftは列指向のストレージである点が特徴的であり、テーブル設計やクエリ実行時に、通常のRDBとは異なる発想が求められます。

列指向のストレージとは、テーブルのデータを物理的に保存する際、レコードごとに保存するのではなく、列のデータごとに保存するアーキテクチャです。このアーキテクチャにより、データ圧縮率を向上させたり、クエリ実行時のディスクI/Oを削減できます。したがって、Redshiftのテーブル設計は、通常のRDBにおける正規化をあまり行わず、分析に必要なデータをすべて1つのテーブルの列に追加する形式を取ることが多々あります。クエリ実行時にも、「SELECT *」などのすべての列を取得するクエリはパフォーマンスを低下させるため、必要な列に絞ってクエリを実行することが必要になります。

2-1-4 Google BigQuery

Google BigQuery（以下BigQuery）は、ビッグデータ解析のためのGoogle上のクラウドサービスです。クラウドサービスである点では前述のRedshiftと似ていますが、Redshiftと違い、自分で計算ノードのインスタンスを管理する必要はなく、利用時間ではなく読み込んだデータ量で課金される点が特徴的です。この特徴により手軽に利用を開始でき、扱うデータ量が少ない間は低コストで運用することが可能です。また、有料版Google AnalyticsのデータをBigQueryで取り扱い可能であったり、Google Cloud Storageから手軽にデータをロードできるなど、他のGoogleサービスとも親和性が高いため、Googleサービス利用者にとってはメリットが大きいでしょう。

特徴とデメリット

BigQueryのインターフェイスもSQLライクなクエリ言語ですが、BigQueryのクエリは「レガシーSQL」と「スタンダードSQL（標準SQL）」の2種類があります。元々BigQueryが提供していたインターフェイスがレガシーSQLですが、FROMに複数テーブルを指定するとUNIONになったり、COUNTが概算値であるなど、SQL標準と異なる挙動があり、戸惑う点が多くありました。

しかし、2016年にはSQL標準に準拠したスタンダードSQLが登場し、CTE（WITH句）や相関サブクエリに対応するなど、一般的な構文でクエリを記述可能となっています。執筆時ではレガシーSQLがデフォルトですが、クエリ実行時のオプションやアノテーションで簡単にスタンダードSQLに切り替える可能です（2017年3月現在）。ちなみに、本書では他のミドルウェアとの互換性を保つため、BigQueryのクエリはすべてスタンダードSQLを用いて記載しています。

また、BigQueryのデメリットとして、読み込んだデータ量で課金される、その料金体系から予想外の利用料となってしまう可能性がある点が挙げられます。たとえ、クエリの結果が1行だけであっても、その結果を計算するために大量のデータをロードする必要がある場合には、データ量に応じた金額が発生します。そこで、クエリ発行時のデータロードを抑えるために、一緒に読み込まれやすいデータの範囲ごとにテーブルを分割したり、必要なカラムだけ選択してSELECTするなどのテクニックが必要となります。BigQueryも前述のRedshiftと同様に、列指向ストレージのアーキテクチャであるため、クエリ実行時には必要なカラムに絞り込むことが有効です。

2-1-5 SparkSQL

SparkSQLとは、MapReduceに続く分散処理フレームワークであるApache Sparkの機能のうち、SQLインターフェイスに関連する機能を指す言葉です。Sparkはオープンソースのフレームワークであるため、無料で利用できます。これまでのオープンソースのプロダクトの中でも非常に早いスピードで

開発が進んでいるプロダクトであり、機械学習やグラフ処理、リアルタイムストリーミング処理など、さまざまな処理を手軽に分散処理できる機能を提供しています。

メリットと特徴

Sparkの最大のメリットは、ビッグデータ活用に関連するほとんどの処理をオールインワンで実現できることにあります。ビッグデータ活用のステップは一般的に複雑な工程を経ることが多いです。例えば、データの取得をPythonのスクリプトで行い、収集したデータをSQLで整形し、Rで統計分析や機械学習にかけ、結果をエクセルに出力してレポートにまとめる、といった工程です。

Sparkは、インターフェイスとしてSQLだけでなく、PythonやScala、Java、Rなどのプログラミング言語に対応し、データのインポートやエクスポート機能も豊富に実装されており、上記の工程のほとんどをSparkのプログラム上で実行できます。

また、2016年にリリースされたSpark 2.0では、DataFramesと呼ばれるAPIが標準となり、どの言語を用いても類似する記述で実装でき、言語ごとに異なっていたパフォーマンスも自動的に最適化されるようになりました。さらに、SQLインターフェイスもSQL:2003に準拠した形で大幅に機能が拡張され、HiveQLと遜色ないレベルでSQLを実行可能となっています。

DataFramesのAPIは、SQLに似た宣言的な構文でデータを操作できるだけでなく、手続き型のプログラミングに近い方法でプログラムを実装できます。例えば、中間データを逐一標準出力に書き出して、途中経過を確認しながら処理を実行したり、データ処理を細かいモジュールに分割して単体テストやアノテーションを実装できるメリットに加え、機械学習や統計処理、グラフ処理など、SQLだけでは難しい処理を統合したり、リアルタイムに受け取ったデータをストリーム処理するなど、多種多様なメリットを享受できます。ただし、Sparkのこれらの機能は本書で扱う領域を大きく超えてしまうので、本書ではSparkSQLのインターフェイスに絞った形で紹介します。

2-2 データ

レポート業務を担うに当たって、どのようなデータを取り扱うのか、どのような性質を持つデータなのかを把握せずに業務を進めると、誤ったレポートを提出したり、レポートの受け取り側を混乱させてしまいます。また、業務の前にどんな種類のデータを持ち合わせているのかを把握することで、可能なことと不可能なことの見当が付くため、手戻りなくスムーズに業務を進めることができます。

本節では、どのようなデータがあるのか整理しましょう。

2-2-1 データの種類

マーケティング担当者が扱うデータを大きく2つに分類すると、業務に必要なデータと業務に直接必要ではないが、分析のために取得しているデータがあります。本書では、前者を「業務データ」と呼び、後者を「ログデータ」と呼びます。

業務データ

本書では、下記を業務データと呼びます。

- サービス・システムを運用する目的で構築されたデータベースに存在するデータ

重要なことは、業務データのほとんどが「更新型」のデータであることです。例えば、商品が追加されたら、新しいデータとして1行が挿入されますが、その商品の価格が変更された場合は、データを挿入するのではなく、該当データを書き換えます。

そして、業務データには、「トランザクションデータ」と「マスタデータ」の2つがあります。

トランザクションデータ

トランザクションデータは、購入データやクチコミデータ、ゲームのプレイデータなど、サービス・システムを通じて、ユーザーの行動を記録したデータを指します。データは日付、時刻、マスタデータの会員ID、商品ID、数量、価格等が含まれていることが多いです。これらのデータはユーザーや運営の都合で更新されたり、削除されたりすることがあります。

このトランザクションデータを用いて、レポート業務を行うことは可能です。ただし、会員IDと商品IDで格納されているケースが多く、会員の性別や居住地、商品のカテゴリや商品名などをレポートとして扱うことはできません。これらを用いたレポートを作成する場合は、後述するマスタデータが必要になります。

マスタデータ

都道府県マスタ、カテゴリマスタ、商品マスタなど、サービス・システムが定義するデータをマスタデータと呼びます。会員に関する情報もユーザーマスタとして、マスタデータに分類されます。

前述のトランザクションデータは、会員IDや商品コード値が格納されており、トランザクションデータのみを確認しても、売れ筋商品の名称や商品カテゴリは分からない状態になっていることが多いです。そこでトランザクションデータの商品IDとマスタデータを突き合わせることで、商品名や商品カテゴリ、発売日などが明らかになり、レポート業務の幅が広がります。

トランザクションデータのみでは、分析範囲が限定されてしまうため、トランザクションデータに含まれるマスタデータはレポート業務の前に、一通り揃っていることを確認しておくことが重要です。

ログデータ

本書では、下記をログデータと呼びます。

- 集計・分析を主な用途として設計されているデータ
- サイトに特定のタグを埋め込んで送信するデータ
- 特定の行動に対して、サーバー側で出力するデータ

エンジニアであれば、ログデータとはWebサーバーのアクセスログやファイルとして読み出せるものを想定するはずですが、本書では、ファイル形式やデータベースに保存されている形式を問わず、ログデータと呼びます。また、送信形式も問いません。

重要なことは、「追記型」データであることです。追記型とは出力時点の情報を蓄積していくため、ログ出力以降に、価格が変更されたり、ユーザー情報が更新されても、既に出力したデータが後から書き換えられることはありません。データを更新した後のログデータからその変更が反映されます。

続いて、業務データとログデータに関して、それぞれの特徴と注意点を解説します。

2-2-2 業務データ

ビッグデータ基盤やデータ解析基盤と呼ばれる類のシステムを持ち合わせていない場合、最初に触れるデータがこの業務データになります。エンジニアであれば、業務データの性質は十分に理解しているはずですが、分析担当者が業務データに触れるには、いくつかの前提を理解する必要があります。本節では、業務データの特徴や扱いを理解しましょう。

業務データの特徴

1 データの精度が高い

業務データは連続したデータ処理の途中で何らかの不具合が発生した際には、ロールバックしてその処理を取り消すことが可能で、正常に処理できたものが保存される仕組みを持っています（トランザクション機能）。そのため、正確な値であることが求められる、売上に関するレポートはこの業務データを利用します。

2 更新型のデータゆえ、抽出結果は時期によって変わる可能性がある

業務データは日々、さまざまなデータの追加、更新、削除が実行されています。例えば、下記に挙げるデータ更新が考えられます。

- ユーザーが退会した場合、データを物理的に削除
- 注文をキャンセルした場合、ステータスの変更、フラグ更新による論理削除
- 引っ越しして居住地が変わった際のユーザー情報の更新

上記の通り、データが変更されたり削除される場合があるので、データを抽出するタイミングによって、その結果が変わることを意識しましょう。

3 扱うテーブルの数が多い

多くのサービスは、データベースにRDBを採用しており、データの冗長性を排除し、データの整合性を保ちやすくするよう、データを保存します（正規化）。そのため、1つのテーブルを参照するだけでは、そのデータが誰の行動なのか、何を購入したのかなどが把握できません。業務データを扱う際には、ER図と呼ばれるデータの構造を示す設計書類を理解し、複数のテーブルを結合し、データの全容を探ることになります。

業務データを蓄積する方法

業務データを分析用の環境に転送するには、「Apache Sqoop」などのRDBからデータロードを行うシステムを利用したり、自前でシステムを構築してデータを蓄積しますが、本項ではその転送方法は割愛して、どのようにデータを蓄積すべきかをまとめます。既に業務データが解析できる状況にある場合、上述の特徴の他に、どのように蓄積されているデータかを理解した上で扱いましょう

1 全レコードを入れ替える

マスタデータのように更新・削除が少なく、日次でデータが積み上がるものではない場合、そのテーブルの全レコードを入れ替え、最新の状態にしておきます（例:都道府県マスタ、商品カテゴリマスタ）。

頻繁に更新されるテーブルや、日付が経過すると状態が変化するテーブル（例：ユーザーの有料サービス申し込み）で、全レコードを入れ替えてしまうと、常に最新の状態が保存されているため、レポート作成時の状態を説明することは可能ですが、過去にどのような状態であったかは説明できません。

2 全レコードのスナップショットを日付ごとに保存する

マスタデータであっても、ユーザーマスタなどのデータは、日次で状態が変化する可能性があります。前述の「業務データの特徴」で触れた通り、出力結果が抽出時点のデータとなると、信頼性が低くなってしまいます。データ容量との兼ね合いもありますが、例えば、2016年1月1日時点のユーザーマスタ、2016年1月2日時点のユーザーマスタと、全レコードを日付ごと積み上げることで、その影響を低くすることが可能です。

3 前日からの差分のレコードを積み上げる

トランザクションデータの中で、更新・削除がなく追加のみされるテーブルは全レコードを入れ替えても構いませんが、データ転送量が膨れあがり処理時間を要するため、前日からの差分データのみを積み上げるのが良いでしょう。

業務データの扱い

業務データの特徴、どのように蓄積されているのかを理解した上で、業務上はどのように扱っているか、扱う際にどのようなことを意識しているのかを紹介します。

1 主に売上額、ユーザー数など、正しい値を求められる場合に活用する

トランザクション機能によってデータが正しい状態であることを前提に、データを扱えるため、SQLが間違っていない限り、抽出結果は正しいものになります。ログデータは送信方法によっては欠損する可能性があるので、正しい値を求められるシーンでは、業務データを利用しましょう。

2 サービスにおける訪問回数、ページビュー、回遊などのデータ分析はできない

例えば、購入データでは、どの端末で購入したかはサービス上は必要ないデータと判断した場合、ユーザーエージェントはあえて保存しません。また、どのページを閲覧したか、閲覧時のユーザーエージェント、アクセス元のIPアドレスなどのデータを逐次保存すると、サービス処理に影響が出るため、この種のデータは保存しないことが多いです。

そのため、サイトの訪問回数や回遊状況を分析するには、ログデータで別途管理して取り使えるようにする必要があります。

3 更新型ゆえ、抽出結果が変わる可能性があることを意識する

業務データでは、データの整合性に関してトランザクション機能で保証されており、正しい値ではありますが、更新型のテーブルであるため抽出時期によって結果が変わることがあります。また、データがどのように蓄積されているのかを理解・把握した上で分析を行ないましょう。

逆に言うと、更新型の影響を最小限に抑えるように、データの蓄積を検討する必要があります。どうしても更新型の影響を抑えきれず、無視できない誤差になる場合は、「抽出時点の情報を元に」と予め共有した上で、レポートを作成・提出する方が、後から混乱が生じることはないでしょう。

業務データは基本的に正しくはありますが、それはサービスを提供する際に必要な整合性を保っているに過ぎず、レポート要件を満たすための整合性を保つものではありません。レポート要件に対して、どのようにデータを蓄積していくのが適切か検討もしくは見直しましょう。

2-2-3 ログデータ

例え自社でデータ解析基盤を構築しても、ログデータの特徴を理解せずにデータを扱うと、「既存のシステム、他のツールと数値が異なる」、「どちらが正しいのか」などの議論から脱却できません。多くのアクセス解析ツールはログデータを集計し、閲覧するためのツールを提供しています。ログデータがどのような特徴を持っているのかを理解することで、既存ツールへの理解も深まりますし、レポートの要件に対してデータが十分かどうか検討するためにも役立ちます。

ログデータの特徴

1 時刻、ユーザーエージェント、IP、URL、リファラー、Cookieなどの情報を保存する

業務データは、サービス・システムを構築する際には必要のないデータは積極的に保存しませんが、ログデータでは、サービスの処理への影響が少ないことから、ユーザーエージェント、IPアドレス、URL、リファラー、Cookieなどの情報を積極的に保存します。

2 取得方法によって、ログデータの精度が異なる

ログをどのように取得しているか、集計対象となるデータが何であるかを意識せずに使用すると、誤った判断を下してしまう場合があります。後述の「ログデータを蓄積する方法」で詳しく解説します。

3 追記型のデータゆえに過去のデータを更新しない

出力時点の情報を記録しているので、商品の価格を変更しても、過去のログデータが変更されることはありません。

ログデータを蓄積する方法

取り扱うデータがどのようなログデータであるのか把握し、以降のデータの扱い方を検討したり、レポートで説明する範囲を明確にする必要があります。ログデータを送信・蓄積するには、大きく2つの方法があります。

1 タグ、SDKを埋め込んでユーザー端末からデータを送信し出力する（ビーコン型）

Google Analyticsなどのように、HTMLに特定タグを埋め込んでデータを送信する形式を指し、Webサイトでは一般的な方法です。外部のツールを利用してデータを参照することが多いのですが、昨今はそのデータを独自で扱えるように環境を整備したり、そのデータ自体を購入して分析している現場もあります。

WebサイトでJavaScriptを介して送信する場合、JavaScriptを解釈できないクローラーやブラウザのデータは送信されず欠落します。すべてのクローラーが解釈できない訳ではありませんが、クローラーの影響を受けにくい方法です。

2 サーバー側でデータを取得し出力する（サーバー型）

タグを埋め込まずに、サーバー側のプログラムでデータを出力します。この場合は、サーバーのリクエストがあった際に出力するため、クローラーのアクセスでも意図的に除外する実装ではない限り出力されます。出力を除外する実装でも、日々クローラーは追加されていくので、出力時点ですべてを除去することは困難です。

そのため、このデータを使用すると、ユーザーの行動を集計・分析したいのに、クローラーの意図しないデータが混入し、誤った判断を下してしまう場合があります。意図しないログは除去して、ログの精度を高める必要があります。

ログデータの扱い

ログデータの特徴と蓄積の違いを理解した上で、業務上はどのように扱っているか、扱う際にどのようなことを意識しているのかをいくつか紹介します。

1 主にサイトの訪問回数、ページビュー、回遊状況を集計・分析するために使用する

業務データでは管理しきれない閲覧ページ、リファラー、ユーザーエージェントなどを保存することが可能なので、アクセス解析ツールで確認する訪問回数やページビュー、アクション数、デバイス別訪問者数などの指標を求める際に使用できます。

2 出力時点の情報で分析可能だが、最新の状況を考慮した分析には向かない

追記型のデータゆえにログ出力時点の情報が記録されているため、当時の状況における分析には利用可能ですが、ログ出力以降のデータ変更を適用して分析するには、データの加工が必要になります（例：商品カテゴリの見直し、ユーザーの居住地変更など）。

3 追記型のデータゆえ、抽出結果が変わる状況は少ない

ログデータは更新・削除が発生しないので、期間を指定して集計したクエリの結果が変わることはありません。業務データの場合、システムによってはレコードが削除される可能性があります。

3 データの正確さについては、業務データよりも劣る

ログの取得方法によっては、欠損するユーザーが発生したり、クローラーのログが混入する可能性があるため、正確な値を求める目的には向きません。

ログデータは基本的には出力したら以降変更を加えず、ただ蓄積していくのみです。情報の正確性に欠ける部分がありますが、そのメリットとデメリットを理解した上で、業務データとどのように扱いを棲み分けるか、相互に補完可能にするかを考えてみると良いでしょう。

2-2-4 2つのデータを利用することで生まれる価値

本項では、業務データとログデータをどのように活用するべきか、また2つのデータを横断的に利用することで生まれる新しい価値、そもそものデータ利用の価値を説明します。

各データ単体での利用価値

業務データ

売上額の推移はもちろんのこと、どの商品が人気なのか明らかになれば、ユーザーにもっと周知して購入してもらうことが可能です。また、商品によっては季節性があり、ある時期になると売れるものが過去の傾向から掴めていれば、その時期には表示順を変えたり、特集を組むなどして、商品の露出を高めることで、より多くのユーザーに購入してもらうことができます。

日本には四季があり、また南北に長い形状です。この地理的な特性を活かし、3月～4月に沖縄で売れた商品が、6月～7月の本州で売れる傾向を見つけ、その年の流行を早期に掴むことで、在庫をコントロールして売上の向上を図っていると、アパレル系での事例を聞いたことがあります。季節が冬の場合は、北海道での売れ行きを確認して、本州や沖縄へと反映させる模様です。この事例はWebに限らず、店舗で売れたデータ、つまりオフラインのデータを活用することができます。

ログデータ

多くのアクセス解析ツールは、Webサイトにビーコン形式でログを送信する機構を埋め込み、そのログを利用して集計した結果をレポートとして提供しています。基本的には、ページビューやアクション、そのデータに含まれる値（リファラー、ユーザーエージェント、カスタム変数など）を集計し表示します。

ECサイト、ニュースサイト、コミュニティサイトなど、多種多様なサービスがありつつも、アクセス解析ツールのレポートは、どのサービスでも共通で利用できるフォーマットで設計されています。

もちろん、ある程度のカスタマイズや絞り込みは可能ですが、レポートのフォーマットや計算結果を自ら考えて集計するには、その機能が実装されていない限りは困難です。ある程度までは可能ですが、レポート業務がアクセス解析ツールの搭載機能で制限されてしまいます。

ビッグデータ基盤があれば、自ら欲しいレポートを自由に定義でき、データの収集から加工、集計を自由に行えるので、アクセス解析ツールの制限を受けることはありません。

2つのデータを利用することで生まれる新しい価値

ログデータは主にWebサイトでの行動を記録し活用します。上述で触れている通り、業務データはWebサイトに限らず、オフラインでのデータも利用できます。POSデータや来店成約に至ったデータ、対応履歴などさまざまなデータが考えられます。

ログデータと業務データを横断して集計することが可能になれば、Webサイトでの行動がオフラインの行動にどう影響するか、調査することが可能になります。例えば、特定のメディアや広告から遷移してきたユーザーは成約に至る可能性が高いことが分かれば、そのメディアや広告への出稿額を上げることで、より多くの成約を得ることが可能になります。来店前にWebサイトでどの商品を閲覧していたかが分かれば、来店時の接客にも活かすことも可能です。

Webサイトからオフラインへ誘導するサービスの場合は、従来はログデータもしくは業務データのいずれか一方のみしか利用していないのであれば、分析の可能性は大きく広がります。

データの利用価値

いずれのデータでも、ユーザーやサービス・組織に対して価値の提供が行えない限り、データ活用の意味がありません。どのようにすれば、価値のあるデータ分析ができるのか、世の中に出回るデータ活用の事例を、下記の通りまとめてみました。

■ 目標を管理し、計画立てて、サービス・組織の成長を促す（目標管理）

売上、アクセス数、ユーザー数など、サービスが目指す目標に対して、現時点の進捗はどれぐらいかを把握して、不足していれば、達成できるように施策を検討・実施し、サービス・組織の成長を促すことが可能です。

■ ユーザーの行動から傾向を発見し、売上やサービスの改善を計る（サービス改善）

ユーザーインタビューをその都度実施することは、時間も費用も掛かります。さらに、そのインタビューが潜在的なニーズを明確にしてくれるとは限りません。また、サンプル数が少なく、全体に適応可能なのかどうか判断することは困難です。そのため、大量のデータの中からユーザーの傾向を探り、売上向上やサービス改善が可能なことがデータの価値といえます。

■ 過去の傾向から、未来の行動を予測する（未来予測）

季節と地理的な特性を活かす、前述のアパレル系での事例の通り、データを活用することで将来の在庫をコントロールすることが可能です。Webサイトで特定の行動を取るユーザーは近い将来サービスを退会する、契約を終了する傾向があると判明すれば、事前に対策を施すことが可能です（予兆検知）。また、「あなたにオススメの商品」などのレコメンドも未来予測の１つといえます。

上記に挙げる事例がデータ利用の価値であり、ビッグデータが注目を浴びる所以と考えます。このようなことが可能になるデータがどんなデータであるかを検討し、データを収集しましょう。

本書では、業務データとログデータを中心に進めますが、現状の把握や未来予測を行うために必要な要因として、大学や公的機関が提供している天気や気温などのオープンデータが役立つのであれば、必要に応じて外部データも収集しましょう。

1%の当選確率なら100回やれば1回当たる?

ソーシャルゲームのガチャで、「レアなキャラクターが当たる確率は1%」と聞くと、100回に1回は当たる!と想像する人も多いのではないでしょうか。

玩具店や百貨店などの一角にあるカプセルトイは、100個中1個の当たりがある場合、カプセルが全部なくなるまで引けば必ず当たりますが、ソーシャルゲームのガチャはカプセルトイとは異なり、引いたものを除外せず常に同一の確率が適用されます(「ボックスガチャ」は例外です。詳細な説明は省きます)。

当選確率1%のガチャを1万人が100回引いたらどうなるか、プログラムを作って検証してみましょう。

当選回数	0回	1回	2回	3回	4回	5回	6回	合計
人数	3,654	3,699	1,845	601	157	38	6	10,000
構成比	36.5%	37.0%	18.5%	6.0%	1.6%	0.4%	0.1%	100%

上表がその構成比ですが、100回ガチャを引いても一度も当たらない人が36.5%もいます。100回に1回は当たるだろうと想像していた方には意外な結果ではないでしょうか。

100回のガチャで1回以上当たる確率は、ハズレを引き続ける確率を求めて、全体(100%)から引くことで算出できます。

> 100回引いて1回以上当選する確率 = 全体 - ハズレを引き続ける確率

ハズレを引き続ける確率は99/100の100乗≒0.366となります。上表の一度も当たらない人が36.5%もいることは、ほぼ計算通りの結果だったのです。全体からハズレを引き続ける確率を引くと、1−0.366=0.634となります。したがって、100回ガチャを引いて1回以上当たる確率は63.4%となります。

運営側はこの確率計算を意識しないと、予期せぬゲームバランスとなったり、お目当てのキャラクターがなかなか出現せず、ユーザーのストレスを貯めてしまう結果になるので、注意が必要です。

Chapter 3

データ加工のためのSQL

データ分析の前に本書で扱う基本的なSQLを紹介します。
基本的な値の操作や集計を中心に
本書で扱う基本的なSQLを学び
分析で扱いにくいデータ形式への対応方法や
分散しているデータへの対応方法などを理解しましょう。

3-1
1つの値に対する操作

取り扱うデータの一部は元来の用途が分析ではなく、他の用途向けに準備されているものがあり、分析するシーンでは取り扱いにくいものもあります。取り扱いにくい状態のデータであっても、加工して分析可能にすることもエンジニア（分析担当者）の仕事の1つです。

本節では、取り扱うデータが分析に適切な形式ではない場合の対応方法として、1つの値に対するデータ加工を中心に説明します。

データを加工する必要性

データの加工が必要なシーンが、どのような理由で発生するのか整理してみましょう。

扱うデータが分析用途として想定されていない場合

業務データを取り扱う場合、データベースにコード値を格納し、そのコード値は他のテーブルが意味付けを管理しているケースもあれば、プログラム側で管理しているケースもあります。この場合、コード値のままレポートを作成しても、そのレポートから状況を正確に読み取ることは困難です。

また、アクセスログはある行動を1行のログで表現するケースが多々あります。例えば、その慣例に従って、複数の情報を1行で表現する際に、カンマなど特定の文字列で区切ってデータを保存すると、カンマ区切りのデータはSQLでは取り扱いが難しいため、分析に適したデータ形式に加工する必要があります。

時には、保存データの粒度が細かすぎて、求められる単位での集計が困難なケースもあります。そんなケースでも、「データがないから集計できない」「その類いのデータはないから集計できない」と断らず、データを加工して集計を行ないましょう。

計算する際に比較可能な状態にする、エラーを回避する場合

ログデータと業務データを同時に取り扱うケースでは、必ずしもデータの型が一致しているとは限りません。両データを比較する際に型を揃える必要に迫られる場合もあります。

また、計算時では、NULLに対して演算を行なうと結果がNULLになり得ます。エラーの発生で本来得たいデータとは異なる結果になったり、必要な結果が得られない場合は、エラーを回避するためにデータを加工する必要があります。

3-1-1 コード値をラベルに置き換える

SQL CASE式

ログデータや業務データに保存されているコード値のまま集計すると、レポートの可読性が低くなったり、レポート側でコード値を置き換える必要が出てきます。予め判明しているコード値は、ラベルを付けて扱いやすくしましょう。本項ではマスタデータがないケースで、簡易的にデータ抽出するシーンを想定しています。

下記データに示すテーブルを例に、会員登録時のデバイスを格納するカラム（regist_device）に、コード値（1：PC、2：SP、3：アプリ）で保存されているものをラベル化します。

データ3.1.1.1：ユーザーマスタ(mst_users)テーブル

```
 user_id | register_date | register_device
---------+---------------+-----------------
 U001    | 2016-08-26    | 1
 U002    | 2016-08-26    | 2
 U003    | 2016-08-27    | 3
```

コード値をラベル化するなど、特定の条件に基づいて値を決定するには、CASE式を用います。

CASE式の構文は、CASE句の後に「WHEN（条件式）THEN（条件を満たす場合の値）」を列挙し、最後にENDで終わります。どの条件式も該当しない場合の値はNULLですが、すべてのWHEN句のあとに「ELSE（値）」を指定すると、デフォルト値も指定できます。

コード3.1.1.1：コードをラベルに置き換えるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id
  , CASE
      WHEN register_device = 1 THEN 'PC'
      WHEN register_device = 2 THEN 'SP'
      WHEN register_device = 3 THEN 'アプリ'
      -- デフォルトの値を指定する場合はELSE句を用いる
      -- ELSE ''
    END AS device_name
FROM mst_users
;
```

```
 user_id | device_name
---------+-------------
 U001    | PC
 U002    | SP
 U003    | アプリ
```

まとめ

CASE式は基本的なSQL構文ですが、データの変換に非常によく使用します。複雑な構文とCASE式を組合せて使用するケースも多いため、しっかりと基本を確認しておきましょう。

3-1-2 URLから要素を取り出す

SQL URL関数、正規表現

分析の現場では、サービスローンチ時にログ要件・分析要件を詳細に検討できず、ひとまず最低限の要件として、リファラーとページURLを保存しておく動きがあります。URLを分析時に取り扱うには、粒度が細かすぎるため、URLから大雑把な意味付けが必要とされるシーンがあります。

本項では、下記に示すサンプルデータを利用して、URLに対する操作を紹介します。

データ3.1.2.1：アクセスログ(access_log)テーブル

```
-[ RECORD 1 ]----------------------------------------------
stamp    | 2016-08-26 12:02:00
referrer | http://www.other.com/path1/index.php?k1=v1&k2=v2#Ref1
url      | http://www.example.com/video/detail?id=001
-[ RECORD 2 ]----------------------------------------------
stamp    | 2016-08-26 12:02:01
referrer | http://www.other.net/path1/index.php?k1=v1&k2=v2#Ref1
url      | http://www.example.com/video#ref
-[ RECORD 3 ]----------------------------------------------
stamp    | 2016-08-26 12:02:01
referrer | https://www.other.com/
url      | http://www.example.com/book/detail?id=002
```

リファラーから参照元ホスト名を取得する

参照元Webページを判別するのであれば、リファラーを集計すれば把握できますが、ページ単位では粒度が細かすぎるため、ホスト名単位で集計する処理を下記のコード例に示します。

ちなみに、HiveやBigQueryではURLを扱う関数が用意されていますが、実装されていないミドルウェアでは、正規表現でホスト名のパターンを抽出する必要があります。さらにRedshiftでは、正規表現にパーレン () を用いたグループ化が使用できないため、正規表現の記述に工夫が必要です。

コード3.1.2.1：リファラーのドメインを抽出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    stamp
  -- referrerのホスト名部分を抽出する
  -- ■ PostgreSQLの場合、substring関数と正規表現を用いる
  , substring(referrer from 'https?://([^/]*)') AS referrer_host
```

```
  -- ■ Redshiftの場合、正規表現にバーレンが使用できないため、
  --   regexp_substr関数とregexp_replace関数を組み合わせる
  -- , regexp_replace(regexp_substr(referrer, 'https?://[^/]*'), 'https?://', '')
  -- AS referrer_host
  -- ■ Hive, SparkSQLの場合、parse_url関数でホスト名を抽出できる
  -- , parse_url(referrer, 'HOST') AS referrer_host
  -- ■ BigQueryの場合はhost関数が利用できる
  -- , host(referrer) AS referrer_host
FROM access_log
;
```

```
        stamp        | referrer_host
---------------------+--------------
 2016-08-26 12:02:00 | www.other.com
 2016-08-26 12:02:01 | www.other.net
 2016-08-26 12:02:01 | www.other.com
```

URLからパスやクエリパラメーターの値を取り出す

商品に関するレポートを作成する際、どの商品が閲覧されたか特定するIDをデータとして出力していないケースでも、データにURLさえ含まれていれば、URLを加工してパスや特定キーを取り出して、商品レポートを作成可能です。下記コード例に、URLのパスとGETパラメーターにある特定キーの値を取り出すクエリを示します。

コード3.1.2.2：URLのパスとGETパラメーターにある特定のキーの値を取り出すクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    stamp
  , url
  -- URLのパスやGETパラメータのid値を抽出する
  -- ■ PostgreSQLの場合、substring関数と正規表現を用いる
  , substring(url from '//[^/]+([^?#]+)') AS path
  , substring(url from 'id=([^&]*)') AS id
  -- ■ Redshiftの場合、regexp_substr関数とregexp_replace関数を組み合わせる
  -- , regexp_replace(regexp_substr(url, '//[^/]+[^?#]+'), '//[^/]+', '') AS path
  -- , regexp_replace(regexp_substr(url, 'id=[^&]*'), 'id=', '') AS id
  -- ■ BigQueryの場合、正規表現にregexp_extract関数を用いる
  -- , regexp_extract(url, '//[^/]+([^?#]+)') AS path
  -- , regexp_extract(url, 'id=([^&]*)') AS id
  -- ■ Hive、SparkSQLの場合、parse_url関数でURLのパス部分やクエリパラメータ部分の値を抽出する
  -- , parse_url(url, 'PATH') AS path
  -- , parse_url(url, 'QUERY', 'id') AS id
FROM access_log
;
```

```
        stamp        |                    url                     |     path      | id
---------------------+--------------------------------------------+---------------+----
 2016-08-26 12:02:00 | http://www.example.com/video/detail?id=001 | /video/detail | 001
 2016-08-26 12:02:01 | http://www.example.com/video#ref           | /video        |
 2016-08-26 12:02:01 | http://www.example.com/book/detail?id=002  | /book/detail  | 002
```

まとめ

URLの取り扱いはWebサービスのログ解析で頻出する技術ですが、ミドルウェアによって関数名が異なっていたり、正規表現を記述する必要に迫られるなど、躓くポイントが多い箇所です。本書でもURLを取り扱う項をいくつも登場するので、しっかりと基礎を身に付けましょう。

3-1-3 文字列を配列に分解する

SQL split_part関数、split関数

ビッグデータ解析で最も頻出するデータ型は文字列ですが、文字列型は汎用的なデータ型のため、分析ではさらに細かい粒度に分解する必要に迫られるケースがあります。例えば、英語の文章をスペースで分割して単語に区切る、カンマ区切りのデータから値を１つずつ取り出すなどです。

本項では、前項で利用したアクセスログをサンプルに、ページ階層ごとにURLを集約することを考えます。下記コード例に、URLのパスをスラッシュで分割して階層を抽出するクエリを示します。

コード3.1.3.1：URLのパスをスラッシュで分割して階層を抽出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    stamp
  , url
  -- パスをスラッシュで配列に分解し、階層ごとに抽出する
  -- パスが必ずスラッシュから始まるので、2番目の要素が最初の階層となる
  -- ■ PostgreSQLの場合、split_partでn番目の要素を抽出できる
  , split_part(substring(url from '//[^/]+([^?#]+)'), '/', 2) AS path1
  , split_part(substring(url from '//[^/]+([^?#]+)'), '/', 3) AS path2
  -- ■ Redshiftの場合も、split_partでn番目の要素を抽出できる
  -- , split_part(regexp_replace(
  --     regexp_substr(url, '//[^/]+[^?#]+'), '//[^/]+', ''), '/', 2) AS path1
  -- , split_part(regexp_replace(
  --     regexp_substr(url, '//[^/]+[^?#]+'), '//[^/]+', ''), '/', 3) AS path2
  -- ■ BigQueryの場合、split関数で配列に分解するが、配列のインデックス指定が特殊になる
  -- , split(regexp_extract(url, '//[^/]+([^?#]+)'), '/')[SAFE_ORDINAL(2)] AS path1
  -- , split(regexp_extract(url, '//[^/]+([^?#]+)'), '/')[SAFE_ORDINAL(3)] AS path2
  -- ■ Hive, SparkSQLの場合も、split関数で配列に分解する
  --   ただし、配列のインデックスが0から始まる
  -- , split(parse_url(url, 'PATH'), '/')[1] AS path1
  -- , split(parse_url(url, 'PATH'), '/')[2] AS path2
```

```
FROM access_log
;
```

```
        stamp        |                   url                    | path1 | path2
---------------------+------------------------------------------+-------+-------
 2016-08-26 12:02:00 | http://www.example.com/video/detail?id=001 | video | detail
 2016-08-26 12:02:01 | http://www.example.com/video#ref           | video |
 2016-08-26 12:02:01 | http://www.example.com/book/detail?id=002  | book  | detail
```

まとめ

Redshiftは公式で配列型のデータ型をサポートしていませんが、split_part関数を用いて文字列を分割し、n番目の要素を取り出すことが可能です。

配列のインデックスは一般的に1から始まりますが、BigQueryの場合、配列の値にアクセスする方法が少し特殊であるため、配列インデックスを0から開始するにはOFFSET、1の場合はORDINALを指定します。また、配列長以上のインデックスにアクセスすると、通常はNULLを返しますが、BigQueryではエラーとなります。NULLを返すためには、SAFE_OFFSETやSAFE_ORDINALを指定する必要があります。

3-1-4 日付やタイムスタンプを扱う

SQL 日付/時刻関数、date型、timestamp型、文字列関数

ログデータの取り扱いでは、日付やタイムスタンプなど時間情報の扱いを避けることはできません。しかし、各ミドルウェアでは時間情報を扱うデータ型や関数にばらつきがあります。本項では、日付やタイムスタンプを取り扱う手法をまとめて説明します。

現在の日付とタイムスタンプを取得する

現在の日付とタイムスタンプを取得するクエリを、次のコード例に示します（コード3.1.4.1）。ミドルウェアごとに取得方法が少しずつ異なるので注意してください。また、同じ記述方法でも、返り値がミドルウェアごとに異なる場合もあります。

PostgreSQLでは、CURRENT_TIMESTAMPの返り値はタイムゾーン付きのタイムスタンプ型で、デフォルト設定されたタイムゾーンが付与されます。それ以外のミドルウェアではタイムゾーンなしのタイムスタンプになるため、返り値の型を合わせる場合、PostgreSQLではLOCALTIMESTAMPを用いると良いでしょう。また、BigQueryにはタイムゾーン付きのタイムスタンプ型は実装されてなく、タイムゾーンはUTC固定になります。そのため、CURRENT_TIMESTAMPで返される時刻が日本時間とずれるため、予期せぬところで躓くことがあります。

コード3.1.4.1：現在の日付とタイムスタンプを取得するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
 -- ■ PostgreSQL, Hive, BigQueryの場合は、
 -- CURRENT_DATE定数やCURRENT_TIMESTAMP定数が利用できる
   CURRENT_DATE AS dt
 , CURRENT_TIMESTAMP AS stamp

 -- ■ Hive, BigQuery, SparkSQLの場合は、
 -- CURRENT_DATE()関数やCURRENT_TIMESTAMP()関数が利用できる
 --   CURRENT_DATE() AS dt
 -- , CURRENT_TIMESTAMP() AS stamp

 -- ■ Redshiftの場合は、現在の日付にCURRENT_DATE、現在のタイムスタンプにGETDATE()を用いる
   CURRENT_DATE AS dt
  , GETDATE() AS stamp

 -- ■ PostgreSQLの場合、CURRENT_TIMESTAMPはタイムゾーン付きのタイムスタンプ型になる
 -- タイムゾーンなしのタイムスタンプを取得したい場合は、LOCALTIMESTAMPを用いる
 -- , LOCALTIMESTAMP AS stamp
;
```

```
     dt     |           stamp
------------+----------------------------
 2017-01-30 | 2017-01-30 18:42:57.584993
```

指定した値の日付や時刻データを取得する

現在時刻ではなく、文字列として指定した日付や時刻から日付型やタイムスタンプ型のデータを作成する場合、ミドルウェアごとにいくつかの方法がありますが、下記コード例に示す通り、CAST関数を用いて型を変換する方法が最も汎用的です。その他の形式に関しては、コメントを参照してください。

コード3.1.4.2：文字列を日付型、タイムスタンプ型に変換するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
-- 文字列から日付型/タイムスタンプ型に変換する
SELECT
    -- ■ PostgreSQL, Hive, Redshift, BigQuery, SparkSQLのすべてで、
    --   「CAST(value AS type)」の形式が利用できる
    CAST('2016-01-30' AS date) AS dt
  , CAST('2016-01-30 12:00:00' AS timestamp) AS stamp

    -- ■ Hive, BigQueryの場合、「type(value)」の形式が利用できる
    --  date('2016-01-30') AS dt
    --, timestamp('2016-01-30 12:00:00') AS stamp
```

```
    -- ■ PostgreSQL, Hive, Redshift, BigQuery, SparkSQLのすべてで、
    --   「type value」の形式が利用できる
    --   ※ ただし、valueは定数である必要があり、カラム名は指定できない
    --   date '2016-01-30' AS dt
    -- , timestamp '2016-01-30 12:00:00' AS stamp

    -- ■ PostgreSQL, Redshiftの場合、「value::type」の形式が利用できる
    --  '2016-01-30'::date AS dt
    --, '2016-01-30 12:00:00'::timestamp AS stamp
;
```

```
     dt     |        stamp
------------+---------------------
 2016-01-30 | 2016-01-30 12:00:00
```

日付/時刻から特定のフィールドを取り出す

タイムスタンプ型のデータから、年や月など特定のフィールドの値を取り出すにはEXTRACT関数を使用します。EXTRACT関数に対応していないHiveやSparkSQLでは、それぞれのフィールドに対応した関数を用います。下記にEXTRACT関数の使用例を示します。

コード3.1.4.3：タイムスタンプ型のデータから年、月、日などを取り出すクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
SELECT
    stamp
  -- ■ PostgreSQL, Redshift, BigQueryの場合、EXTRACT関数が利用できる
  , EXTRACT(YEAR  FROM stamp) AS year
  , EXTRACT(MONTH FROM stamp) AS month
  , EXTRACT(DAY   FROM stamp) AS day
  , EXTRACT(HOUR  FROM stamp) AS hour

  -- ■ Hive, SparkSQLの場合、EXTRACT関数の代わりにそれぞれに対応する関数を用いる
  -- , YEAR(stamp)  AS year
  -- , MONTH(stamp) AS month
  -- , DAY(stamp)   AS day
  -- , HOUR(stamp)  AS hour
FROM
  (SELECT CAST('2016-01-30 12:00:00' AS timestamp) AS stamp) AS t
;
```

```
       stamp        | year | month | day | hour
--------------------+------+-------+-----+------
 2016-01-30 12:00:00 | 2016 |     1 |  30 |   12
```

別解 日付型やタイムスタンプ型を利用しなくとも、タイムスタンプを単なる文字列と割り切ってしまい、文字列操作でフィールドを取り出す方法もあります。

下記コード例に、substring関数を用いて文字列を抽出するクエリを示します。年と月を同時に抽出して月別のレポートを作成したい、などのケースでより簡潔に記述でき、ミドルウェアごとの表記揺れも少なくて済みます。

コード3.1.4.4：タイムスタンプを表す文字列から年、月、日などを取り出すクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    stamp
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLでは、substring関数が利用できる
  , substring(stamp,  1, 4) AS year
  , substring(stamp,  6, 2) AS month
  , substring(stamp,  9, 2) AS day
  , substring(stamp, 12, 2) AS hour
  -- 年と月を同時に抽出する
  , substring(stamp,  1, 7) AS year_month

  -- ■ PostgreSQL, Hive, BigQuery, SparkSQLでは、substr関数が利用できる
  -- , substr(stamp,  1, 4) AS year
  -- , substr(stamp,  6, 2) AS month
  -- , substr(stamp,  9, 2) AS day
  -- , substr(stamp, 12, 2) AS hour
  -- , substr(stamp,  1, 7) AS year_month
FROM
  -- ■ PostgreSQL, Redshiftの場合、文字列型としてtextが利用できる
  (SELECT CAST('2016-01-30 12:00:00' AS text) AS stamp) AS t
  -- ■ Hive, BigQuery, SparkSQLの場合、文字列型としてstringを用いる
  -- (SELECT CAST('2016-01-30 12:00:00' AS string) AS stamp) AS t
;
```

```
       stamp        | year | month | day | hour | year_month
--------------------+------+-------+-----+------+------------
 2016-01-30 12:00:00 | 2016 | 01    | 30  | 12   | 2016-01
```

まとめ

日付や時刻情報はログデータに欠かせない情報ですが、タイムゾーンの考慮やミドルウェアごとの差異など、注意すべき点がいくつかありますので、本項で改めて確認しましょう。

3-1-5 欠損値をデフォルト値に置き換える

SQL COALESCE関数

文字列や数値を扱う際に、NULLが混入する可能性のあるカラムには注意が必要です。NULLと文字列を結合するとNULLになり、NULLと任意の数値で四則演算を実行するとNULLとなります。取り扱うデータは必ずしも分析に都合のよい形式ではないため、データ加工が必要になります。

購入額とNULLを含むクーポン使用額が格納されている下記データ例から、クーポン値引き後の売上金額を求めるクエリをコード例に示します。

データ3.1.5.1：クーポン付き購入ログ(purchase_log_with_coupon)テーブル

```
 purchase_id | amount | coupon
-------------+--------+--------
       10001 |   3280 |   null
       10002 |   4650 |    500
       10003 |   3870 |   null
```

コード3.1.5.1：購入額から割引クーポン値引き後の売上金額を求めるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    purchase_id
  , amount
  , coupon
  , amount - coupon AS discount_amount1
  , amount - COALESCE(coupon, 0) AS discount_amount2
FROM
    purchase_log_with_coupon
;
```

```
 purchase_id | amount | coupon | discount_amount1 | discount_amount2
-------------+--------+--------+------------------+------------------
       10001 |   3280 |   null |             null |             3280
       10002 |   4650 |    500 |             4150 |             4150
       10003 |   3870 |   null |             null |             3870
```

上記コード例では、discount_price1はpriceからcouponをそのまま減算しています。その結果、couponがNULLのレコードはNULLとなります。discount_price2はcouponがNULLの場合、0に置き換えているので、discount_price2にcouponを使用していない場合でも、正しく計算されていることが分かります。

3-2 複数の値に対する操作

データ分析では、複数の値を集約して1つの値にまとめたり、さまざまな値を比較する場面などが多々あります。本節では値を操作する目的を整理して、レコードに含まれる異なる値を組み合わせて、新しい値を計算する手法を紹介します。

新たな集計対象を定義する

元々の業務データが分析には困難な状態で保存されているケースがあります。

例えば、業務データでは郵便番号が、前半の3桁と後半の4桁に分割されて保存されている場合、後半4桁のみを分析しても有効な結果は得られません。データを結合して郵便番号として意味のあるデータに加工することで、郵便番号マスタとの照合が可能になります。

新たな指標を定義する

例えば、あるページが表示された回数をページビュー、あるページを表示したユーザー数を訪問者数と定義することができます。この2つの指標は単純にデータを集計したものです。ここから、(ページビュー)÷(訪問者数)を求めることで、ユーザー1人当たりが何回閲覧しているかを示す指標を導き出すことが可能です。

また、Webサイトでは、訪問・表示したユーザー数の中で、特定の行動(クリックや購入等)を実行したユーザー数の割合を求め、CTR(クリック率:Click Through Rate)、CVR(コンバージョン率:Conversion Rate)と呼ばれる指標を定義し、推移を確認するシーンが数多くあります。

単純に実数で比較すると数値の大きいデータしか注目されませんが、「1人当たり」や「割合」の指標を用いることで、別の視点が得られます。

3-2-1 文字列を連結する

SQL CONCAT関数、‖演算子

レポート作成では、その用途などに合わせて、複数のデータを連結して、扱いやすい形式に変換するシーンがあるので、覚えておきましょう。

本項では下記のサンプルデータを対象に、都道府県と市区町村名のカラムがあった際に、「東京都千代田区」と、文字列を連結するSQLを下記のコード例に示します。

データ3.2.1.1：ユーザーの住所情報(mst_user_location)テーブル

```
 user_id | pref_name | city_name
---------+-----------+-----------
 U001    | 東京都    | 千代田区
 U002    | 東京都    | 渋谷区
 U003    | 千葉県    | 八千代市
```

コード3.2.1.1：文字列を連結するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id
    -- ■ PostgreSQL, Hive, Redshift, BigQuery, SparkSQLのすべてでCONCAT関数が利用できる
    --   ただし、Redshiftの場合は2引数のみ対応
  , CONCAT(pref_name, city_name) AS pref_city
    -- ■ PostgreSQL, Redshiftの場合は‖演算子も利用できる
    -- , pref_name || city_name AS pref_city
FROM
  mst_user_location
;
```

▼

```
 user_id |   pref_city
---------+---------------
 U001    | 東京都千代田区
 U002    | 東京都渋谷区
 U003    | 千葉県八千代市
```

ワンポイント

多くのミドルウェアでは、CONCAT関数を用いて任意の数の文字列を結合できますが、RedshiftのCONCATは引数に2つの文字列しか取れません。したがって、Redshiftの場合は、CONCAT関数ではなく‖演算子の方が使い勝手がよいでしょう。

3-2-2 複数の値を比較する

SQL CASE式、SIGN関数、greatest関数、least関数、四則演算子

本項では、1レコードに含まれる複数の値を比較する手法を紹介します。下記のデータ例に示す四半期売上(quarterly_sales)テーブルは、各年の四半期売上(Q1～Q4)をカラムで表現したテーブルです。

2017年の第3四半期と第4四半期(Q3とQ4)は売上金額が確定していないため、NULLとなっています。このテーブルをサンプルに解説を進めます。

データ3.2.2.1:四半期売上(quarterly_sales)テーブル

```
 year |  q1   |  q2   |  q3   |  q4
------+-------+-------+-------+-------
 2015 | 82000 | 83000 | 78000 | 83000
 2016 | 85000 | 85000 | 80000 | 81000
 2017 | 92000 | 81000 |       |
```

四半期ごとの売上増減を判定する

まず、四半期ごとの売上が増加しているか減少しているかを判定してみます。SQLでは、1レコードに含まれている値であれば、SELECT句で同時に参照できます。

各カラムの値の大小を比較するためには、下記コード例のjudge_q1_q2カラムに示す通り、CASE式を用いて条件を記述し、その条件にマッチした場合の値を指定します。Q1よりQ2の売上が多い場合には「+」、同じ場合には空白、売上が少ない場合は「-」を出力します。

また、値の差分を求めるには、diff_q2_q1カラムに示す通り、単純にカラムの引き算で実現できます。この差分とSIGN関数を組み合わせると、CASE式より簡単に値の増減判定が実現できます。SIGN関数は、引数が正の値であれば1、0であれば0、負の値であれば-1を返す関数です。

コード3.2.2.1: Q1, Q2の2つのカラムの比較を行うクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    year
  , q1
  , q2
    -- Q1とQ2の売上変化を評価する
  , CASE
      WHEN q1 < q2 THEN '+'
      WHEN q1 = q2 THEN ' '
      ELSE '-'
    END AS judge_q1_q2
```

```
    -- Q1とQ2の売上額の差を計算する
  , q2 - q1 AS diff_q2_q1
    -- Q1とQ2の売上変化を1, 0, -1で表現する
  , SIGN(q2 - q1) AS sign_q2_q1
FROM
  quarterly_sales
ORDER BY
  year
;
```

```
 year |  q1   |  q2   | judge_q1_q2 | diff_q2_q1 | sign_q2_q1
------+-------+-------+-------------+------------+------------
 2015 | 82000 | 83000 | +           |       1000 |          1
 2016 | 85000 | 85000 |             |          0 |          0
 2017 | 92000 | 81000 | -           |     -11000 |         -1
```

コード例のQ1とQ2の比較と同様に、Q2とQ3、Q3とQ4の比較も可能です。ただし、前年度のQ4と翌年のQ1の比較は、同じ方法では実現できません。本項で利用したサンプルテーブルでは、1レコードに同じ年の四半期売上しか格納されていないため、年をまたがる値を比較するためには、複数レコードに渡る値を組み合わせなければなりません。その手法は、後述の「3-3 1つのテーブルに対する操作」(P.57)で紹介します。

年間の最大/最小の四半期売上を見つける

ここまで、2つのカラムの大小を比較する方法を紹介しました。カラム数が3つ以上になっても、2つのカラムの比較を組み合わせて実現できますが、カラム数が多くなると、比較するカラムの組み合わせが膨大となり、現実的ではありません。

任意数のカラムの値から最大値や最小値を見つけるには、greatest関数やleast関数を使用できます。greatest関数とleast関数はSQL標準には含まれていませんが、多くのSQLクエリエンジンに共通で実装されています。

コード3.2.2.2: 年間の最大/最小の四半期売上を見つけるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    year
    -- Q1～Q4の最大売上を取得する
  , greatest(q1, q2, q3, q4) AS greatest_sales
    -- Q1～Q4の最小売上を取得する
  , least(q1, q2, q3, q4)    AS least_sales
FROM
```

```
  quarterly_sales
ORDER BY
  year
;
```

```
 year | greatest_sales | least_sales
------+----------------+-------------
 2015 |          83000 |       78000
 2016 |          85000 |       80000
 2017 |          92000 |       81000
```

年間の平均四半期売上を計算する

greatest関数やleast関数は組み込みの関数ですが、複数カラムに対して独自の処理をするケースを考えてみます。例えば、Q1からQ4の売上の平均を計算するにはどうしたら良いでしょうか。

単純に実現するならば、下記コード例に示す通り、Q1からQ4の値を加算して4で割れば、Q1からQ4の平均値を求めることができます。

コード3.2.2.3: 単純な加算による平均四半期売上を求めるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    year
  , (q1 + q2 + q3 + q4) / 4 AS average
FROM
  quarterly_sales
ORDER BY
  year
;
```

```
 year | average
------+---------
 2015 |   81500
 2016 |   82750
 2017 |
```

次のステップ NULLの値を四則演算で扱うために、本章1節で紹介したCOALESCE関数を用いて適切な値に変換しましょう。次に示すコード例のクエリ結果の通り、2017年の平均売上を求めることができます。ただし、Q3とQ4の売上を0に置換しているため、Q1とQ2の合計売上を4で割った値となり、平均値としては大幅に低く計算されています。

コード3.2.2.4:COALESCEを用いてNULLを0に置換した場合の平均値を求めるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    year
  , (COALESCE(q1, 0) + COALESCE(q2, 0) + COALESCE(q3, 0) + COALESCE(q4, 0)) / 4
    AS average
FROM
  quarterly_sales
ORDER BY
  year
;
```

```
 year | average
------+---------
 2015 |   81500
 2016 |   82750
 2017 |   43250
```

次のステップ 2017年の平均売上を、Q1とQ2の売上のみの平均とするには、分母としてNULLではないカラムの数を計算しなければなりません。COALESCE関数とSIGN関数を組み合わせて、分母の値を適切に計算します。コード例を下記に示します。そのクエリ結果では、2017年はQ1とQ2のみの平均売上として計算できています。

コード3.2.2.5:NULLでないカラムのみを使用して平均値を求めるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    year
  , (COALESCE(q1, 0) + COALESCE(q2, 0) + COALESCE(q3, 0) + COALESCE(q4, 0))
    / (SIGN(COALESCE(q1, 0)) + SIGN(COALESCE(q2, 0))
      + SIGN(COALESCE(q3, 0)) + SIGN(COALESCE(q4, 0)))
    AS average
FROM
  quarterly_sales
ORDER BY
  year
;
```

```
 year | average
------+---------
 2015 |   81500
 2016 |   82750
 2017 |   86500
```

ワンポイント

複数の値の比較や差分は、1つのレコードに値を収めて、カラム同士の計算に持ち込むことで容易に計算できます。一方、最大値や最小値、平均値を求めるなどの処理は、複数カラムにまたがる値の場合は処理が複雑になります。こうした操作が必要なケースでは、後述の「3-3 1つのテーブルに対する操作」(P.57) で紹介する集約関数が適用できる形式にデータを整形すると処理が容易になります。

3-2-3 2つの値の比率を計算する

SQL 除算、CAST構文、CASE式、NULLIF関数

本項では、1レコードに含まれる値を組み合わせて、割合を計算する手法を紹介します。下記に示すデータ例の広告統計情報（advertising_stats）テーブルは、1日ごとの広告のインプレッション数とクリック数を集計しています。2017-04-02のad_id:001のデータは広告自体が表示されず、インプレッション数とクリック数が0となっています。このテーブルをサンプルデータとして解説を進めます。

データ3.2.3.1:広告統計情報(advertising_stats)テーブル

```
     dt     | ad_id | impressions | clicks
------------+-------+-------------+--------
 2017-04-01 | 001   |      100000 |   3000
 2017-04-01 | 002   |      120000 |   1200
 2017-04-01 | 003   |      500000 |  10000
 2017-04-02 | 001   |           0 |      0
 2017-04-02 | 002   |      130000 |   1400
 2017-04-02 | 003   |      620000 |  15000
```

整数型のデータの除算を行う

次に示すコード例では、1日分（2017-04-01）のデータで各広告のCTR（Click Through Rate）を計算します。CTRの定義は、「クリック数／インプレッション数」です。

1レコードに含まれる値の除算は、SELECT句中に「/」を使用します。ただし、PostgreSQLの場合は、advertising_statsテーブルのclicksとimpressionsカラムが整数型のため、計算結果も整数型となり値が0となります。整数型同士の除算で小数点以下も取得する場合は、事前に小数点を扱える型に変換する必要があります。CAST関数を用いてclicksをdouble precision型（倍精度浮動小数点数型）に変換すると、除算の結果もdouble precision型になります。

また、結果をパーセント表記にする場合は、ctrカラムの結果に100を掛けることでも取得できますが、ctr_as_percentカラムに示す通り、clicksに100.0を掛けることで、型変換も同時に実行できるため、クエリが簡潔になります。

コード3.2.3.1: 整数型のデータの除算を行うクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    dt
  , ad_id

    -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、
    --    整数の割り算でも結果が自動的に実数へ変換される
  , clicks / impressions AS ctr
    -- ■ PostgreSQLの場合、整数の割り算では小数点が切り捨てられてしまうので、明示的に型を変換する
    -- , CAST(clicks AS double precision) / impressions AS ctr

    -- 実数の定数を先頭で掛け算すると、暗黙的に型変換がされる
  , 100.0 * clicks / impressions AS ctr_as_percent
FROM
  advertising_stats
WHERE
  dt = '2017-04-01'
ORDER BY
  dt, ad_id
;
```

```
     dt     | ad_id | ctr  | ctr_as_percent
------------+-------+------+----------------
 2017-04-01 | 001   | 0.03 |          3.00
 2017-04-01 | 002   | 0.01 |          1.00
 2017-04-01 | 003   | 0.02 |          2.00
```

0除算を避ける

2017-04-02のデータにはimpressionsが0となるレコードが存在するため、上記のコード例に示したクエリをそのまま適用すると、0除算エラーとなってしまいます。

0除算を回避する方法の1つは、CASE式を用いてimpressionsが0かどうかに応じて場合分けする方法です。次に示すコード例のctr_as_percent_by_caseカラムでは、impressionsが0より大きい場合にはCTRを計算し、それ以外の場合はNULLを返します。

また、SQLにおけるNULLの伝搬を利用する方法でも0除算を回避できます。NULLの伝搬とは、NULLを含むデータの演算結果がすべてNULLとなる、SQLの性質です。コード例（3.2.3.2）のctr_as_percent_by_nullの中で、NULLIF(impressions, 0)の部分はimpressionsの値が0の場合にNULLとなります。その結果、impressionsの値が0の場合には、NULLの伝搬によりCTRの値もNULLとなり、CASE式を用いた方法と同じ値を得られます。

コード3.2.3.2:0除算を避けてCTRを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    dt
  , ad_id

    -- CASE式で分母が0の場合を分岐して、0除算を避ける方法
  , CASE
      WHEN impressions > 0 THEN 100.0 * clicks / impressions
    END AS ctr_as_percent_by_case

    -- 分母が0の場合はNULLに変換し、0除算を避ける方法
    -- ■ PostgreSQL, Redshift, BigQuery, SparkSQLの場合、NULLIF関数が利用できる
  , 100.0 * clicks / NULLIF(impressions, 0) AS ctr_as_percent_by_null
    -- ■ Hiveの場合、NULLIFの代わりにCASE式を用いる
    -- , 100.0 * clicks
    --   / CASE WHEN impressions = 0 THEN NULL ELSE impressions END
    --   AS ctr_as_percent_by_null
FROM
  advertising_stats
ORDER BY
  dt, ad_id
;
```

```
     dt     | ad_id | ctr_as_percent_by_case | ctr_as_percent_by_null
------------+-------+------------------------+------------------------
 2017-04-01 | 001   |                   3.00 |                   3.00
 2017-04-01 | 002   |                   1.00 |                   1.00
 2017-04-01 | 003   |                   2.00 |                   2.00
 2017-04-02 | 001   |                        |
 2017-04-02 | 002   |                   1.07 |                   1.07
 2017-04-02 | 003   |                   2.41 |                   2.41
```

ワンポイント

本項で紹介した比率も、データ分析では頻出する計算ですが、整数のまま除算を実行したり、0で割ってしまうなどの失敗は、クエリの構文自体には間違いがないため見落としてしまいがちです。注意すべき点をしっかりと押さえておきましょう。

3-2-4 2つの値の距離を計算する

SQL abs関数、power関数、sqrt関数、point型、<->演算子

本項では、2つの値を入力して、両値がどれだけ離れているか「距離」を計算する手法を紹介します。

「距離」は物理的な空間の長さを想像させますが、データ分析のさまざまな場面でも、距離の概念が登場します。例えば、テストの平均点からどれだけ点数が離れているか、前年度と今年度の売上にどの程度の差があるか、なども距離の一例です。また、あるユーザと購買傾向が近いユーザを抽出するなど、応用的な場面でも距離の概念が重要になります。

数値データの絶対値、二乗平均平方根(RMS)を計算する

データ3.2.4.1: 一次元の位置情報(location_1d)テーブル

```
 x1 | x2
----+----
  5 | 10
 10 |  5
 -2 |  4
  3 |  3
  0 |  1
```

上記に示すデータ例のlocation_1dテーブルで、数値データx1, x2に対する距離として、絶対値と二乗平均平方根を計算します。絶対値の計算は、多くの処理系でABS関数（abstract）が利用できます。二乗平均平方根は2値の差を二乗して平方根を取った値です。累乗の計算にはPOWER関数、平方根にはSQRT関数を利用できます。値が一次元データの場合、絶対値と二乗平均平方根は同じ結果です。

コード3.2.4.1 : 一次元のデータに対して絶対値と二乗平均平方根を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    abs(x1 - x2) AS abs
  , sqrt(power(x1 - x2, 2)) AS rms
FROM location_1d
;
```

```
 abs | rms
-----+-----
   5 |   5
   5 |   5
   6 |   6
   0 |   0
   1 |   1
```

xy平面上で2点間のユークリッド距離を計算する

データ3.2.4.2: 二次元の位置情報(location_2d)テーブル

```
 x1 | y1 | x2 | y2
----+----+----+----
  0 |  0 |  2 |  2
  3 |  5 |  1 |  2
  5 |  3 |  2 |  1
```

続いて、xy平面上の2点(x1, y1)と(x2, y2)間のユークリッド距離を計算します。ユークリッド距離は、物理的な空間で距離を測るときの一般的な距離の定義で、前述の二乗平均平方根と同じ計算方法になります。

また、PostgreSQLにはPOINT型と呼ばれる座標を扱うデータ構造があるため、POINT型のデータに変換することで、距離演算子<->を使用できます。計算の定義はユークリッド距離と同じです。

コード3.2.4.2：二次元のデータに対して二乗平均平方根(ユークリッド距離)を計算するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
SELECT
    sqrt(power(x1 - x2, 2) + power(y1 - y2, 2)) AS dist
    -- ■ PostgreSQLの場合はpoint型と距離演算子<->が使用できる
    -- , point(x1, y1) <-> point(x2, y2) AS dist
FROM location_2d
;
```

```
       dist
------------------
 2.82842712474619
 3.60555127546399
 3.60555127546399
```

ワンポイント

距離の計算をどのようにデータ活用に活かせばよいか、まだピンと来ないかもしれませんが、後述の「Chapter 7 データ活用の精度を高めるための分析術」(P.315) で紹介する類似度計算や「Chapter 8 データを武器にするための分析術」(P.367のレコメンド実装の基礎になる概念です。まずは計算の方法を押さえておきましょう。

3-2-5 日付・時刻を計算する

SQL interval型、日付・時刻関数

前述の「3-1-4 日付やタイムスタンプを扱う」(P.31) では、日付型やタイムスタンプ型の作成方法や要素の抽出など、単独の日付・時刻データを取り扱う方法を紹介しました。本項では、2つの日付・時刻データの差分を計算したり、1時間後の時刻を計算するなど、日付・時刻データに対する計算の手法について紹介します。

下記のデータ例に示す、サービスへの会員登録時刻とユーザーの生年月日情報を含むユーザーマスタをサンプルデータとして、会員登録日時点の年齢と現時点の年齢を計算するクエリを考えてみます。

年齢は時間の経過に応じて変化する値なので、通常は生年月日を保存しておき、購入時の年齢、会員登録しているユーザーの年齢の分布など、分析したい条件に応じて年齢を計算する必要があります。

データ3.2.5.1：登録時刻と誕生日を含むユーザーマスタ(mst_users_with_dates)テーブル

```
 user_id |   register_stamp    | birth_date
---------+---------------------+------------
 U001    | 2016-02-28 10:00:00 | 2000-02-29
 U002    | 2016-02-29 10:00:00 | 2000-02-29
 U003    | 2016-03-01 10:00:00 | 2000-02-29
```

日付・時刻データと定数の足し算・引き算を計算する

基本的な日付・時刻データの計算手法として、定数の足し算と引き算を用いて、未来や過去の日付・時刻を計算します。未来や過去の日付・時刻を計算するコード例として、会員登録時刻の1時間後と30分前の時刻、登録日の翌日と1ヶ月前の日付を計算するクエリを下記に示します。日付・時刻の計算方法は、ミドルウェアによって相応の違いがあることが分かります。

コード3.2.5.1：未来や過去の日付・時刻を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id
  -- ■ PostgreSQLの場合、interval型のデータを用いて日付/時刻の四則演算ができる
  , register_stamp::timestamp AS register_stamp
  , register_stamp::timestamp + '1 hour'::interval     AS after_1_hour
  , register_stamp::timestamp - '30 minutes'::interval AS before_30_minutes

  , register_stamp::date AS register_date
  , (register_stamp::date + '1 day'::interval  )::date AS after_1_day
  , (register_stamp::date - '1 month'::interval)::date AS before_1_month
```

```
  -- ■ Redshiftの場合、dateadd関数を用いる
  -- , register_stamp::timestamp AS register_stamp
  -- , dateadd(hour, 1, register_stamp::timestamp) AS after_1_hour
  -- , dateadd(minute, -30, register_stamp::timestamp) AS before_30_minutes
  --
  -- , register_stamp::date AS register_date
  -- , dateadd(day, 1, register_stamp::date) AS after_1_day
  -- , dateadd(month, -1, register_stamp::date) AS before_1_month

  -- ■ BigQueryの場合、timestamp_add/sub, date_add/sub等の関数を用いる
  -- , timestamp(register_stamp) AS register_stamp
  -- , timestamp_add(timestamp(register_stamp), interval 1 hour) AS after_1_hour
  -- , timestamp_sub(timestamp(register_stamp), interval 30 minute) AS before_30_minutes
  --
  -- タイムスタンプの文字列から直接日付型に変換できないため、タイムスタンプ型に直してから日付型に変換する
  -- , date(timestamp(register_stamp)) AS register_date
  -- , date_add(date(timestamp(register_stamp)), interval 1 day) AS after_1_day
  -- , date_sub(date(timestamp(register_stamp)), interval 1 month) AS before_1_month

  -- ■ Hive, SparkSQLの場合、時刻を計算する関数が実装されていないため、
  -- 一度unixtimeに変換して秒単位の計算を行い、もう一度タイムスタンプに戻す
  -- , CAST(register_stamp AS timestamp) AS register_stamp
  -- , from_unixtime(unix_timestamp(register_stamp) + 60 * 60) AS after_1_hour
  -- , from_unixtime(unix_timestamp(register_stamp) - 30 * 60) AS before_30_minutes
  --
  -- タイムスタンプの文字列から日付を得るには、to_date関数を用いる
  -- ※ ただし、Hive 2.1.0より前のバージョンでは文字列型として返される
  -- , to_date(register_stamp) AS register_date
  --
  -- 日や月の計算には、date_add関数やadd_months関数が利用できる
  -- ただし、年の計算をする関数は実装されていない
  -- , date_add(to_date(register_stamp), 1) AS after_1_day
  -- , add_months(to_date(register_stamp), -1) AS before_1_month
FROM mst_users_with_dates
;
```

```
-[ RECORD 1 ]-----+--------------------
user_id           | U001
register_stamp    | 2016-02-28 10:00:00
after_1_hour      | 2016-02-28 11:00:00
before_30_minutes | 2016-02-28 09:30:00
register_date     | 2016-02-28
after_1_day       | 2016-02-29
before_1_month    | 2016-01-28
-[ RECORD 2 ]-----+--------------------
...
```

日付データ同士の差分を計算する

2つの日付データを用いて、日付の差分を計算する方法を紹介します。下記コード例に、会員登録日と現在の日付との日数差、そして会員登録日と生年月日との日数差を計算するクエリを示します。日数差を計算する方法はミドルウェアごとに表現の違いはありますが、いずれも比較的簡単に記述可能です。

コード3.2.5.2：2つの日付の日数差を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id

  -- ■ PostgreSQL, Redshiftの場合、日付型同士の引き算ができる
  , CURRENT_DATE AS today
  , register_stamp::date AS register_date
  , CURRENT_DATE - register_stamp::date AS diff_days

  -- ■ BigQueryの場合、date_diff関数が利用できる
  -- , CURRENT_DATE AS today
  -- , date(timestamp(register_stamp)) AS register_date
  -- , date_diff(CURRENT_DATE, date(timestamp(register_stamp)), day) AS diff_days

  -- ■ Hive, SparkSQLの場合、datediff関数が利用できる
  -- , CURRENT_DATE() AS today
  -- , to_date(register_stamp) AS register_date
  -- , datediff(CURRENT_DATE(), to_date(register_stamp)) AS diff_days
FROM mst_users_with_dates
;
```

```
 user_id |   today    | register_date | diff_days
---------+------------+---------------+-----------
 U001    | 2017-02-05 | 2016-02-28    |       343
 U002    | 2017-02-05 | 2016-02-29    |       342
 U003    | 2017-02-05 | 2016-03-01    |       341
```

ユーザーの生年月日から年齢を計算する

ここまでの単純な日付の差分に比べ、年齢を計算することはそれほど容易ではありません。閏年なども考慮する必要があり、単純に日数差を365日で割って求められるものではないからです。

年齢を計算するために専用の関数が実装されているのはPostgreSQLのみです。PostgreSQLでは、日付型のデータから年齢を計算するage関数が利用できます。

age関数を用いて年齢を計算するクエリを下記のコード例に示します。返り値はinterval型で日数単

位で返されますので、EXTRACT関数で年の部分を抽出します。age関数はデフォルトで現在の年齢を返しますが、任意の日付を引数に加えることで、任意時点での年齢を計算できます。

なお、Redshiftでもage関数が動作する場合がありますが、公式にはサポートされていません。

コード3.2.5.3:age関数を用いて年齢を計算するクエリ

PostgreSQL

```
SELECT
    user_id

  -- ■ PostgreSQLの場合、age関数とEXTRACT関数を用いて年単位の年齢を計算できる
  , CURRENT_DATE AS today
  , register_stamp::date AS register_date
  , birth_date::date AS birth_date
  , EXTRACT(YEAR FROM age(birth_date::date)) AS current_age
  , EXTRACT(YEAR FROM age(register_stamp::date, birth_date::date)) AS register_age
FROM mst_users_with_dates
;
```

```
 user_id |   today    | register_date | birth_date | current_age | register_age
---------+------------+---------------+------------+-------------+--------------
 U001    | 2017-02-05 | 2016-02-28    | 2000-02-29 |          16 |           15
 U002    | 2017-02-05 | 2016-02-29    | 2000-02-29 |          16 |           16
 U003    | 2017-02-05 | 2016-03-01    | 2000-02-29 |          16 |           16
```

Redshiftのdatediff関数やBigQueryのdate_diff関数は、出力の単位にdayだけでなくyearを指定できるため、一見すると年齢の計算も可能に見えます。しかし、下記のコード例に示す通り、これらの関数でyearを指定した場合、年部分のみの差分が計算されるだけで、その年の誕生日を越えているかなどは考慮されていないため、正しく年齢を計算できません。

コード3.2.5.4:年部分の差分を計算するクエリ

Redshift BigQuery

```
SELECT
    user_id

  -- ■ Redshiftの場合、datediff関数にyearを指定しても、年部分での差分しか計算できない
  , CURRENT_DATE AS today
  , register_stamp::date AS register_date
  , birth_date::date AS birth_date
  , datediff(year, birth_date::date, CURRENT_DATE)
  , datediff(year, birth_date::date, register_stamp::date)

  -- ■ BigQueryの場合、date_diff関数にyearを指定しても、年部分での差分しか計算できない
  -- , CURRENT_DATE AS today
  -- , date(timestamp(register_stamp)) AS register_date
  -- , date(timestamp(birth_date)) AS birth_date
```

```
  -- , date_diff(CURRENT_DATE, date(timestamp(birth_date)), year) AS current_age
  -- , date_diff(date(timestamp(register_stamp)), date(timestamp(birth_date)), year)
       AS register_age
FROM mst_users_with_dates
;
```

```
 user_id |   today    | register_date | birth_date | current_age | register_age
---------+------------+---------------+------------+-------------+--------------
 U001    | 2017-02-05 | 2016-02-28    | 2000-02-29 |          17 |           16
 U002    | 2017-02-05 | 2016-02-29    | 2000-02-29 |          17 |           16
 U003    | 2017-02-05 | 2016-03-01    | 2000-02-29 |          17 |           16
```


別解 専用の関数を用いずに年齢を計算するには、日付を固定桁の整数で表現し、その差を計算する方法が簡単です。下記コード例に、日付を整数で表現して年齢を計算するクエリを示します。日付の整数表現は、例えば2016年2月28日は「20160228」、2000年2月29日は「20000229」となります。これらの日付を引き算し結果を10,000で割って整数部分のみを残すと、年齢が計算できます。

コード3.2.5.5: 日付を整数で表現して年齢を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
-- 誕生日が2000年2月29日の人の、2016年2月28日時点での年齢を計算
SELECT floor((20160228 - 20000229) / 10000) AS age;
```

```
 age
-----
  15
```

本項冒頭のデータにある文字列型の日付を整数表現に変換し、登録時点と現在時点での年齢を計算するクエリを、下記のコード例に示します。基本的には、文字列からハイフンを取り除き、整数にキャストするだけですが、ミドルウェアごとにデータ型や関数の実装に差異があるため、クエリを少しずつ修正する必要があります。

コード3.2.5.6: 文字列型の誕生日から、登録時点と現在時点での年齢を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id
  , substring(register_stamp, 1, 10) AS register_date
  , birth_date
    -- 登録時点での年齢を計算
```

```
  , floor(
      ( CAST(replace(substring(register_stamp, 1, 10), '-', '') AS integer)
        - CAST(replace(birth_date, '-', '') AS integer)
      ) / 10000
    ) AS register_age
    -- 現在時点での年齢を計算
  , floor(
      ( CAST(replace(CAST(CURRENT_DATE AS text), '-', '') AS integer)
        - CAST(replace(birth_datey, '-', '') AS integer)
      ) / 10000
    ) AS current_age

    -- ■ BigQueryの場合、textをstring, integerをint64に置き換える
    --  ( CAST(replace(CAST(CURRENT_DATE AS string), '-', '') AS int64)
    --    - CAST(replace(birth_date, '-', '') AS int64)
    --  ) / 10000

    -- ■ Hive, SparkSQLの場合、replaceをregexp_replace, textをstring,
    --   integerをintに置き換える
    --   SparkSQLの場合はさらにCURRENT_DATEをCURRENT_DATE()に置き換える
    --  ( CAST(regexp_replace(CAST(CURRENT_DATE() AS string), '-', '') AS int)
    --    - CAST(regexp_replace(birth_date, '-', '') AS int)
    --  ) / 10000
FROM mst_users_with_dates
;
```

```
 user_id | register_date | birth_date | register_age | current_age
---------+---------------+------------+--------------+-------------
 U001    | 2016-02-28    | 2000-02-29 |           15 |          16
 U002    | 2016-02-29    | 2000-02-29 |           16 |          16
 U003    | 2016-03-01    | 2000-02-29 |           16 |          16
```

まとめ

日付・時刻データの計算も、ミドルウェアで表現や実現できることに差異が多く、躓くケースが多いポイントです。実務上は、日付・時刻データを数値や文字列型などの一般的な型で取り扱う方が楽なケースもあります。

3-2-6 IPアドレスを扱う

SQL inet型、<<演算子、split_part関数、lpad関数

ログデータにユーザーのIPアドレスを保存するケースは、Web系サービスではごく一般的なことです。IPアドレスをログとして保存する場合、多くは文字列型で保存します。個々のIPアドレス単位で集計を行う場合はそれで十分ですが、IPアドレス同士を比較したり、同一ネットワーク内のIPアドレスの判定などは困難です。本項では、IPアドレス同士を比較する手法を紹介します。

IPアドレス型を活用する

PostgreSQLには、IPアドレスを取り扱うためのデータ型としてinet型が実装されており、inet型を用いることでIPアドレスの比較が容易となります。inet型同士の大小比較は、「<」や「>」を用います。

コード3.2.6.1：inet型を用いたIPアドレスの比較クエリ

PostgreSQL

```
SELECT
    CAST('127.0.0.1' AS inet) < CAST('127.0.0.2'   AS inet) AS lt
  , CAST('127.0.0.1' AS inet) > CAST('192.168.0.1' AS inet) AS gt
;
```

▼

```
 lt | gt
----+----
 t  | f
```

また、address/y形式のネットワーク範囲に、IPアドレスが含まれているかどうかも判定できます。判定には、「<<」や「>>」の演算子を用います。

コード3.2.6.2：inet型を用いてIPアドレスの範囲を扱うクエリ

PostgreSQL

```
SELECT CAST('127.0.0.1' AS inet) << CAST('127.0.0.0/8' AS inet) AS is_contained;
```

▼

```
 is_contained
--------------
 t
```

整数や文字列としてIPアドレスを扱う

inet型などIPアドレス専用のデータ型が用意されていないミドルウェアの場合、IPアドレスを比較するには、ひと工夫が必要です。

IPアドレスを整数型に変換する

1つ目の方法は、IPアドレスを整数型に変換して、数値の大小比較が可能な形式にすることです。下記コード例に示す通り、まずはテキスト型で定義されたIPアドレスの4つのオクテット部分（ドットで区切られた各値）を整数型として切り出します。

コード3.2.6.3：IPアドレスから4つのオクテット部分を切り出すクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    ip
  -- ■ PostgreSQL, Redshiftの場合、split_partで文字列を分解する
  , CAST(split_part(ip, '.', 1) AS integer) AS ip_part_1
  , CAST(split_part(ip, '.', 2) AS integer) AS ip_part_2
  , CAST(split_part(ip, '.', 3) AS integer) AS ip_part_3
  , CAST(split_part(ip, '.', 4) AS integer) AS ip_part_4

  -- ■ BigQueryの場合、split関数で配列に分解し、n番目の要素を取り出す
  --, CAST(split(ip, '.')[SAFE_ORDINAL(1)] AS int64) AS ip_part_1
  --, CAST(split(ip, '.')[SAFE_ORDINAL(2)] AS int64) AS ip_part_2
  --, CAST(split(ip, '.')[SAFE_ORDINAL(3)] AS int64) AS ip_part_3
  --, CAST(split(ip, '.')[SAFE_ORDINAL(4)] AS int64) AS ip_part_4

  -- ■ Hive, SparkSQLの場合、split関数で配列に分解し、n番目の要素を取り出す
  -- ただし、ピリオドが特殊文字のため、バックスラッシュでエスケープを行う
  --, CAST(split(ip, '\\.')[0] AS int) AS ip_part_1
  --, CAST(split(ip, '\\.')[1] AS int) AS ip_part_2
  --, CAST(split(ip, '\\.')[2] AS int) AS ip_part_3
  --, CAST(split(ip, '\\.')[3] AS int) AS ip_part_4
FROM
  (SELECT '192.168.0.1' AS ip) AS t
  -- ■ PostgreSQLの場合、明示的に型の指定が必要
  -- (SELECT CAST('192.168.0.1' AS text) AS ip) AS t
;
```

```
     ip      | ip_part_1 | ip_part_2 | ip_part_3 | ip_part_4
-------------+-----------+-----------+-----------+-----------
 192.168.0.1 |       192 |       168 |         0 |         1
```

上記コード例で取り出した4つのオクテット部分の数字に対して、第1オクテットからそれぞれ2^24, 2^16, 2^8, 2^0を掛け算し、それらを足し合わせて整数型の表記に変換します。次の出力例に示す通り、IPアドレスを整数型の値に変換することで、大小比較や範囲判定が可能になります。

コード3.2.6.4：IPアドレスを整数型の表記に変換するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    ip
  -- ■ PostgreSQL, Redshiftの場合、split_partで文字列を分解する
  ,   CAST(split_part(ip, '.', 1) AS integer) * 2^24
    + CAST(split_part(ip, '.', 2) AS integer) * 2^16
    + CAST(split_part(ip, '.', 3) AS integer) * 2^8
    + CAST(split_part(ip, '.', 4) AS integer) * 2^0
    AS ip_integer

  -- ■ BigQueryの場合、split関数で配列に分解し、n番目の要素を取り出す
  --,   CAST(split(ip, '.')[SAFE_ORDINAL(1)] AS int64) * pow(2, 24)
  --   + CAST(split(ip, '.')[SAFE_ORDINAL(2)] AS int64) * pow(2, 16)
  --   + CAST(split(ip, '.')[SAFE_ORDINAL(3)] AS int64) * pow(2,  8)
  --   + CAST(split(ip, '.')[SAFE_ORDINAL(4)] AS int64) * pow(2,  0)
  --   AS ip_integer

  -- ■ Hive, SparkSQLの場合、split関数で配列に分解し、n番目の要素を取り出す
  --   ただし、コロンが特殊文字のため、バックスラッシュでエスケープを行う
  --,   CAST(split(ip, '\\.')[0] AS int) * pow(2, 24)
  --   + CAST(split(ip, '\\.')[1] AS int) * pow(2, 16)
  --   + CAST(split(ip, '\\.')[2] AS int) * pow(2,  8)
  --   + CAST(split(ip, '\\.')[3] AS int) * pow(2,  0)
  --   AS ip_integer
FROM
  (SELECT '192.168.0.1' AS ip) AS t
  -- ■ PostgreSQLの場合、明示的に型の指定が必要
  -- (SELECT CAST('192.168.0.1' AS text) AS ip) AS t
;
```

▼

```
     ip      | ip_integer
-------------+------------
 192.168.0.1 | 3232235521
```

IPアドレスを0埋めする

IPアドレス同士を比較するもう1つの方法は、各オクテット部分を3桁の数字になるように0で埋めて、文字列として扱うことです。

コード3.2.6.5：IPアドレスを0埋めした文字列に変換するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    ip
  -- ■ PostgreSQL, Redshiftの場合、lpad関数で0埋めする
  ,   lpad(split_part(ip, '.', 1), 3, '0')
    || lpad(split_part(ip, '.', 2), 3, '0')
    || lpad(split_part(ip, '.', 3), 3, '0')
    || lpad(split_part(ip, '.', 4), 3, '0')
```

```
    AS ip_padding

  -- ■ BigQueryの場合、split関数で配列に分解し、n番目の要素を取り出す
  --, CONCAT(
  --    lpad(split(ip, '.')[SAFE_ORDINAL(1)], 3, '0')
  --  , lpad(split(ip, '.')[SAFE_ORDINAL(2)], 3, '0')
  --  , lpad(split(ip, '.')[SAFE_ORDINAL(3)], 3, '0')
  --  , lpad(split(ip, '.')[SAFE_ORDINAL(4)], 3, '0')
  --  ) AS ip_padding

  -- ■ Hive, SparkSQLの場合、split関数で配列に分解し、n番目の要素を取り出す
  --    ただし、コロンが特殊文字のため、バックスラッシュでエスケープを行う
  --, CONCAT(
  --    lpad(split(ip, '\\.')[0], 3, '0')
  --  , lpad(split(ip, '\\.')[1], 3, '0')
  --  , lpad(split(ip, '\\.')[2], 3, '0')
  --  , lpad(split(ip, '\\.')[3], 3, '0')
  --  ) AS ip_padding
FROM
  (SELECT '192.168.0.1' AS ip) AS t
  -- ■ PostgreSQLの場合、明示的に型の指定が必要
  -- (SELECT CAST('192.168.0.1' AS text) AS ip) AS t
;
```

```
     ip      |  ip_padding
-------------+--------------
 192.168.0.1 | 192168000001
```

上記コード例のlpad関数は、文字列を指定した文字数になるように左埋めする関数です。コード例では、すべてのオクテット値が3桁になるように、文字列の0で左埋めしています。最後に0埋めした第1から第4オクテットの文字列を、|| 演算子を用いて連結します。

この通り、IPアドレスの各オクテットを0埋めして、固定長の文字列に変換することで、文字列のままで正しく大小比較が可能になります。

関連項目	7-1-1 IPアドレスから国・地域を補完する 7-2-4 特定のIPからのデータを除外する

3-3

1つのテーブルに対する操作

プログラミング言語としてのSQLの特徴は、データを集合として取り扱うことです。前節までは、テーブルに含まれる1つのレコードに対して、同じ操作を一度に適用する方法を紹介しましたが、データ分析の業務では、数万から数億レコードにものぼる大量のデータを、1行単位で確認することは非現実的です。そこで、大量のデータを集約して、少ない指標でデータ全体の特徴を捉える技術が重要となります。

本節では、下記にあげるデータの集約とデータの加工をテーマに、1つの表（テーブル）を対象としたデータの集約や加工手法を紹介します。

データの集約

SQLには多くの集約関数と呼ばれる関数が準備されています。レコードの件数を数えたり、レコードに格納されている値の合計や平均、最大、最小を計算するものから、統計処理に使用するものなど、さまざまな関数があります。さらに、SQL:2003で導入されたウィンドウ関数（分析関数とも呼ばれます）を用いると、順序を考慮したデータの取り扱いをはじめ、複数レコードにまたがるデータを同時に取り扱うなど、従来のSQLでは手間が掛かる処理を簡単に実現できます。

本書で紹介するレポートやSQLには、何らかのデータを加工し、集約関数を用いてレコード数やデータの合計、平均を求めるものが数多くあります。データの集約に関して基本的な記述を習得しましょう。

データの加工

前述の「3-1 1つの値に対する操作」（P.26）と「3-2 複数の値に対する操作」（P.36）では、単一の値または複数の値に対するデータ加工の手法を紹介しました。本節では、対象テーブルの形式自体が処理したい集計に対して適していない状態の場合、テーブル形式を加工して集計可能な状態にする手順を紹介します。

本来は、本節で紹介するプロセスを踏まずとも取り扱える状態のデータが望ましいのですが、サービスを構築するために設計された業務データを処理するケースでは、稀に発生する作業です。テーブル形式が求める形式でない場合、頭を抱えてしまう場合もあるかと思いますが、分析しやすい形式に加工する手順を習得しましょう。

3-3-1 グループの特徴を捉える

SQL 集約関数、GROUP BY句、ウィンドウ関数、OVER(... PARTITION BY～)句

1つの表に対する操作として、まずは集約関数を紹介します。集約関数とは、複数レコードにまたがる値を入力として、単一の値を返す関数です。例えば、全レコードの総数を返すCOUNT関数や、値の合計値を返すSUM関数などがあります。

下記データ例に示す商品評価（review）テーブルは、商品（product_id）に対するユーザーの評価値（score）を格納しています。本項ではこのテーブルをサンプルに、全体的な特徴を掴むために利用する集約関数、SUM関数やAVG関数などを紹介します。

データ3.3.1.1：商品評価(review)テーブル

```
 user_id | product_id | score
---------+------------+-------
U001     | A001       |   4.0
U001     | A002       |   5.0
U001     | A003       |   5.0
U002     | A001       |   3.0
U002     | A002       |   3.0
U002     | A003       |   4.0
U003     | A001       |   5.0
U003     | A002       |   4.0
U003     | A003       |   4.0
```

テーブル全体の特徴量を計算する

下記コード例に、頻繁に使用する基本的な集約関数をreviewテーブルに対して適用するクエリを示します。COUNT関数は指定されたカラムの行数を返却する関数です。カラム名の前にDISTINCT句を指定すると、行の総数ではなく、ユニークな値の数を計算します。また、SUM関数は合計、AVG関数は平均値を求める関数で、カラムのデータ型が整数や実数などの数値型である必要があります。

MAX関数とMIN関数はそれぞれ最大値と最小値を求める関数で、大小比較が可能なデータ型に適用できます。つまり、数値だけでなく、文字列型やタイムスタンプ型にも適用可能です。

コード3.3.1.1：集約関数を用いてテーブル全体の特徴量を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    COUNT(*) AS total_count
  , COUNT(DISTINCT user_id) AS user_count
  , COUNT(DISTINCT product_id) AS product_count
  , SUM(score) AS sum
```

```
  , AVG(score) AS avg
  , MAX(score) AS max
  , MIN(score) AS min
FROM
  review
;
```

▼

```
 total_count | user_count | product_count | sum  | avg  | max | min
-------------+------------+---------------+------+------+-----+-----
           9 |          3 |             3 | 37.0 | 4.11 | 5.0 | 3.0
```

グルーピングしたデータの特徴量を計算する

テーブル全体ではなく、さらに細かい粒度でデータを分割したい場合は、GROUP BY句を用いて、データをグルーピングするキーを指定し、そのキーごとにデータを集約できます。下記に示すコード例では、GROUP BY句を指定して、3人のuser_idごとにデータをグルーピングし、それぞれのデータ集合ごとに集約関数を適用しています。

コード3.3.1.2：ユーザーごとにテーブルを分割して集約関数を適用するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
SELECT
    user_id
  , COUNT(*) AS total_count
  , COUNT(DISTINCT product_id) AS product_count
  , SUM(score) AS sum
  , AVG(score) AS avg
  , MAX(score) AS max
  , MIN(score) AS min
FROM
  review
GROUP BY
  user_id
;
```

▼

```
 user_id | total_count | product_count | sum  | avg  | max | min
---------+-------------+---------------+------+------+-----+-----
 U001    |           3 |             3 | 14.0 | 4.67 | 5.0 | 4.0
 U002    |           3 |             3 | 10.0 | 3.33 | 4.0 | 3.0
 U003    |           3 |             3 | 13.0 | 4.33 | 5.0 | 4.0
```

ここで1つ注意点があります。GROUP BY句を使用したクエリでは、SELECT句に指定できるカラムは、GROUP BY句に指定したカラムか集約関数だけになります。上記コード例では、SELECT句中にproduct_idやscoreを指定できません。

GROUP BY句を使用したクエリでは、GROUP BY句に指定したカラムをユニークキーとして、新たなテーブルが作られます。その過程でGROUP BY句に指定していないカラムの値は失われてしまうからです。そのため、集約関数を適用した値と集約前の値は同時に利用できません。

集約関数を適用した値と集約前の値を同時に扱う

SQL:2003以降で定義されたウィンドウ関数がサポートされている環境であれば、ウィンドウ関数を用いることで、簡潔で効率的に集約関数の結果と元の値を組み合わせることが可能です。ウィンドウ関数を用いて個別のレビュースコアとユーザーの平均レビュースコアとの差を計算するクエリを、下記コード例に示します。

コード3.3.1.3：ウィンドウ関数を用いて集約関数の結果と元の値を同時に扱うクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id
  , product_id
    -- 個別のレビュースコア
  , score
    -- 全体の平均レビュースコア
  , AVG(score) OVER() AS avg_score
    -- ユーザーの平均レビュースコア
  , AVG(score) OVER(PARTITION BY user_id) AS user_avg_score
    -- 個別のレビュースコアと、ユーザーの平均レビュースコアとの差
  , score - AVG(score) OVER(PARTITION BY user_id) AS user_avg_score_diff
FROM
  review
;
```

```
user_id | product_id | score | avg_score | user_avg_score | user_avg_score_diff
--------+------------+-------+-----------+----------------+--------------------
U001    | A001       |   4.0 |      4.11 |           4.67 |               -0.67
U001    | A002       |   5.0 |      4.11 |           4.67 |                0.33
U001    | A003       |   5.0 |      4.11 |           4.67 |                0.33
U002    | A001       |   3.0 |      4.11 |           3.33 |               -0.33
U002    | A002       |   3.0 |      4.11 |           3.33 |               -0.33
U002    | A003       |   4.0 |      4.11 |           3.33 |                0.67
U003    | A001       |   5.0 |      4.11 |           4.33 |                0.67
U003    | A002       |   4.0 |      4.11 |           4.33 |               -0.33
U003    | A003       |   4.0 |      4.11 |           4.33 |               -0.33
```

集約関数をウィンドウ関数として使用するには、集約関数の後ろにOVER句を指定します。OVER句に引数を指定しない場合は、テーブル全体に集約関数を適用した値が返ります。引数にPARTITION BY＋カラム名を指定すると、そのカラムの値ごとにグループ化して集約関数を適用します。

上記コード例では、テーブル全体でのscoreの平均値とuser_idごとのscoreの平均値が、reviewテーブルの元レコードを損なわない形式で付加されていることが分かります。さらに、集約関数の結果と元の値を組み合わせた計算として、元のscoreとuser_idごとのscoreの平均値との差分を出力しています。

3-3-2 グループの中での順序を扱う

SQL ウィンドウ関数、OVER(... ORDER BY ~)句、OVER(... ROWS ~)句

前項に引き続き、ウィンドウ関数を活用したデータ加工を紹介します。

SQLで取り扱うテーブルには、基本的に順序の概念がありません。したがって、SQLでランキングを作成したり、時系列データなど順序を考慮したデータを扱うために、かつては特別な工夫が必要でした。しかし、ウィンドウ関数の登場で、SQLで順序を扱う処理が簡潔に記述可能になっています。

本項では、下記のデータ例に示す、人気商品の商品IDとカテゴリ、スコア情報を持つテーブルをサンプルに、SQLでのランキング作成、順序を考慮した計算を実行する手法を紹介します。

データ3.3.2.1：人気商品(popular_products)テーブル

```
 product_id | category | score
------------+----------+-------
 A001       | action   |    94
 A002       | action   |    81
 A003       | action   |    78
 A004       | action   |    64
 D001       | drama    |    90
 D002       | drama    |    82
 D003       | drama    |    78
 D004       | drama    |    58
```

ORDER BY句で順序を定義する

最初にウィンドウ関数で順序を扱う基本的な構文を紹介しましょう。そもそもウィンドウ関数とは、テーブルの中で「ウィンドウ」と呼ばれるある種の範囲を定義し、その範囲内に含まれる値を特定のレコードから自由に利用できるように導入されたものです。

ただし、そのウィンドウ内から特定の値を参照するためには、その値の場所を特定する方法が必要です。そこでウィンドウ関数では、OVER句の中でORDER BY句を用いることで、ウィンドウ内のデータに順序を定義できます。

下記のコード例に、ウィンドウ関数でORDER BY句でテーブル内の順序を扱うクエリを示します。

「ORDER BY score DESC」の記述で、テーブル内の商品をスコアの高い順に順序を定義しています。ROW_NUMBER関数は、その順序に対して一意なランキングの番号を付与します。RANK関数とDENSE_RANK関数は、同順位のレコードに対して同じランキングの番号を付与します。RANK関数は同順位のレコードの後の順位は番号を飛ばし、DENSE_RANK関数の場合は順位を飛ばしません。

また、LAG関数やLEAD関数は、現在の行より前の行や後の行の値を取得する関数です。第2引数に数値を指定することで、前後n番目の値を取得することが可能です。

コード3.3.2.1：ウィンドウ関数のORDER BY句を用いて、テーブル内の順序を扱うクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    product_id
  , score

    -- スコア順に一意なランキングを付与する
  , ROW_NUMBER()         OVER(ORDER BY score DESC) AS row
    -- 同順位を許容するランキングを付与する
  , RANK()               OVER(ORDER BY score DESC) AS rank
    -- 同順位を許容し、同順位の次の順位を飛ばさないランキングを付与する
  , DENSE_RANK()         OVER(ORDER BY score DESC) AS dense_rank

    -- 現在の行より前の行の値を取得する
  , LAG(product_id)      OVER(ORDER BY score DESC) AS lag1
  , LAG(product_id, 2)   OVER(ORDER BY score DESC) AS lag2

    -- 現在の行より後の行の値を取得する
  , LEAD(product_id)     OVER(ORDER BY score DESC) AS lead1
  , LEAD(product_id, 2)  OVER(ORDER BY score DESC) AS lead2
FROM popular_products
ORDER BY row
;
```

```
 product_id | score | row | rank | dense_rank | lag1 | lag2 | lead1 | lead2
------------+-------+-----+------+------------+------+------+-------+------
 A001       |    94 |   1 |    1 |          1 |      |      | D001  | D002
 D001       |    90 |   2 |    2 |          2 | A001 |      | D002  | A002
 D002       |    82 |   3 |    3 |          3 | D001 | A001 | A002  | A003
 A002       |    81 |   4 |    4 |          4 | D002 | D001 | A003  | D003
 A003       |    78 |   5 |    5 |          5 | A002 | D002 | D003  | A004
 D003       |    78 |   6 |    5 |          5 | A003 | A002 | A004  | D004
 A004       |    64 |   7 |    7 |          6 | D003 | A003 | D004  |
 D004       |    58 |   8 |    8 |          7 | A004 | D003 |       |
```

ORDER BY句と集約関数を組み合わせる

ORDER BY句と、SUMやAVGなどの集約関数を組み合わせることで、集約関数の適用範囲を柔軟に指定できます。下記コード例に、ORDER BY句と集約関数を組み合わせた計算を行うクエリを示します。ORDER BY句に続くROWS句は、後述するウィンドウのフレーム指定の構文です。

cum_scoreカラムでは、ランキングの上位から現在の行までのスコアを合計した累計スコアを計算しています。local_avgは、現在の行とその前後1行ずつの、合計3行の平均スコアを計算しています。

また、FIRST_VALUEウィンドウ関数とLAST_VALUEウィンドウ関数は、それぞれウィンドウ内の最上位のレコードの値と最下位のレコードの値を抽出する関数です。

コード3.3.2.2：ORDER BY句と集約関数を組合せた計算を行うクエリクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    product_id
  , score

    -- スコア順に一意なランキングを付与する
  , ROW_NUMBER() OVER(ORDER BY score DESC) AS row

    -- ランキング上位からの累計スコア合計を計算する
  , SUM(score)
      OVER(ORDER BY score DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS cum_score

    -- いまの行と前後1行ずつの、合計3行の平均スコアを計算する
  , AVG(score)
      OVER(ORDER BY score DESC
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS local_avg

    -- ランキング最上位の商品IDを取得する
  , FIRST_VALUE(product_id)
      OVER(ORDER BY score DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS first_value

    -- ランキング最下位の商品IDを取得する
  , LAST_VALUE(product_id)
      OVER(ORDER BY score DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS last_value
FROM popular_products
ORDER BY row
;
```

```
 product_id | score | row | cum_score | local_avg | first_value | last_value
------------+-------+-----+-----------+-----------+-------------+-----------
 A001       |    94 |   1 |        94 |     92.00 | A001        | D004
 D001       |    90 |   2 |       184 |     88.67 | A001        | D004
 D002       |    82 |   3 |       266 |     84.33 | A001        | D004
 A002       |    81 |   4 |       347 |     80.33 | A001        | D004
 A003       |    78 |   5 |       425 |     79.00 | A001        | D004
 D003       |    78 |   6 |       503 |     73.33 | A001        | D004
 A004       |    64 |   7 |       567 |     66.67 | A001        | D004
 D004       |    58 |   8 |       625 |     61.00 | A001        | D004
```

ウィンドウのフレーム指定について

上記のコード例に登場するフレーム指定の構文を詳しく見てみましょう。フレーム指定とは、現在のレコードの位置に基づき、相対的なウィンドウを定義するための構文です。

フレーム指定の構文にはいくつか種類がありますが、最も基本的なのは「ROWS BETWEEN start AND end」です。startとendには、「CURRENT ROW」(現在の行)、「n RECEDING」(n行前)、「n FOLLOWING」(n行後)、「UNBOUNDED PRECEDING」(前の行すべて)、「UNBOUNDED FOLLOWING」(後の行すべて) などのキーワードを指定できます。

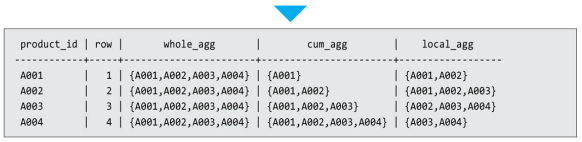
上記コード例のフレーム指定の範囲を分かりやすく確認するため、その範囲内の商品IDを集約するクエリを実行してみましょう。下記コード例に、ウィンドウのフレーム指定ごとに商品IDを集約するクエリを示します。実行結果を確認することで、フレーム指定のイメージを掴んでください。

コード3.3.2.3：ウィンドウのフレーム指定ごとに商品IDを集約するクエリ

PostgreSQL Hive SparkSQL

```
SELECT
    product_id

    -- スコア順に一意なランキングを付与する
  , ROW_NUMBER() OVER(ORDER BY score DESC) AS row

    -- ランキングの最初から最後までの範囲を対象に商品IDを集約
    -- ■ PostgreSQLの場合はarray_agg, Hive, SparkSQLの場合はcollect_listを使用する
  , array_agg(product_id)
    -- , collect_list(product_id)
      OVER(ORDER BY score DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS whole_agg

    -- ランキングの最初から現在の行までの範囲を対象に商品IDを集約
    -- ■ PostgreSQLの場合はarray_agg, Hive, SparkSQLの場合はcollect_listを使用する
  , array_agg(product_id)
    -- , collect_list(product_id)
      OVER(ORDER BY score DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS cum_agg
```

```
    -- ランキングの一つ前から一つ後までの範囲を対象に商品IDを集約
    -- ■ PostgreSQLの場合はarray_agg, Hive, SparkSQLの場合はcollect_listを使用する
  , array_agg(product_id)
    -- , collect_list(product_id)
      OVER(ORDER BY score DESC
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS local_agg
FROM popular_products
WHERE category = 'action'
ORDER BY row
;
```



```
 product_id | row |       whole_agg       |        cum_agg         |    local_agg
------------+-----+-----------------------+------------------------+-----------------
 A001       |   1 | {A001,A002,A003,A004} | {A001}                 | {A001,A002}
 A002       |   2 | {A001,A002,A003,A004} | {A001,A002}            | {A001,A002,A003}
 A003       |   3 | {A001,A002,A003,A004} | {A001,A002,A003}       | {A002,A003,A004}
 A004       |   4 | {A001,A002,A003,A004} | {A001,A002,A003,A004}  | {A003,A004}
```

なお、Redshiftには、array_agg関数やcollect_list関数と類似の関数としてlistagg関数がありますが、フレーム指定と同時に使用できないため、上記コード例の対象から外しています。

ちなみに、ウィンドウ関数にフレーム指定を行わないと、ORDER BY句がない場合はすべての行、ORDER BY句がある場合は最初の行から現在の行までが、デフォルトのフレーム指定となります。

PARTITION BY と ORDER BYを組み合わせる

前項で紹介したウィンドウ関数のPARTITION BY句とORDER BY句を組み合わせて使用することも、もちろん可能です。下記コード例に、PARTITION BY句とORDER BY句を組み合わせて、カテゴリごとの順位を計算するクエリを示します。

コード3.3.2.4：ウィンドウ関数を用いてカテゴリごとの順位を計算するクエリ

```
SELECT                                  PostgreSQL  Hive  Redshift  BigQuery  SparkSQL
    category
  , product_id
  , score

    -- カテゴリごとに、スコア順に一意なランキングを付与する
  , ROW_NUMBER()
      OVER(PARTITION BY category ORDER BY score DESC)
    AS row

    -- カテゴリごとに、同順位を許容するランキングを付与する
```

```
  , RANK()
      OVER(PARTITION BY category ORDER BY score DESC)
    AS rank

    -- カテゴリごとに、同順位を許容し、同順位の次の順位を飛ばさないランキングを付与する
  , DENSE_RANK()
      OVER(PARTITION BY category ORDER BY score DESC)
    AS dense_rank
FROM popular_products
ORDER BY category, row
;
```

```
 category | product_id | score | row | rank | dense_rank
----------+------------+-------+-----+------+------------
 action   | A001       |    94 |   1 |    1 |          1
 action   | A002       |    81 |   2 |    2 |          2
 action   | A003       |    78 |   3 |    3 |          3
 action   | A004       |    64 |   4 |    4 |          4
 drama    | D001       |    90 |   1 |    1 |          1
 drama    | D002       |    82 |   2 |    2 |          2
 drama    | D003       |    78 |   3 |    3 |          3
 drama    | D004       |    58 |   4 |    4 |          4
```

各カテゴリの上位n件を取得する

カテゴリごとの順位を計算できたことで、各カテゴリの上位n件を取得することも簡単に実現できます。ただし、SQLの仕様上、ウィンドウ関数をWHERE句に記載できないため、SELECT句でウィンドウ関数を使用した結果をサブクエリとして、その外側のWHERE句でランクを絞り込む必要があります。下記コード例に、カテゴリごとのランキング上位2件までの商品を抽出するクエリを示します。

コード3.3.2.5：カテゴリごとのランキング上位2件までの商品を抽出するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
SELECT *
FROM
  -- サブクエリ内でランキングを計算
  ( SELECT
        category
      , product_id
      , score
        -- カテゴリごとに、スコア順に一意なランキングを付与する
      , ROW_NUMBER()
          OVER(PARTITION BY category ORDER BY score DESC)
        AS rank
    FROM popular_products
  ) AS popular_products_with_rank
```

```
-- 外側のクエリでランクを絞り込む
WHERE rank <= 2
ORDER BY category, rank
;
```

```
 category | product_id | score | rank
----------+------------+-------+------
 action   | A001       |    94 |    1
 action   | A002       |    81 |    2
 drama    | D001       |    90 |    1
 drama    | D002       |    82 |    2
```

なお、カテゴリごとにスコア順の上位1件の商品IDを取得する場合には、下記コード例に示す通り、FIRST_VALUEウィンドウ関数で最上位の商品IDを取得し、SELECT DISTINCT句を用いて結果を集約する方法もあります。Hiveでは残念ながらDISTINCTの部分が動作しませんが、サブクエリを使わないため、場合によってはこちらの方法がシンプルに記述できます。

コード3.3.2.6：カテゴリごとのランキング最上位の商品を抽出するクエリ

PostgreSQL Redshift BigQuery SparkSQL

```
-- DISTINCT句を付与して重複を排除する
SELECT DISTINCT
    category
    -- カテゴリごとにランキング最上位の商品IDを取得する
  , FIRST_VALUE(product_id)
      OVER(PARTITION BY category ORDER BY score DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS product_id
FROM popular_products
;
```

```
 category | product_id
----------+------------
 drama    | D001
 action   | A001
```

3-3-3 縦持ちのデータを横持ちに変換する

SQL MAX(CASE～)構文、string_agg関数、listagg関数、collect_list関数

SQLは行（レコード）に対して一度に処理を実行することが得意なため、データを保存する際には可能な限りデータを行に分割して保持すると、操作が容易になります。しかし、最終的な出力では、データを列に展開した方が、可読性が高い形式となることも多々あります。

本項では、行単位で保持された「縦持ち」のデータを、列やカンマ区切りの文字列などの「横持ち」の形式に変換する手法を紹介します。

行を列に変換する

最初に、行を列に変換する方法を考えてみましょう。SQLにおける列はクエリによって固定的である必要があるため、列に展開されるデータの種類や数が予め判明している場合にのみ、この手法を利用できます。例えば、下記データ例に示す日次KPIデータは、日付ごとに「インプレッション数」「セッション数」「ユーザー数」の3指標の値を1レコードごとに保持したテーブルです。

データ3.3.3.1：日次KPIデータ(daily_kpi)テーブル

```
     dt     |  indicator  | val
------------+-------------+------
 2017-01-01 | impressions | 1800
 2017-01-01 | sessions    |  500
 2017-01-01 | users       |  200
 2017-01-02 | impressions | 2000
 2017-01-02 | sessions    |  700
 2017-01-02 | users       |  250
```

上述の3指標の日付ごとの推移を確認しやすくするために、それぞれの指標を列に展開しましょう。

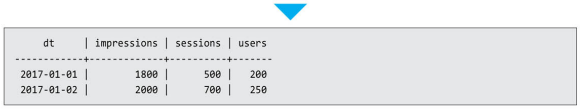
下記コード例に、行で保存された指標の値を列に変換するクエリを示します。日付ごとに1つのレコードにデータを集約するため、GROUP BY dtを指定します。さらに、「MAX（CASE～）」構文を利用して、「インプレッション数」「セッション数」「ユーザー数」に該当するレコードのみをCASE式で抽出し、MAX関数でその値を取得しています。今回のクエリでは、1つの日付でCASE式の条件がtrueになるレコードは1件しかないため、その1件をMAX関数で取り出すことになります。

コード3.3.3.1：行で保存された指標の値を列に変換するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    dt
  , MAX(CASE WHEN indicator = 'impressions' THEN val END) AS impressions
  , MAX(CASE WHEN indicator = 'sessions'    THEN val END) AS sessions
  , MAX(CASE WHEN indicator = 'users'       THEN val END) AS users
```

```
FROM daily_kpi
GROUP BY dt
ORDER BY dt
;
```



```
     dt     | impressions | sessions | users
------------+-------------+----------+-------
 2017-01-01 |        1800 |      500 |   200
 2017-01-02 |        2000 |      700 |   250
```

行をカンマ区切りの文字列に集約する

前述の通り、行を列に変換する方法は予め列の種類や数が分からなければ使用できません。例えば、下記のデータ例に示す商品の購入詳細ログを考えてみましょう。1つの購入注文で複数個の商品を購入している場合に、商品ごとのデータを1レコードずつ保持しているテーブルです。ここで、購入IDごとに1レコードにデータを集約しようとしても、1つの購入注文でいくらの商品が購入されるかは予め分からないため、単純に列に展開できません。

データ3.3.3.2：購入詳細ログ(purchase_detail_log)テーブル

```
 purchase_id | product_id | price
-------------+------------+-------
      100001 | A001       |   300
      100001 | A002       |   400
      100001 | A003       |   200
      100002 | D001       |   500
      100002 | D002       |   300
      100003 | A001       |   300
```

予め列の数を決められないケースでは、データをカンマなどのセパレータ区切りの文字列に変換する方法が考えられます。下記のコード例に、行を集約してカンマ区切りの文字列に変換するクエリを示します。なお、行を文字列に集約する関数はミドルウェアごとに実装が異なり、PostgreSQLではstring_agg関数、Redshiftではlistagg関数が使用できます。

コード3.3.3.2：行を集約してカンマ区切りの文字列に変換するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    purchase_id

    -- 商品IDを配列に集約し、カンマ区切りの文字列に変換
    -- ■ PostgreSQL, BigQueryの場合はstring_aggを用いる
  , string_agg(product_id, ',') AS product_ids
```

```
    -- ■ Redshiftの場合はlistaggを用いる
    -- , listagg(product_id, ',') AS product_ids

    -- ■ Hive, SparkSQLの場合はcollect_listとconcat_wsを用いる
    -- , concat_ws(',', collect_list(product_id)) AS product_ids
  , SUM(price) AS amount
FROM purchase_detail_log
GROUP BY purchase_id
ORDER BY purchase_id
;
```

```
 purchase_id |  product_ids   | amount
-------------+----------------+--------
      100001 | A001,A002,A003 |    900
      100002 | D001,D002      |    800
      100003 | A001           |    300
```

まとめ

本項で紹介した通り、システムで扱いやすいよう縦持ちで保存されたデータを、列や文字列などの横持ちのデータに変換することで、人間が直感的に理解しやすい形式で表示できます。

3-3-4 横持ちのデータを縦持ちに変換する

SQL unnest関数、explode関数、CROSS JOIN、LATERAL VIEW、regexp_split_to_table関数

レコードで保持された縦持ちのデータを横持ちに変換することは比較的簡単ですが、横持ちのデータを縦持ちに変換することは容易ではありません。しかし、既に保存データがカンマ区切りや列持ちの形式で、該当データを分析する必要に迫られるケースも多々あります。本項では、横持ちのデータを縦持ちに変換し、加工が容易なデータ形式にする手法を紹介します。

列で表現された値を行に変換する

横持ちのデータ例として、前述の「3-2-2 複数の値を比較する」(P.38) でも取り上げた、四半期売上テーブル（データ3.2.2.1）を再掲します。このテーブルは、年ごとにレコードを１つ保持しており、その年のQ1からQ4までの売上をカラムとして保持しています。このテーブルを、「年」「四半期ラベル」「売上」の３カラムで構成される縦持ちのテーブルに変換することを考えてみましょう。

データ3.2.2.1:四半期売上(quarterly_sales)テーブル(再掲)

```
 year |  q1   |  q2   |  q3   |  q4
------+-------+-------+-------+-------
 2015 | 82000 | 83000 | 78000 | 83000
 2016 | 85000 | 85000 | 80000 | 81000
 2017 | 92000 | 81000 |       |
```

カラムで表現された横持ちデータの特徴には、データの数が固定である点が挙げられます。上記データ例では、1レコードは必ずQ1からQ4までの4つのデータを保持しています。行に展開するべきデータ数が固定の場合は、そのデータ数と同じ数の連番を持つピボットテーブルを作成し、CROSS JOINを用いてピボットテーブルを結合します。ピボットテーブルの作成方法は、後述の「3-4-5 擬似的なテーブルを作成する」(P.88)で解説します。

下記に示すコード例では、ピボットテーブルを結合後にCASE式を用いてラベル名や売上の値を取得することで、列を行に変換しています。

コード3.3.4.1:連番を持つピボットテーブルを用いて列を行に変換するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    q.year
  -- Q1からQ4までのラベル名を表示
  , CASE
      WHEN p.idx = 1 THEN 'q1'
      WHEN p.idx = 2 THEN 'q2'
      WHEN p.idx = 3 THEN 'q3'
      WHEN p.idx = 4 THEN 'q4'
    END AS quarter
  -- Q1からQ4までの売上を表示
  , CASE
      WHEN p.idx = 1 THEN q.q1
      WHEN p.idx = 2 THEN q.q2
      WHEN p.idx = 3 THEN q.q3
      WHEN p.idx = 4 THEN q.q4
    END AS sales
FROM
    quarterly_sales AS q
  CROSS JOIN
    -- 行に展開したい列数分の連番テーブルを作成する
    (           SELECT 1 AS idx
      UNION ALL SELECT 2 AS idx
      UNION ALL SELECT 3 AS idx
      UNION ALL SELECT 4 AS idx
    ) AS p
;
```

```
 year | quarter | sales
------+---------+-------
 2015 | q1      | 82000
 2015 | q2      | 83000
 2015 | q3      | 78000
 2015 | q4      | 83000
 2016 | q1      | 85000
 2016 | q2      | 85000
 2016 | q3      | 80000
 2016 | q4      | 81000
 2017 | q1      | 92000
 2017 | q2      | 81000
 2017 | q3      |
 2017 | q4      |
```

任意長の配列を行に展開する

固定長のデータを行に展開することは比較的容易ですが、データの長さが未確定の場合は、少し複雑になります。サンプルデータとして、データ3.3.4.1に示すカンマ区切りの商品IDを含むレコードを用いて、商品IDごとに1レコードに展開する方法を考えてみましょう。

データ3.3.4.1：購入ログ(purchase_log)テーブル

```
 purchase_id |  product_ids
-------------+---------------
      100001 | A001,A002,A003
      100002 | D001,D002
      100003 | A001
```

配列をレコードに展開する際、テーブル関数（表関数）を実装しているミドルウェアでは、比較的容易に実現できます。ここでのテーブル関数とは、関数の返り値がテーブルとなる関数を意味します。

代表的なテーブル関数には、PostgreSQLやBigQueryのunnest関数、HiveやSparkSQLのexplode関数があります。これらの関数は、引数として配列を受け取り、受け取った配列をレコード分割に分割して返す関数です。

下記のコード例に、unnest関数やexplode関数の使用例を示します。

コード3.3.4.2：テーブル関数を用いて配列を行に展開するクエリ

PostgreSQL Hive BigQuery SparkSQL

```
-- ■ PostgreSQLの場合、unnest関数を用いる
SELECT unnest(ARRAY['A001','A002','A003']) AS product_id;

-- ■ BigQueryの場合もunnest関数を用いるが、テーブル関数はFROM句内でしか使えない
```

```
-- SELECT * FROM unnest(ARRAY['A001','A002','A003']) AS product_id;

-- ■ Hive, SparkSQLの場合、explode関数を用いる
-- SELECT explode(ARRAY('A001','A002','A003')) AS product_id;
```

```
 product_id
------------
 A001
 A002
 A003
```


上記のテーブル関数を用いて、前記のデータ3.3.4.1の購入ログをレコードに展開しましょう。

実際のデータにテーブル関数を用いる際には、1つ注意が必要です。通常のSELECT句内には、レコードに含める単体の値（スカラ値）を返す関数やカラム名を指定できますが、テーブル関数は集合として「テーブル」を返します。HiveやBigQueryなどではスカラ値とテーブルを同列に扱えません。上記のコード例で、BigQueryの場合はunnest関数をFROM句内に記述しているのはそのためです。

スカラ値とテーブル関数の返り値を同時に取得したい場合は、テーブル関数をFROM句内に記述し、JOIN句を用いて元のテーブルとテーブル関数の返り値を結合する必要があります。コード例を下記に示します。

なお、コード例のコメントにある通り、PostgreSQLとBigQueryの場合はCROSS JOIN、HiveとSparkSQLの場合はLATERAL VIEWを用います。

コード3.3.4.3：テーブル関数を用いてカンマ区切りのデータを行に展開するクエリ

PostgreSQL Hive BigQuery SparkSQL

```
SELECT
    purchase_id
  , product_id
FROM
    purchase_log AS p
  -- string_to_array関数で文字列を配列に変換し、unnest関数でテーブルに変換する
  CROSS JOIN unnest(string_to_array(product_ids, ',')) AS product_id

  -- ■ BigQueryの場合は、文字列の分解にsplit関数を用いる
  -- CROSS JOIN unnest(split(product_ids, ',')) AS product_id

  -- ■ HiveやSparkSQKの場合は、LATERAL VIEW explodeを用いる
  -- LATERAL VIEW explode(split(product_ids, ',')) e AS product_id;
```

```
 purchase_id | product_id
-------------+------------
      100001 | A001
      100001 | A002
      100001 | A003
      100002 | D001
      100002 | D002
      100003 | A001
```

なお、PostgreSQLの場合はSELECT句中にスカラ値とテーブル関数を同時に指定でき、なおかつ文字列をセパレータで分割してテーブル化するregexp_split_to_table関数が実装されているので、下記コード例に示す通り、簡潔な記述でカンマ区切りの文字列を行に展開できます。

コード3.3.4.4：PostgreSQLでカンマ区切りのデータを行に展開するクエリ

PostgreSQL

```
SELECT
    purchase_id
    -- カンマ区切りの文字列を一度に行に展開する
  , regexp_split_to_table(product_ids, ',') AS product_id
FROM purchase_log;
```

```
 purchase_id | product_id
-------------+------------
      100001 | A001
      100001 | A002
      100001 | A003
      100002 | D001
      100002 | D002
      100003 | A001
```

Redshiftで文字列を行に展開する

本項では、カンマ区切りの文字列を配列に変換し、データを行に展開する方法を紹介していますが、Redshiftでは配列型が公式にはサポートされてなく、紹介した方法は使用できません。

したがって、カンマ区切りの文字列を行に展開するには相応の工夫が必要となり、実現できることも限定されてしまいます。Redshiftでこの種のデータを取り扱うケースでは、データをテーブルに入れる前に、行に展開する前処理を実行するのがよいでしょう。

ここでは、Redshiftで行展開の必要に迫られたときに利用できる手法を紹介します。

ピボットテーブルを用いて文字列を行に展開する

カンマ区切りの文字列に含まれるデータの最大数をNとして、1からNまでの整数を1行ずつ持つピボットテーブルを作成します。下記コード例では、カンマ区切りの文字列に含まれる最大データ数が3なので、3行のピボットテーブルを作成します。

コード3.3.4.5：連番を持つピボットテーブルを作成するクエリ

PostgreSQL Redshift

```
SELECT *
FROM (
            SELECT 1 AS idx
  UNION ALL SELECT 2 AS idx
  UNION ALL SELECT 3 AS idx
) AS pivot
;
```

```
 idx
-----
   1
   2
   3
```

次のステップ 続いて、使用する関数がsplit_part関数です。split_part関数は、文字列をカンマなどのセパレータで分割し、そのn番目の要素を取り出す関数です。

コード3.3.4.6：split_part関数の使用例

PostgreSQL Redshift

```
SELECT
    split_part('A001,A002,A003', ',', 1) AS part_1
  , split_part('A001,A002,A003', ',', 2) AS part_2
  , split_part('A001,A002,A003', ',', 3) AS part_3
;
```

```
 part_1 | part_2 | part_3
--------+--------+--------
 A001   | A002   | A003
```

次のステップ また、カンマ区切りで表現された商品ID数をカウントするため、replace関数を用いて商品IDリストの文字列からカンマを取り除き、文字数をカウントするchar_length関数で元の文字数との差分を計算し、商品数を計算します。

コード3.3.4.7：文字数の差を用いて商品数をカウントするクエリ

PostgreSQL Redshift

```
SELECT
    purchase_id
  , product_ids
    -- 商品IDの文字列からカンマを取り除き、文字数の差を計算して商品数を取得する
  , 1 + char_length(product_ids)
      - char_length(replace(product_ids, ',', ''))
    AS product_num
FROM
  purchase_log
;
```

```
 purchase_id |  product_ids   | product_num
-------------+----------------+-------------
      100001 | A001,A002,A003 |           3
      100002 | D001,D002      |           2
      100003 | A001           |           1
```

次のステップ 前期のコード例（3.3.4.5～3.3.4.7）の出力結果を組み合わせて、purchase_logの各レコードに対して商品の数だけ連番を付与し、その連番順の商品IDを取り出します。下記コード例に最終的なクエリを示します。

コード3.3.4.8：ピボットテーブルを用いて文字列を行に展開するクエリ

PostgreSQL Redshift

```
SELECT
    l.purchase_id
  , l.product_ids
    -- 商品の数だけ連番を付与する
  , p.idx
    -- 文字列をカンマ区切りで分割し、idx番目の要素を取り出す
  , split_part(l.product_ids, ',', p.idx) AS product_id
FROM
    purchase_log AS l
  JOIN
    (           SELECT 1 AS idx
      UNION ALL SELECT 2 AS idx
      UNION ALL SELECT 3 AS idx
    ) AS p
    -- ピボットテーブルのidが、商品の数以下の場合のみ結合する
```

```
    ON p.idx <=
        (1 + char_length(l.product_ids)
           - char_length(replace(l.product_ids, ',', '')))
;
```

```
 purchase_id |  product_ids   | idx | product_id
-------------+---------------+-----+------------
      100001 | A001,A002,A003 |   1 | A001
      100001 | A001,A002,A003 |   2 | A002
      100001 | A001,A002,A003 |   3 | A003
      100002 | D001,D002      |   1 | D001
      100002 | D001,D002      |   2 | D002
      100003 | A001           |   1 | A001
```

まとめ

SQLで処理するには、事前にレコード単位でデータを分割しておくことが基本ですが、さまざまな事情により1つのレコードにデータが集約されているケースもあります。そうしたデータに直面しても決して慌てないよう、データを行に変換するテクニックを身に付けておきましょう。

3-4 複数のテーブルに対する操作

データ分析の実務では、複数のテーブルにまたがるデータを用いて分析することが多々あります。どのようなケースにどのような理由で複数テーブルに対して操作するのか簡単に説明しましょう。本節では、上述のケースに対応する方法の他にも、複数のテーブルのデータを扱う際に、複雑になりがちなSQLを可読性高く記述する方法、不足するデータをSQL側で補い複数のテーブルを効率よく利用する方法も紹介します。

業務データを利用する場合

業務データはRDBで管理されているケースが多いため、冗長なデータは正規化し、複数のテーブルでデータを管理します。したがって、分析用途でデータを使用する際にも、複数のテーブルからその全容を探る必要があります。

また、リレーショナル構造を紐付けるだけではなく、サービス側では別の行動として複数のテーブルに保存されているものを、分析時には1つのデータとして捉えたいシーンもあります。

例えば、SNSサイトのケースで、「コメント」や「いいね」、「フォロー」などが別テーブルで保存されている場合でも、分析要件として「何らかのアクションをしたユーザー」を抽出する際には、複数のテーブルを1つのテーブルとして扱いたいシチュエーションがあります。

ログデータを利用する場合

さまざまな行動を記録している1つの巨大なログファイルが、1つのテーブルに格納されているケースでも、複雑な処理を実行するために、複数のSELECT句を組み合わせたり、自己結合をして他のレコードと比較する場合は、実質的に複数テーブルを取り扱うケースと同じ操作が必要になります。

3-4-1 複数のテーブルを縦に並べる

SQL UNION ALL句

データを分析する際に、複数テーブルに分散して保存されている似通った形式のデータを一括して同じ処理を施したい場面があります。例えば、下記データ例に示す通り、異なるアプリごとにユーザーのマスタテーブルが存在しているケースを考えてみましょう。

データ3.4.1.1：アプリ1のユーザーマスタ(app1_mst_users)テーブル

```
 user_id |  name  |       email
---------+--------+--------------------
 U001    | Sato   | sato@example.com
 U002    | Suzuki | suzuki@example.com
```

データ3.4.1.2：アプリ2のユーザーマスタ(app2_mst_users)テーブル

```
 user_id |   name   |     phone
---------+----------+--------------
 U001    | Ito      | 080-xxxx-xxxx
 U002    | Tanaka   | 070-xxxx-xxxx
```

似通った構造を持つテーブルのデータを一括して処理したい場合、下記コード例に示す通り、UNION ALL句を用いて複数のテーブルを縦に結合できます。結合時には、テーブルのカラムが完全に同一である必要があるため、片方のテーブルにしか存在しないカラムは、コード例のphoneカラムでの処理に示す通りSELECT句で排除するか、emailカラムのようにデフォルト値を設定します。

また、結合後のデータが元々どのテーブルのデータか識別するため、出自を示すカラム（コード例ではapp_name）を付与すると、以降の分析で役に立ちます。

コード3.4.1.1：UNION ALL句を用いてテーブルを縦に並べるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
  SELECT 'app1' AS app_name, user_id, name, email FROM app1_mst_users
UNION ALL
  SELECT 'app2' AS app_name, user_id, name, NULL AS email FROM app2_mst_users;
```

▼

```
 app_name | user_id |  name  |       email
----------+---------+--------+--------------------
 app1     | U001    | Sato   | sato@example.com
 app1     | U002    | Suzuki | suzuki@example.com
 app2     | U001    | Ito    |
 app2     | U002    | Tanaka |
```

ワンポイント

UNION ALL句の代わりにUNION DISTINCT（または単にUNION）句を用いると、データの重複を排除した結果が得られます。ただし、UNION ALLに比べると、使用頻度は多くなく計算コストも高くなるため、まずはUNION ALL句の使い方をマスターしましょう。

3-4-2 複数のテーブルを横に並べる

SQL LEFT JOIN、相関サブクエリ

前項では複数の表を縦に並べて、一括で同じ処理を施す操作を紹介しましたが、複数の表を横に並べて、データを比較したり値を組み合わせたい場合もあります。

例えば、下記データ例3.4.2.1に示すカテゴリーマスタテーブルのデータに、カテゴリー別売上の売上額（データ3.4.2.2）やカテゴリー別商品売上ランキング（データ3.4.2.3）から、カテゴリー内で最も売れた商品IDを組み合わせて一覧表示するケースを考えてみましょう。

データ3.4.2.1：カテゴリーマスタ(mst_categories)テーブル

```
 category_id | name
-------------+------
           1 | dvd
           2 | cd
           3 | book
```

データ3.4.2.2：カテゴリー別売上(category_sales)テーブル

```
 category_id | sales
-------------+--------
           1 | 850000
           2 | 500000
```

データ3.4.2.3：カテゴリー別商品売上ランキング(product_sale_ranking)テーブル

```
 category_id | rank | product_id | sales
-------------+------+------------+-------
           1 |    1 | D001       | 50000
           1 |    2 | D002       | 20000
           1 |    3 | D003       | 10000
           2 |    1 | C001       | 30000
           2 |    2 | C002       | 20000
           2 |    3 | C003       | 10000
```

複数のテーブルを横に並べていく方法として、最も一般的なのはJOINを用いることです。ただし、マスタテーブルに対してJOINを用いると、結合できなかったレコードが抜け落ちてしまったり、逆に複数行で結合してしまい、マスタデータが重複した状態になることもあります。

次のコード例の出力結果に、上記データ例のテーブルをカテゴリーIDで単純に結合した結果を示します。カテゴリーマスタに存在していたbookカテゴリーが結合できずに欠落している、複数の商品IDが紐付くDVDやCDカテゴリーが複数行に渡るなどの出力結果となっています。

コード3.4.2.1：複数のテーブルを結合して横に並べるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    m.category_id
  , m.name
  , s.sales
  , r.product_id AS sale_product
FROM
    mst_categories AS m
  JOIN
    -- カテゴリー別の売上額を結合する
    category_sales AS s
    ON m.category_id = s.category_id
  JOIN
    -- カテゴリー別の商品を結合する
    product_sale_ranking AS r
    ON  m.category_id = r.category_id
;
```

```
 category_id | name | sales  | sale_product
-------------+------+--------+--------------
           1 | dvd  | 850000 | D001
           1 | dvd  | 850000 | D002
           1 | dvd  | 850000 | D003
           2 | cd   | 500000 | C001
           2 | cd   | 500000 | C002
           2 | cd   | 500000 | C003
```

マスタテーブルの行数を変えずにデータを横に並べるには、LEFT JOIN（右外部結合）を用いて結合できなかったレコードを残しつつ、結合するレコードが必ず1行以下になるように条件を絞り込む必要があります。

下記コード例に、マスタテーブルの行数を変えずに複数テーブルを横に並べるクエリを示します。LEFT JOINを使用することで、カテゴリー別売上テーブルなどに存在しないカテゴリ（category_id=3）のレコードを残しつつ、カテゴリー別商品売上ランキングの第1位の商品のみに絞り込むことで、カテゴリが複数行になることを回避しています。

コード3.4.2.2：マスタテーブルの行数を変えずに複数のテーブルを横に並べるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    m.category_id
  , m.name
  , s.sales
  , r.product_id AS top_sale_product
FROM
    mst_categories AS m
```

```
  -- 右外部結合を用いて結合できなかったレコードを残す
  LEFT JOIN
    -- カテゴリー別の売上額を結合する
    category_sales AS s
    ON m.category_id = s.category_id
  -- 右外部結合を用いて結合できなかったレコードを残す
  LEFT JOIN
    -- カテゴリー別のトップ売上商品を1件取得し結合する
    product_sale_ranking AS r
    ON  m.category_id = r.category_id
    AND r.rank = 1
;
```

▼

```
 category_id | name | sales  | top_sale_product
-------------+------+--------+------------------
           1 | dvd  | 850000 | D001
           2 | cd   | 500000 | C001
           3 | book |        |
```

別解 SELECT句中で相関サブクエリを使用できるミドルウェアの場合、JOINを用いずに複数のテーブルの値を横に並べていくことが可能です。

下記コード例に、相関サブクエリを用いて前述のコード例（3.4.2.1）を書き換えたクエリを示します。JOINをなくすことで、元のマスタテーブルの行数を変えてしまう心配がなくなるため、データの欠落や重複データの危険性を回避できます。

また、前述のコード例（3.4.2.1）では、カテゴリー別売上テーブルにカテゴリーごとのランク（rank）を事前にカラムとして保持する必要がありますが、相関サブクエリ内でORDER BY句やLIMIT句を用いることで、事前処理をすることなくデータを1件に絞り込むことが可能です。

コード3.4.2.3：相関サブクエリで複数のテーブルを横に並べるクエリ

PostgreSQL Redshift BigQuery

```
SELECT
    m.category_id
  , m.name
    -- 相関サブクエリでカテゴリー別の売上額を取得
  , (SELECT s.sales
     FROM category_sales AS s
     WHERE m.category_id = s.category_id
     ) AS sales
     -- 相関サブクエリでカテゴリー別のトップ売上商品を1件取得（ランクによる絞込が不要）
  , (SELECT r.product_id
     FROM product_sale_ranking AS r
     WHERE m.category_id = r.category_id
```

```
    ORDER BY sales DESC
    LIMIT 1
    ) AS top_sale_product
FROM
   mst_categories AS m
;
```

```
 category_id | name | sales  | top_sale_product
-------------+------+--------+-----------------
           1 | dvd  | 850000 | D001
           2 | cd   | 500000 | C001
           3 | book |        |
```


3-4-3 条件のフラグを0と1で表現する

SQL CASE式、SIGN関数

本項では、前項で説明した複数の表を横に並べる手法を応用して、マスタテーブルにさまざまなデータを集約し、マスタデータの属性条件を0または1のフラグで表現する手法を紹介します。

例えば、下記のデータ例に示すクレジットカード番号付きユーザーマスタテーブル（データ3.4.3.1）に、購入ログテーブル（データ3.4.3.2）を結合して、ユーザーごとに「クレジットカードの登録有無」「購入履歴の有無」の2つの条件を、それぞれ0と1で表現することを考えてみましょう。

データ3.4.3.1：クレジットカード番号付きユーザーマスタ(mst_users_with_card_number)テーブル

```
 user_id |     card_number
---------+---------------------
 U001    | 1234-xxxx-xxxx-xxxx
 U002    |
 U003    | 5678-xxxx-xxxx-xxxx
```

データ3.4.3.2：購入ログ(purchase_log)テーブル

```
 purchase_id | user_id | amount |        stamp
-------------+---------+--------+---------------------
       10001 | U001    |    200 | 2017-01-30 10:00:00
       10002 | U001    |    500 | 2017-02-10 10:00:00
       10003 | U001    |    200 | 2017-02-12 10:00:00
       10004 | U002    |    800 | 2017-03-01 10:00:00
       10005 | U002    |    400 | 2017-03-02 10:00:00
```

ユーザーマスタに購入ログを結合する際、LEFT JOINを用いたあとにユーザーIDでGROUP BYすることで、ユーザーマスタのレコード数を変えることなく、購入ログの情報を結合できます。得られたテーブルから、条件フラグを0と1の値に変換する方法は主に2つあります。1つはCASE式を用いて場合分けする方法で、もう1つはSIGN関数を用いて0以上の数字を0と1に変換する方法です。

下記のコード例に、CASE式とSIGN関数でクレジットカード登録と購入履歴の有無を0と1のフラグで表現するクエリを示します。クレジットカード番号を登録していない場合は、card_numberカラムの値がNULLになるので、CASE式を用いてNULLでない場合は1、NULLの場合は0に変換します。

また、購入回数をCOUNT関数を用いて計算すると、COUNTの結果は必ず0以上の整数となるので、SIGN関数と組み合わせることで購入履歴の有無を0と1の値に変換できます。

コード3.4.3.1：クレジットカード登録と購入履歴の有無を0, 1のフラグで表現するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    m.user_id
  , m.card_number
  , COUNT(p.user_id) AS purchase_count
    -- クレジットカード番号を登録している場合は1,登録していない場合は0で表現する
  , CASE WHEN m.card_number IS NOT NULL THEN 1 ELSE 0 END AS has_card
    -- 購入履歴がある場合は1,ない場合は0で表現する
  , SIGN(COUNT(p.user_id)) AS has_purchased
FROM
    mst_users_with_card_number AS m
  LEFT JOIN
    purchase_log AS p
    ON m.user_id = p.user_id
GROUP BY m.user_id, m.card_number
;
```

▼

```
 user_id |     card_number     | purchase_count | has_card | has_purchased
---------+---------------------+----------------+----------+---------------
 U001    | 1234-xxxx-xxxx-xxxx |              3 |        1 |             1
 U002    |                     |              2 |        0 |             1
 U003    | 5678-xxxx-xxxx-xxxx |              0 |        1 |             0
```

ワンポイント

本項で紹介した通り、特定条件を満たしているかをフラグで表現する手法を用いると、さまざまな分析を簡単に実現できます。例えば、後述の「5-1-5 ベン図でユーザーのアクションを集計する」(P.148)で紹介するベン図の元データとして利用したり、「5-2-2 継続率と定着率を算出する」(P.174)で紹介する手法で、複雑な条件を満たしている割合をAVG関数で簡単に求めるなどの応用が可能です。

3-4-4 計算したテーブルに名前を付けて再利用する

SQL CTE（WITH句）

複雑な処理をSQLで記述すると、サブクエリのネストがどんどん深くなる、似通った処理を何度も記述する必要があるなど、クエリの可読性を下げる要因になるケースが多々あります。SQL99で導入された共通テーブル式（CTE：Common Table Expression）を用いると、1つのクエリの中で使える一時的なテーブルに名前を付けて利用できるため、クエリの可読性が大きく向上します。

本項では、下記データ例に示すカテゴリー別商品売上情報をサンプルとして、カテゴリーごとのランキングを横に展開して、「DVD」「CD」「Book」カテゴリーの商品売上ランキングを横断的に閲覧できる形式に変形するクエリを考えてみましょう。

データ3.4.4.1：カテゴリー別商品売上(product_sales)テーブル

```
 category_name | product_id | sales
---------------+------------+-------
 dvd           | D001       | 50000
 dvd           | D002       | 20000
 dvd           | D003       | 10000
 cd            | C001       | 30000
 cd            | C002       | 20000
 cd            | C003       | 10000
 book          | B001       | 20000
 book          | B002       | 15000
 book          | B003       | 10000
 book          | B004       |  5000
```

カテゴリーごとのランキングをウィンドウ関数を用いて計算し、同じ順位の商品をJOINで横に並べる方法を検討してみましょう。まず、カテゴリーごとにROW_NUMBER関数でユニークな順位を付与したテーブルに対して、CTEを用いて名前を付けます。CTEの構文は、WITH句を用いて「WITH テーブル名 AS（SELECT～）」となります。下記コード例に、カテゴリーごとの順位を付与したテーブルに名前を付けるクエリを示します。

コード3.4.4.1：カテゴリーごとの順位を付与したテーブルに名前を付けるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
product_sale_ranking AS (
  SELECT
      category_name
    , product_id
    , sales
    , ROW_NUMBER() OVER(PARTITION BY category_name ORDER BY sales DESC) AS rank
```

```
  FROM
    product_sales
)
SELECT *
FROM product_sale_ranking
;
```

```
 category_name | product_id | sales | rank
---------------+------------+-------+------
 book          | B001       | 20000 |    1
 book          | B002       | 15000 |    2
 book          | B003       | 10000 |    3
 book          | B004       |  5000 |    4
 cd            | C001       | 30000 |    1
 cd            | C002       | 20000 |    2
 cd            | C003       | 10000 |    3
 dvd           | D001       | 50000 |    1
 dvd           | D002       | 20000 |    2
 dvd           | D003       | 10000 |    3
```

次のステップ カテゴリーごとの売上に順位を付与したproduct_sale_rankingテーブルができたところで、テーブルを自己結合でカテゴリーの数だけ横に並べていきます。その前に、カテゴリーごとに含まれる商品数が異なるので、最大の商品数に合わせて結果を計算するため、ランキング順位のユニークな一覧を計算します。

下記コード例に、カテゴリーごとのランキングからユニークな順位の一覧を計算するクエリを示します。WITH句を用いて複数のテーブルを定義する際は、カンマを用いてテーブルを列挙します。

コード3.4.4.2：カテゴリーごとのランキングからユニークな順位の一覧を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
product_sale_ranking AS (
  -- コード3.4.4.1参照
)
, mst_rank AS (
  SELECT DISTINCT rank
  FROM product_sale_ranking
)
SELECT *
FROM mst_rank
;
```

```
 rank
------
    1
    2
    3
    4
```

次のステップ ユニークな順位の一覧が得られたら、その一覧に対して「DVD」「CD」「Book」カテゴリーのランキングをLEFT JOINで結合していきます。下記コード例に、カテゴリーごとのランキングを横断的に表示するクエリを示します。

コード3.4.4.3：カテゴリーごとのランキングを横断的に表示するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
product_sale_ranking AS (
  -- コード3.4.4.1参照
)
, mst_rank AS (
  -- コード3.4.4.2参照
)
SELECT
    m.rank
  , r1.product_id AS dvd
  , r1.sales      AS dvd_sales
  , r2.product_id AS cd
  , r2.sales      AS cd_sales
  , r3.product_id AS book
  , r3.sales      AS book_sales
FROM
    mst_rank AS m
  LEFT JOIN
    product_sale_ranking AS r1
    ON m.rank = r1.rank
    AND r1.category_name = 'dvd'
  LEFT JOIN
    product_sale_ranking AS r2
    ON m.rank = r2.rank
    AND r2.category_name = 'cd'
  LEFT JOIN
    product_sale_ranking AS r3
    ON m.rank = r3.rank
    AND r3.category_name = 'book'
ORDER BY m.rank
;
```

```
 rank | dvd  | dvd_sales |  cd  | cd_sales | book | book_sales
------+------+-----------+------+----------+------+------------
    1 | D001 |     50000 | C001 |    30000 | B001 |      20000
    2 | D002 |     20000 | C002 |    20000 | B002 |      15000
    3 | D003 |     10000 | C003 |    10000 | B003 |      10000
    4 |      |           |      |          | B004 |       5000
```

ワンポイント

コード3.4.4.3では、CTEを用いて作成したproduct_sale_rankingテーブルを何度も結合していますが、これをCTEを用いずに記述すると、ROW_NUMBERでランキングを計算する処理が何度も登場し、非常に冗長な記述となってしまいます。可読性が高く保守のしやすいクエリにするために、CTEを積極的に活用していきましょう。また、何度も参照される一時テーブルは、可能であれば物理的なテーブルとして保存して再利用すると、パフォーマンスの面では効果的ですが、分析担当者の権限的な制約により、テーブルを作成できないこともあります。そのような制約下でも、CTEで一時的なテーブルを作成する方法が有効です。

3-4-5 擬似的なテーブルを作成する

SQL UNION ALL句、VALUES句、explode関数、generate_series関数

分析を進めていると、転送されていないテーブルが存在したり、「こんなテーブルを作成したら、こんなレポートが作成できるかも」など、ちょっとしたひらめきを試したくなるシーンがあります。しかし、ユーザーによってはテーブルの作成権限がなく、他の作業を待ったり、そのひらめきを諦めざるを得ないケースも珍しくはありません。

こんなときは、擬似的なテーブルを作成して、実際に集計を試すことができれば、作業の効率は格段に上がります。

任意のレコードを持つ擬似テーブルを作成する

前述の「3-1-1 コード値をラベルに置き換える」(P.27) では、CASE式を用いて対応する方法を紹介していますが、本項では、コード値とラベルの対応表を擬似的なテーブルとして作成し、集計する方法を紹介します。

次のコード例に、擬似的なテーブルをSELECT文を用いて実現する方法を示します。

コード3.4.5.1：デバイスのIDと名前のマスタデータを作成するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_devices AS (
            SELECT 1 AS device_id, 'PC' AS device_name
  UNION ALL SELECT 2 AS device_id, 'SP' AS device_name
  UNION ALL SELECT 3 AS device_id, 'アプリ' AS device_name
)
SELECT *
FROM mst_devices
;
```

```
 device_id | device_name
-----------+-------------
         1 | PC
         2 | SP
         3 | アプリ
```

上記の擬似的なテーブルを用いて、「3-1-1 コード値をラベルに置き換える」(P.27) で紹介したコードを書き換えたものを、下記コード例に示します。

コード3.4.5.2：擬似テーブルを用いてコードをラベルに置き換えるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_devices AS (
  -- コード3.4.5.1参照
)
SELECT
    u.user_id
  , d.device_name
FROM
    users AS u
  LEFT JOIN
    mst_devices AS d
    ON u.regist_device = d.device_id
;
```

上記のコード例は、標準SQLに準拠しており、ほとんどのミドルウェアで動作しますが、UNION ALLの処理が比較的重いため、レコード数が大きくなってくると、パフォーマンスに問題が発生する可能性があります。

VALUES句を用いた擬似テーブルの作成

各ミドルウェアで実装されている拡張機能を用いれば、上記クエリのパフォーマンスを改善可能です。PostgreSQLでは、INSERT文以外でもVALUES句を使ってレコードを作成できるので、下記に示すコード例の記述が可能です。パフォーマンス面で優れているだけでなく、記述量も大幅に削減できます。

コード3.4.5.3：VALUES句を用いて動的にテーブルを作成するクエリ

PostgreSQL

```
WITH
mst_devices(device_id, device_name) AS (
  VALUES
    (1 , 'PC' )
  , (2 , 'SP' )
  , (3 , 'アプリ')
)
SELECT *
FROM mst_devices
;
```

配列型テーブル関数を用いた擬似テーブルの作成

VALUES句を利用できないミドルウェアの場合は、構造化されたデータをテーブルに展開する関数を用いて、動的にテーブルを作成する方法があります。例えば、Hiveでは階層化されたarray型を用いてデータを定義し、explode関数を用いてテーブルに展開できます。array型とexplode関数を用いて書き換えたクエリを、下記コード例に示します。

コード3.4.5.4：配列とexplode関数を用いて動的にテーブルを作成するクエリ

Hive SparkSQL

```
WITH
mst_devices AS (
 SELECT
     -- 配列をカラムに展開する
    d[0] AS device_id
   , d[1] AS device_name
 FROM
     -- 配列をテーブルに展開する
   (SELECT explode(
     array(
       array('1' , 'PC' )
     , array('2' , 'SP' )
     , array('3' , 'アプリ')
    )) d
   ) AS t
)
SELECT *
FROM mst_devices
;
```

カラム値を配列で定義すると、すべてのデータを同じ型で定義する必要があります。前述のコード3.4.5.4ではすべて文字列型で定義しています。実用上、ほとんどのケースは問題ありませんが、異なる型を組み合わせたテーブルを作成する場合、SELECT句中で型を変換するか、下記コード例に示す通り、カラム名が冗長にはなりますが、map型を用いてカラム値を保持するとよいでしょう。

コード3.4.5.5：map型とexplode関数を用いて動的にテーブルを作成するクエリ

Hive SparkSQL

```
WITH
mst_devices AS (
 SELECT
     -- map型のデータをカラムに展開する
     d['device_id'] AS device_id
   , d['device_name'] AS device_name
 FROM
     -- 配列をテーブルに展開する
   (SELECT explode(
     array(
       map('device_id', '1' , 'device_name', 'PC' )
     , map('device_id', '2' , 'device_name', 'SP' )
     , map('device_id', '3' , 'device_name', 'アプリ')
    )) d
   ) AS t
)
SELECT *
FROM mst_devices
;
```

連番を用いてテーブルを作成する

ここまではレコードを1つずつ手動で定義して擬似テーブルを作成する方法を説明しましたが、レコード数が膨れあがると、作成やクエリ管理の手間が増大します。

ミドルウェアの一部には、連番を自動生成するテーブル関数が実装されています。このテーブル関数を用いることで、任意のレコード数を持つ疑似テーブルを手軽に作成できます。下記コード例に、連番を持つ疑似テーブルを作成するクエリを示します。generate_series関数はPostgreSQLのみがサポートしていますが、Redshiftでもリーダーノードで実行される場合は動作する可能性があります。

コード3.4.5.6：連番を持つ擬似テーブルを作成するクエリ

PostgreSQL BigQuery

```
WITH
series AS (
  -- 1から5までの連番を生成する
  -- ■ PostgreSQLの場合、generate_seriesを用いる
  SELECT generate_series(1, 5) AS idx
  -- ■ BigQueryの場合、generate_arrayを用いる
```

```
  -- SELECT idx FROM unnest(generate_array(1, 5)) AS idx
)
SELECT *
FROM series
;
```

```
 idx
-----
   1
   2
   3
   4
   5
```

HiveやSparkSQLにはgenerate_series関数が実装されていないため、上記のクエリは動作しませんが、少々トリッキーな方法で同様のテーブルを作成可能です。

下記に示すコード例は、指定文字列をn回繰り返すrepeat関数を応用して、split関数と組合せて任意長の配列を生成し、explode関数で行に展開したあと、ROW_NUMBER関数で連番を付与することで、generate_seriesと同様の結果を得られるクエリです。

コード3.4.5.7：repeat関数を応用して連番を作成するクエリ

Hive SparkSQL

```
SELECT
  ROW_NUMBER() OVER(ORDER BY x) AS idx
FROM
  -- repeat関数とsplit関数を組合せて任意長の配列を生成し、explodeで行に展開する
  (SELECT explode(split(repeat('x', 5 - 1), 'x')) AS x) AS t
;
```

```
 idx
-----
   1
   2
   3
   4
   5
```

まとめ

擬似的なテーブルを作成して、継続的な利用が予想できたり、効果的な結果が得られ、他のSQLでも使用するシーンが出てきたら、同様のSQLが点在して管理に困る前に、擬似的なテーブルではなく、テーブルとしてデータを準備するようにしましょう。

Chapter 4

売上を把握するための データ抽出

売上を把握する作業は
Webサイトや各種サービスの状況を把握する上でも
重要度が高く高頻度で発生します。
本章で紹介する手法は売上に限らず
さまざまな応用が可能です。
しっかりと習得しましょう。

4-1

時系列に沿ってデータを集約する

分析の現場では、時系列で売上金額やユーザー数、ページビューなど、各種のデータを集計することは、高い頻度で発生します。Webサイトやサービスの状況を把握する上でも、重要度が高い作業です。

例えば、日々の売上を常日頃確認しているだけでは、長期間なスパンで下降傾向にあることに気付かないケースがあります。時系列で売上金額を集計することによって、何らかの規則性を発見できたり、ある期間と比較してその変化幅を確認することも可能です。

また、本節ではデータの集約だけではなく、変化をわかりやすく表現する、レポーティング手法に関しても紹介します。紹介するレポーティング手法を利用すれば、簡単な折れ線グラフで表現された売上では読み解くことができない変化を、明確に可視化することが可能となります。

本節では、ECサイトのデータをサンプルとして、時系列で売上を評価するレポートを紹介し、そのレポートを作成するためのSQLを説明します。

サンプルデータ

本節では、2014年から2015年の2年間にわたる売上データをサンプルに説明します。下表に示すサンプルデータ「purchase_log」を用いて、さまざまな売上レポートを作成します。

データ4.1.0.1：purchase_log

```
     dt     | order_id |  user_id   | purchase_amount
------------+----------+------------+-----------------
 2014-01-01 |        1 | rhwpvvitou |           13900
 2014-01-01 |        2 | hqnwoamzic |           10616
 2014-01-02 |        3 | tzlmqryunr |           21156
 2014-01-02 |        4 | wkmqqwbyai |           14893
 2014-01-03 |        5 | ciecbedwbq |           13054
 2014-01-03 |        6 | svgnbqsagx |           24384
 2014-01-03 |        7 | dfgqftdocu |           15591
 2014-01-04 |        8 | sbgqlzkvyn |            3025
 2014-01-04 |        9 | lbedmngbol |           24215
 2014-01-04 |       10 | itlvssbsgx |            2059
 2014-01-04 |       11 | uwvexwncwp |            7324
 2014-01-04 |       12 | nvlntuwtaz |            9521
...
```

4-1-1 日別の売上を集計する

SQL GROUP BY句、SUM関数、AVG関数　　分析 合計、平均

売上を集計する業務では、横軸に日付、縦軸に金額を表現するグラフは、必要不可欠な表現です。本項では、日別に売上を集計し同時に平均購入額を集計する、下図に示すレポートを作成するSQLを紹介します。

なお、下図には表示していませんが、SQLではCOUNT関数で購入回数も同時に取得しています。

図4.1.1.1:日別の売上と平均購入額の推移

コード4.1.1.1：日別の売上と平均購入額を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    dt
  , COUNT(*) AS purchase_count
  , SUM(purchase_amount) AS total_amount
  , AVG(purchase_amount) AS avg_amount
FROM purchase_log
GROUP BY dt
ORDER BY dt
;
```

```
    dt     | purchase_count | total_amount |   avg_amount
-----------+----------------+--------------+--------------
2014-01-01 |              2 |        24516 |     12258.00
2014-01-02 |              2 |        36049 |     18024.50
2014-01-03 |              3 |        53029 |     17676.33
2014-01-04 |              3 |        29299 |      9766.33
2014-01-05 |              3 |        48256 |     16085.33
...
```

まとめ

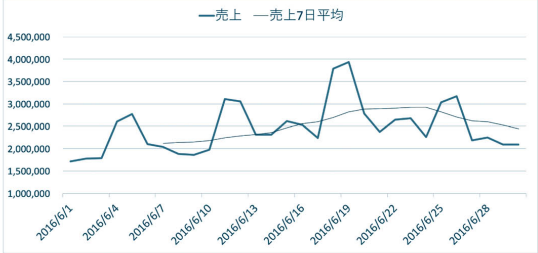
本項で題材とした売上以外にも、ソーシャルゲームにおけるポイントの消費、ECサイトであればクーポンやポイント消費の推移にも適用できます。また、資料請求や申込みなど、合計や平均する対象がない場合は、COUNT関数を使用してその推移を確認できます。

4-1-2 移動平均を用いて日別の推移を見る

SQL OVER(ORDER BY～)　　分析 移動平均

前項で説明した日別売上のレポートでは、売上が周期的に高くなる日があります。下図は土日による変動で、土日に売上が大きく動くサービスであることが分かります。しかし、このレポートでは売上が上昇傾向にあるのか、下降傾向にあるのか、判断が困難です。このようなケースでは、対象日直近7日間の売上を平均した、「7日移動平均」で表現すると分かりやすくなります。

図 4.1.2.1：日別の売上と7日移動平均グラフ



コード4.1.2.1：日別の売上と7日移動平均を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    dt
  , SUM(purchase_amount) AS total_amount

  -- 直近の最大7日間の平均を計算する
  , AVG(SUM(purchase_amount))
    OVER(ORDER BY dt ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
    AS seven_day_avg

  -- 厳密に直近の7日間の平均を計算する
  , CASE
      WHEN
        7 = COUNT(*)
        OVER(ORDER BY dt ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
      THEN
        AVG(SUM(purchase_amount))
        OVER(ORDER BY dt ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
    END
    AS seven_day_avg_strict
FROM purchase_log
GROUP BY dt
ORDER BY dt
;
```

```
    dt     | total_amount | seven_day_avg_amount | seven_day_avg_amount_strict
-----------+--------------+----------------------+-----------------------------
2014-01-01 |        24516 |             24516.00 |
2014-01-02 |        36049 |             30282.50 |
2014-01-03 |        53029 |             37864.66 |
2014-01-04 |        29299 |             35723.25 |
2014-01-05 |        48256 |             38229.80 |
2014-01-06 |        29440 |             36764.83 |
2014-01-07 |        47679 |             38324.00 |                    38324.00
2014-01-08 |        19760 |             37644.57 |                    37644.57
2014-01-09 |        22944 |             35772.42 |                    35772.42
2014-01-10 |        27923 |             32185.85 |                    32185.85
2014-01-11 |        55328 |             35904.28 |                    35904.28
2014-01-12 |        39852 |             34703.71 |                    34703.71
2014-01-13 |        47581 |             37295.28 |                    37295.28
2014-01-14 |        28686 |             34582.00 |                    34582.00
2014-01-15 |        29469 |             35969.00 |                    35969.00
2014-01-16 |        48931 |             39681.42 |                    39681.42
2014-01-17 |        42835 |             41811.71 |                    41811.71
...
```

ちなみに、前述したコード例のseven_day_avgでは、過去７日分のデータを取得できない初回の６日間は、該当日付から遡れる分のみで平均しています。厳密な７日移動平均ではない期間が存在することを回避するには、seven_day_avg_strictを使用してください。

ワンポイント

移動平均のみをレポートで表現してしまうと、日次の変動が捉えにくくなります。日別の推移と移動平均を合わせて表示しておくと良いでしょう。

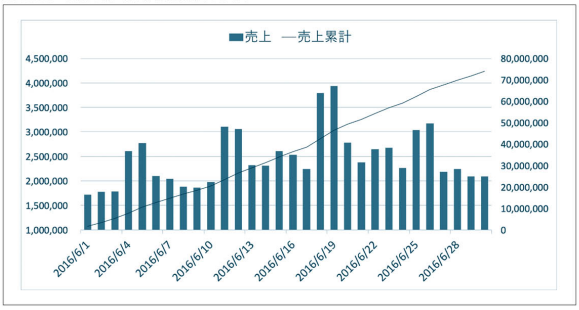
4-1-3 当月売上の累計を求める

SQL OVER(PARTITION BY～ORDER BY～)　分析 累計

月次で目標を設定している現場では、日別の売上を確認するだけではなく、その月にどれぐらいの売上が積み上がっているかを同時に確認することが重要な作業の１つです。本項では、日別の売上を集計して当月売上の累計を求めるレポートを説明します。

日別の売上だけではなく、下図に示すレポートにある通り、当月の売上累計も同時にレポートに表現するには、ウィンドウ関数を用いて計算します。

図 4.1.3.1：日別の売上と当月累計売上のグラフ



コード4.1.3.1：日別の売上と当月累計売上を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    dt
  --「年-月」を抽出する
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、substringで「年-月」部分を抽出する
  , substring(dt, 1, 7) AS year_month
  -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、substrが利用できる
  -- , substr(dt, 1, 7) AS year_month
  , SUM(purchase_amount) AS total_amount
  , SUM(SUM(purchase_amount))
      -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合は下記
      OVER(PARTITION BY substring(dt, 1, 7) ORDER BY dt ROWS UNBOUNDED PRECEDING)
      -- ■ BigQueryの場合、substringをsubstrに修正
      -- OVER(PARTITION BY substr(dt, 1, 7) ORDER BY dt ROWS UNBOUNDED PRECEDING)
    AS agg_amount
FROM
  purchase_log
GROUP BY dt
ORDER BY dt
;
```

```
     dt     | year_month | total_amount | agg_amount
------------+------------+--------------+------------
 2014-01-01 | 2014-01    |        24516 |      24516
 2014-01-02 | 2014-01    |        36049 |      60565
 2014-01-03 | 2014-01    |        53029 |     113594
 2014-01-04 | 2014-01    |        29299 |     142893
 2014-01-05 | 2014-01    |        48256 |     191149
 2014-01-06 | 2014-01    |        29440 |     220589
 2014-01-07 | 2014-01    |        47679 |     268268
 2014-01-08 | 2014-01    |        19760 |     288028
 2014-01-09 | 2014-01    |        22944 |     310972
 2014-01-10 | 2014-01    |        27923 |     338895
...
```

上記のクエリでは、日別の売上と月別の累計売上を同時に集計するため、substring関数を用いて、日付から「年・月」部分を抽出しています。

続いて、GROUP BY dtで日付ごとに集約した合計金額 SUM(purchase_amount)に対して、さらにSUMウィンドウ関数を適用し、SUM(SUM(purchase_amount)) OVER(ORDER BY dt)で日付順に累計売上を計算しています。また、毎月累計を求めるため、OVER句にPARTITION BY substring(dt, 1, 7)を加えて、月ごとにパーティションを作成しています。

別解 クエリとしてはもちろん、上記に示したコードでも間違いはありませんが、可読性に関しては改良の余地があります。繰り返し出現するSUM（purchase_amount）やSUBSTR（dt, 1, 7）をWITH句で外部化し、分かりやすい名前を付けましょう。

また、substring関数で年と月を抽出している部分は、「年」「月」「日」の3つに分割し、あとで結合する方法を採用すると、後述の分析に便利です。

コード4.1.3.2：日別の売上を一時テーブル化するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
daily_purchase AS (
  SELECT
      dt
      -- 「年」「月」「日」をそれぞれ抽出する
      -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合は下記
      -- ■ BigQueryの場合、substringをsubstrに修正
    , substring(dt, 1, 4) AS year
    , substring(dt, 6, 2) AS month
    , substring(dt, 9, 2) AS date
    , SUM(purchase_amount) AS purchase_amount
  FROM purchase_log
  GROUP BY dt
)
SELECT
  *
FROM
  daily_purchase
ORDER BY dt
;
```

```
     dt     | year | month | date | purchase_amount
------------+------+-------+------+-----------------
 2014-01-01 | 2014 | 01    | 01   |           24516
 2014-01-02 | 2014 | 01    | 02   |           36049
 2014-01-03 | 2014 | 01    | 03   |           53029
 2014-01-04 | 2014 | 01    | 04   |           29299
 2014-01-05 | 2014 | 01    | 05   |           48256
...
```

次のステップ 日付から年、月、日を別々のカラムに取り出し、日付ごとに合計金額を計算した一時テーブルを、daily_purchaseと命名します。このdaily_purchaseテーブルに、さきほどの当月売上の累計を集計するクエリを記述すると、下記に示すコードとなります。本項の冒頭で示したクエリと結果は同じですが、SELECT句中のカラムの意味が分かりやすくなっています。

コード4.1.3.3：daily_purchaseテーブルに対して当月累計売上を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
daily_purchase AS (
  -- コード4.1.3.2参照
)
SELECT
    dt
  , concat(year, '-', month) AS year_month
    -- ■ Redshiftの場合はconcat関数を組み合わせるか、||演算子を用いる
    -- , concat(concat(year, '-'), month) AS year_month
    -- , year || '-' || month AS year_month
  , purchase_amount
  , SUM(purchase_amount)
      OVER(PARTITION BY year, month ORDER BY dt ROWS UNBOUNDED PRECEDING)
    AS agg_amount
FROM daily_purchase
ORDER BY dt
;
```

```
     dt     | year_month | purchase_amount | agg_amount
------------+------------+-----------------+------------
 2014-01-01 | 2014-01    |           24516 |      24516
 2014-01-02 | 2014-01    |           36049 |      60565
 2014-01-03 | 2014-01    |           53029 |     113594
 2014-01-04 | 2014-01    |           29299 |     142893
 2014-01-05 | 2014-01    |           48256 |     191149
...
```

まとめ

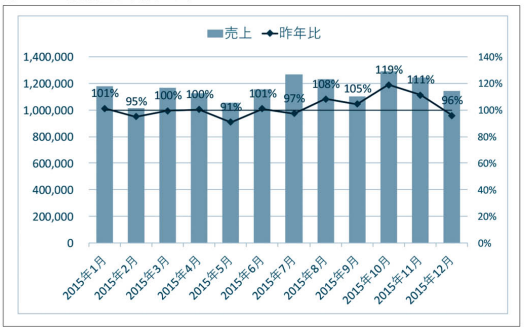
サービスを運用、開発していくために用いるSQLと比べると、ビッグデータの分析におけるSQLは、多少パフォーマンスが劣っても、可読性や再利用性を重視して記述するシーンが多々あります。SQLにもプログラムにおける「前処理」と呼ばれる考え方を取り入れてみましょう。

4-1-4 月別売上の昨対比を求める

SQL SUM(CASE～END)　　分析 昨対比

レポート業務では、日次や月次、年次売上レポートなど、さまざまな時間軸でレポートを作成します。本項では、月別の売上推移を抽出して、昨年の該当月の数字と比較します。下図に示す通り、売上がどれぐらい上昇・下降しているのか、一目で確認できるレポートを作成するSQLを紹介します。

図 4.1.4.1：月別売上と昨年比のグラフ



前項「4-1-3 当月売上の累計を求める」(P.98) の最後で作成した「daily_purchase」テーブルに対して、2015年の売上における昨対比を計算するクエリを次に示します。

2014年と2015年の月別売上を別々に計算し、月ごとに結合して昨対比を計算する方法も考えられますが、本項ではJOINを使わず昨対比を計算する手法を紹介します。

まず、対象データは2014年と2015年のデータを含めて集計し、月ごとにGROUP BYで売上額を計算します。売上額の計算時、SUM関数の中でCASE式を用いて、2014年と2015年のログをそれぞれ絞り込むことで、2014年と2015年の月別売上をそれぞれ別カラムとして出力できます。

最後に2015年の月別売上を2014年の月別売上で割り算することで、両者の比率（昨対比）を計算できます。このとき、「3-2-3 2つの値の比率を計算する」(P.42) で前述した方法を用いて、小数点以下の値まで出力しています。

コード4.1.4.1：月別売上と昨対比を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
daily_purchase AS (
  -- コード4.1.3.2参照(P.100)
)
SELECT
    month
  , SUM(CASE year WHEN '2014' THEN purchase_amount END) AS amount_2014
  , SUM(CASE year WHEN '2015' THEN purchase_amount END) AS amount_2015
  , 100.0
    * SUM(CASE year WHEN '2015' THEN purchase_amount END)
    / SUM(CASE year WHEN '2014' THEN purchase_amount END)
    AS rate
FROM
  daily_purchase
GROUP BY month
ORDER BY month
;
```

```
month | amount_2014 | amount_2015 |  rate
-------+-------------+-------------+-------
01    |     1164751 |     1178122 | 101.14
02    |     1068219 |     1015284 |  95.04
03    |     1171437 |     1165606 |  99.50
04    |     1125529 |     1128273 | 100.24
05    |     1164479 |     1056663 |  90.74
06    |     1146164 |     1156540 | 100.90
07    |     1303216 |     1268495 |  97.33
08    |     1137588 |     1230429 | 108.16
09    |     1056505 |     1104604 | 104.55
10    |     1084765 |     1289990 | 118.91
11    |     1118751 |     1245537 | 111.33
12    |     1193753 |     1143558 |  95.79
```

ワンポイント

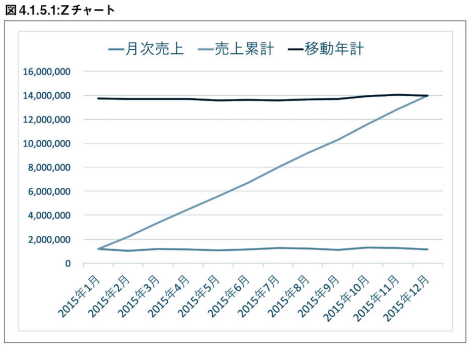
売上が伸びていた場合でも、季節トレンドによって伸びているのか、実施した施策が原因で上昇しているのか判断できません。また、季節トレンドで売上が向上していても、昨対比が下落していたら、成長が鈍化しているとも判断できます。昨対比を正しく判断しましょう。

4-1-5 Zチャートで業績の推移を確認する

SQL SUM(CASE～END) OVER(ORDER BY～)　分析 Zチャート

顧客に提供するサービスや商品、コンテンツには、季節によって売上が変動するものがあります。本項で紹介するZチャートは、「月次売上」「売上累計」「移動年計」の3つの指標で構成され、季節変動の影響を受けずにトレンドを読み解くことができる手法です。

図4.1.5.1:Zチャート



Zチャートの作成

Zチャートを作成するには、「月次売上」「売上累計」「移動年計」の3つの指標が必要になります。各指標ごとに、下記に示す範囲でそれぞれ集計します。

月次売上

売上の合計を月別に集計します。

2016年												2017年		
1月	2月	3月	4月	5月	6月	7月	8月	9月	10月	11月	12月	1月	2月	3月
合計														
	合計													
		合計												

売上累計

該当月の売上に前月までの売上累計を合計した値となります。

2016年												2017年		
1月	2月	3月	4月	5月	6月	7月	8月	9月	10月	11月	12月	1月	2月	3月
合計														
合計														
合計														

移動年計

該当月の売上に過去11ヶ月の売上累計を合計します。

2015年												2016年		
1月	2月	3月	4月	5月	6月	7月	8月	9月	10月	11月	12月	1月	2月	3月
	2016年1月の移動年計													
		2016年2月の移動年計												
			2016年3月の移動年計											

各指標の集計済みデータであるZチャートのイメージは、下表の通りです。

表4.1.5.1：Zチャート集計済みイメージ

年月	月次売上	売上累計	移動年計
2015年1月	1,178,122	1,178,122	13,748,528
2015年2月	1,015,284	2,193,406	13,695,593
2015年3月	1,165,606	3,359,012	13,689,762
2015年4月	1,128,273	4,487,285	13,692,506
2015年5月	1,056,663	5,543,948	13,584,690
2015年6月	…	…	…

Zチャートを読み解くポイント

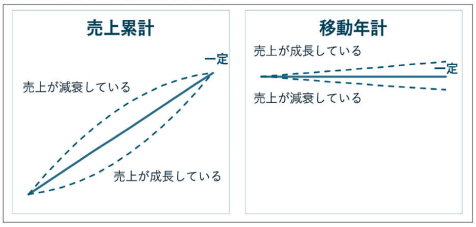
売上累計への着目

月次売上が一定の場合、売上累計は直線になります。横軸で右に行くほどグラフの傾きが急になる曲線であれば、直近の売上は上昇傾向となり、緩やかになっていれば直近の売上は減少傾向となります。グラフの表示期間で、売上がどのように推移しているのかを読み解くことが可能です（次図4.1.5.2：左）。

移動年計への着目

前年から当年にかけて売上が一定の場合は、移動年計は直線になります。右肩上がりになっている場合、売上は上昇傾向となり、右肩下がりであれば減少傾向となります。グラフに表示されていない過去1年の長いスパンで、売上がどのように推移しているのかを読み解くことができます（次図4.1.5.2：右）。

図 4.1.5.2 : Zチャートを読み解くポイント



さまざまな形状のZチャート

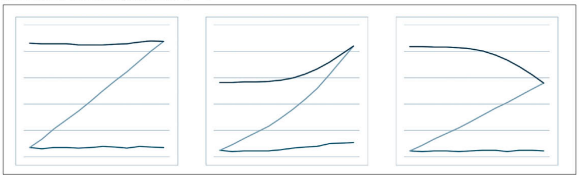
実際にいくつかのパターンで作成したZチャートを下図に示します。それぞれのグラフが何を表しているのかを解説します。

左のグラフは、売上がほぼ一定で推移している状態なので綺麗な「Z」を表しています。

続いて、中央のグラフは、期末に向かって売上が成長した状態です。月次売上と売上累計、移動年計のすべてが右肩上がりを示していることが分かります。

そして、右のグラフは、昨年は期末に向かって売上が成長したが、翌年はその成長が止まり、２年前と同じ水準になった状態を示しています。月次売上と売上累計は直線的な形状を表していますが、移動年計が右肩下がりになっています。

図 4.1.5.3 : さまざまな形状のZチャート



Zチャートを作成するための指標を集計する

Zチャートを作成するために必要な指標を集計するSQLを、下記のコード例に示します。

Zチャートに必要な指標は月単位で集計するので、まずは購入ログを月別の売上に集計します。次に、各月の売上に対して、累計売上と移動年計を計算します。移動年計を計算するためには、Zチャートを表示したい月から過去1年分の売上データが必要ですが、グラフ表示の際には不要となるので、最後に必要な期間のデータだけを抽出します。

コード4.1.5.1：2015年の売上に対するZチャートを作成するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
daily_purchase AS (
 -- コード4.1.3.2参照
)
, monthly_amount AS (
 -- 月別の売上を集計
 SELECT
     year
   , month
   , SUM(purchase_amount) AS amount
 FROM daily_purchase
 GROUP BY year, month
)
, calc_index AS (
 SELECT
     year
   , month
   , amount
     -- 2015年の累計売上を計算
   , SUM(CASE WHEN year = '2015' THEN amount END)
       OVER(ORDER BY year, month ROWS UNBOUNDED PRECEDING)
     AS agg_amount
     -- 当月から11ヶ月前までの合計売上(移動年計)を計算
   , SUM(amount)
       OVER(ORDER BY year, month ROWS BETWEEN 11 PRECEDING AND CURRENT ROW)
     AS year_avg_amount
 FROM
   monthly_purchase
 ORDER BY
   year, month
)
-- 最後に、2015年のデータのみに絞り込む
SELECT
   concat(year, '-', month) AS year_month
   -- ■ Redshiftの場合はconcat関数を組み合わせるか、||演算子を用いる
   -- concat(concat(year, '-'), month) AS year_month
   -- year || '-' || month AS year_month
```

```
 , amount
 , agg_amount
 , year_avg_amount
FROM
 calc_index
WHERE
 year = '2015'
ORDER BY
 year_month
;
```

```
 year_month | amount  | agg_amount | year_avg_amount
------------+---------+------------+----------------
 2015-01    | 1178122 |    1178122 |        13748528
 2015-02    | 1015284 |    2193406 |        13695593
 2015-03    | 1165606 |    3359012 |        13689762
 2015-04    | 1128273 |    4487285 |        13692506
 2015-05    | 1056663 |    5543948 |        13584690
 2015-06    | 1156540 |    6700488 |        13595066
 2015-07    | 1268495 |    7968983 |        13560345
 2015-08    | 1230429 |    9199412 |        13653186
 2015-09    | 1104604 |   10304016 |        13701285
 2015-10    | 1289990 |   11594006 |        13906510
 2015-11    | 1245537 |   12839543 |        14033296
 2015-12    | 1143558 |   13983101 |        13983101
```

コード例では、2015年の累計売上を計算するために、SUM関数の中でCASE式を用いて2015年の売上のみに絞り込み、SUMウィンドウ関数を用いて累計を計算しています。

また、移動年計を計算するために、同じくSUMウィンドウ関数を用いています。ここで、当月を含む過去11ヶ月の売上を合計したいので、ROWS BETWEEN 11 PRECEDING AND CURRENT ROWの指定で、現在行から11行前までのデータを合計しています。

コード例では、すべての月の売上が必ず存在する前提でこの構文を用いています。売上データが存在しない月が発生するケースでは、2015年の累計売上を計算した場合と同様に、SUM関数の中でCASE式を用いて、集計対象を絞り込むとよいでしょう。

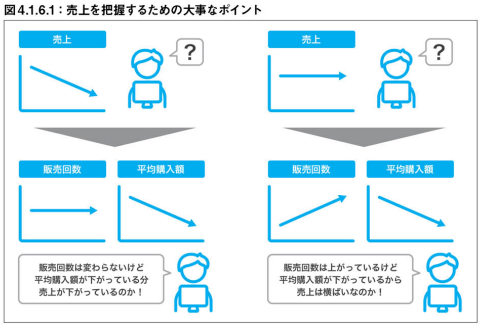
まとめ

本項で説明した通り、季節トレンドの影響を受けずに、売上の成長や衰退をさまざまな角度から、読み解くことができるのがZチャートです。常日頃からチェックする必要があるグラフではありませんが、定期的に確認しましょう。

4-1-6 売上を把握するための大事なポイント

売上だけを集計していると、売上の上昇や下降のみしか分からず、なぜ売上が上昇しているのか、もしくは下降しているのか、その理由までは分かりません。結果である売上に対して、その元となる購入回数や購入単価など、周辺データと見比べることで、売上の「なぜ」が分かるようになります。したがって、売上のレポートを求められる場合は、周辺データも含めてレポートすることが望ましいといえます。

図4.1.6.1：売上を把握するための大事なポイント



上図に示す通り、売上の「なぜ」が分かったら、関連するデータ分析を行って原因の特定や改善案を検討します。関連するデータは、販売回数であれば、訪問回数や商品数、会員登録数などを確認すれば、販売回数の上昇・下降の理由が見出せるはずです。平均購入額であれば、期間内に売れた商品の内訳を確認することで、数値に対する根拠が見出せます。

本節で紹介したSQLを利用して、下表に示すデータ例を抽出しておくと、現場の要望にも素早く対応できます。また、1つのSQLで複数の用途で使えるように心得ておくと仕事が楽になるはずです。

表4.1.6.1：集計済みデータイメージ

販売月	販売回数	平均購入額	売上額	累計売上額	昨年売上額	昨年比
2015年1月	75	15,708	1,178,122	1,178,122	1,164,751	101.14%
2015年2月	68	14,930	1,015,284	2,193,406	1,068,219	95.04%
2015年3月	75	15,541	1,165,606	3,359,012	1,171,437	99.50%

コード4.1.6.1：売上に関する指標を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
daily_purchase AS (
 -- コード4.1.3.2参照
)
, monthly_purchase AS (
 SELECT
     year
   , month
   , SUM(orders) AS orders
   , AVG(purchase_amount) AS avg_amount
   , SUM(purchase_amount) AS monthly
 FROM daily_purchase
 GROUP BY year, month
)
SELECT
   concat(year, '-', month) AS year_month
   -- ■ Redshiftの場合はconcat関数を組み合わせるか、||演算子を用いる
   -- concat(concat(year, '-'), month) AS year_month
   -- year || '-' || month AS year_month
 , orders
 , avg_amount
 , monthly
 , SUM(monthly)
     OVER(PARTITION BY year ORDER BY month ROWS UNBOUNDED PRECEDING)
   AS agg_amount
   -- 12ヶ月前の売上を求める
 , LAG(monthly, 12)
     OVER(ORDER BY year, month)
     -- ■ SparkSQLの場合は下記を用いる
     -- OVER(ORDER BY year, month ROWS BETWEEN 12 PRECEDING AND 12 PRECEDING)
  AS last_year
   -- 12ヶ月前の売上に対する割合を求める
 , 100.0
   * monthly
   / LAG(monthly, 12)
       OVER(ORDER BY year, month)
       -- ■ SparkSQLの場合は下記を用いる
       -- OVER(ORDER BY year, month ROWS BETWEEN 12 PRECEDING AND 12 PRECEDING)
   AS rate
FROM
 monthly_purchase
ORDER BY
 year_month
;
```

```
 year_month | orders | avg_amount | monthly | agg_amount | last_year | rate
------------+--------+------------+---------+------------+-----------+-------
 2014-01    |     77 | 15126.6363 | 1164751 |    1164751 |           |
 2014-02    |     69 | 15481.4347 | 1068219 |    2232970 |           |
 2014-03    |     78 | 15018.4230 | 1171437 |    3404407 |           |
 2014-04    |     75 | 15007.0533 | 1125529 |    4529936 |           |
 2014-05    |     74 | 15736.2027 | 1164479 |    5694415 |           |
 2014-06    |     74 | 15488.7027 | 1146164 |    6840579 |           |
 2014-07    |     79 | 16496.4050 | 1303216 |    8143795 |           |
 2014-08    |     76 | 14968.2631 | 1137588 |    9281383 |           |
 2014-09    |     74 | 14277.0945 | 1056505 |   10337888 |           |
 2014-10    |     76 | 14273.2236 | 1084765 |   11422653 |           |
 2014-11    |     74 | 15118.2567 | 1118751 |   12541404 |           |
 2014-12    |     78 | 15304.5256 | 1193753 |   13735157 |           |
 2015-01    |     75 | 15708.2933 | 1178122 |    1178122 |   1164751 | 101.14
 2015-02    |     68 | 14930.6470 | 1015284 |    2193406 |   1068219 |  95.04
 2015-03    |     75 | 15541.4133 | 1165606 |    3359012 |   1171437 |  99.50
 2015-04    |     72 | 15670.4583 | 1128273 |    4487285 |   1125529 | 100.24
 2015-05    |     74 | 14279.2297 | 1056663 |    5543948 |   1164479 |  90.74
 2015-06    |     72 | 16063.0555 | 1156540 |    6700488 |   1146164 | 100.90
 2015-07    |     79 | 16056.8987 | 1268495 |    7968983 |   1303216 |  97.33
 2015-08    |     81 | 15190.4814 | 1230429 |    9199412 |   1137588 | 108.16
 2015-09    |     72 | 15341.7222 | 1104604 |   10304016 |   1056505 | 104.55
 2015-10    |     80 | 16124.8750 | 1289990 |   11594006 |   1084765 | 118.91
 2015-11    |     78 | 15968.4230 | 1245537 |   12839543 |   1118751 | 111.33
 2015-12    |     75 | 15247.4400 | 1143558 |   13983101 |   1193753 |  95.79
```

上記に示すコード例と出力結果は、本項冒頭のpurchase_logテーブルに対して、月単位の売上をまとめたmonthly_purchaseテーブルを作成し、そのテーブルにさらにウィンドウ関数を適用しています。

前年の同じ月の売上を求める際に、「4-1-4 月別売上の昨対比を求める」（P.102）ではCASE式の中で対象年を絞り込みましたが、本項ではLAG関数を用いて、12ヶ月前の売上を取得する方法で、より一般化した記述です。LAGで12ヶ月前の値が取得できないレコードに関しては、last_year（昨年売上）やrate（昨対比）カラムはNULLとなります。

このコード例をウィンドウ関数を利用せずに記述すると、各指標ごとにSELECT文を作成し、最後にそれらの結果を1つのテーブルに結合することになります。しかし、複数のSELECT文を記述することは、そのたびにデータの読み込みが発生するため、パフォーマンス的に好ましくありません。

さらに、BigQueryなど読み込みデータ量に課金されるサービスでは、金銭的コストも負担になってしまいます。大量のデータをSQLで扱う場合は、上記クエリで例示するような工夫を施し、無駄なデータ読み込みを可能な限り減らすクエリを心掛けましょう。

なお、今回の集計内容であれば、中間テーブルのmonthly_purchaseテーブルをWITH句で作成せず、1つのSELECT文で最終結果を計算することも可能ですが、コード例に示す通り、計算過程のカラムも一時テーブルを使用して命名しておくと、可読性が向上します。

4-2
多面的な軸を使って データを集約する

時系列で売上金額の推移を表現する以外にも、商品のカテゴリに着目して、売上の内訳を集計したり、構成する割合を集計するなど、レポートの表現方法にはさまざまなものがあります。

前項「4-1 時系列的に沿ってデータを集約する」(P.94) で紹介した内容は、折れ線グラフで売上の推移を示すもので、売上がどのような構成で成り立っているのかは把握できません。

本節では売上を時系列だけではなく、商品のカテゴリや価格、その他の情報を組み合わせて、データの特徴を捉える、レポーティング手法とそのSQLを紹介します。

サンプルデータ

本節では、ECサイトをイメージした下記の「purchase_detail_log」をサンプルデータとして解説します。

このECサイトは一度の注文で複数商品の購入が可能なものとします。そのため、1つのorder_idに対して、複数のitem_idと価格、カテゴリ情報が紐付いています。

データ4.2.0.1：purchase_detail_log

dt	order_id	user_id	item_id	price	category	sub_category
2017-01-08	48291	usr33395	lad533	3730	ladys_fashion	bag
2017-01-08	48291	usr33395	lad329	9730	ladys_fashion	jacket
2017-01-08	48291	usr33395	lad102	11460	ladys_fashion	jacket
2017-01-08	48291	usr33395	lad886	3330	ladys_fashion	bag
2017-01-08	48292	usr52832	dvd871	3280	dvd	documentary
2017-01-08	48292	usr52832	gam167	2600	game	accessories
2017-01-08	48292	usr52832	lad289	5730	ladys_fashion	bag
2017-01-08	48293	usr28891	out977	2860	outdoor	camp
2017-01-08	48293	usr28891	boo256	2250	book	business
2017-01-08	48293	usr28891	lad125	6150	ladys_fashion	jacket
2017-01-08	48294	usr33604	men233	11630	mens_fashion	jacket
2017-01-08	48294	usr33604	cd477	2580	cd	classic
2017-01-08	48294	usr33604	boo468	3100	book	business
2017-01-08	48294	usr33604	foo402	4870	food	meats
2017-01-08	48295	usr38013	foo134	3200	food	fish
2017-01-08	48295	usr38013	lad147	9610	ladys_fashion	jacket
...

4-2-1 カテゴリ別の売上と小計を計算する

SQL UNION ALL句、ROLLUP句　　分析 小計

レポート業務では、最初に全体的な数値の概況を伝えた上で、その内訳をさまざまな切り口でレポートするケースが多々あります。例えば、売上合計額を提示したら、PCサイトとSPサイト各々のレポートに続いて、商品のカテゴリ別ではどのように構成されているか、Webサイトの総ページビュー数に対して、会員によるページビュー数と非会員のページビュー数の比率はどうなっているかなど、トータルの値をレポートしたら、必ずさらなるドリルダウンを求められます。こうした要求に対して、即座に対応できるようにSQLを作成してみましょう。

今回のSQLで取得する形式を下表に示します。1行目の大カテゴリ、小カテゴリがallのレコードが全体の売上の合計（総計）となり、2行目の小カテゴリallのレコードは大カテゴリmensの売上の合計（小計）となります。

表4.2.1.1：カテゴリ別の売上と小計を同時に取得する形式

大カテゴリ	小カテゴリ	売上
all	all	76,000
mens	all	30,400
mens	bag	17,600
mens	shoes	12,800
womens	all	45,600
womens	bag	14,700
womens	shoes	30,900

階層的なカテゴリでそれぞれの小計と総計を1つのレポートに表示するには、階層ごとに集計した結果を同じカラムになるように整形し、UNION ALL句で1テーブルにまとめる方法があります。下記のコード例に、UNION ALL句を用いて小計や総計をまとめるクエリを示します。

コード4.2.1.1：カテゴリ別の売上と小計を同時に取得するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
sub_category_amount AS (
  -- 小カテゴリの売上を集計する
  SELECT
      category     AS category
    , sub_category AS sub_category
    , SUM(price) AS amount
  FROM
    purchase_detail_log
  GROUP BY
    category, sub_category
)
```

```
, category_amount AS (
  -- 大カテゴリの売上を集計する
  SELECT
      category
    , 'all' AS sub_category
    , SUM(price) AS amount
  FROM
    purchase_detail_log
  GROUP BY
    category
)
, total_amount AS (
  -- 全体の売上を集計する
  SELECT
      'all' AS category
    , 'all' AS sub_category
    , SUM(price) AS amount
  FROM
    purchase_detail_log
)
          SELECT category, sub_category, amount FROM sub_category_amount
UNION ALL SELECT category, sub_category, amount FROM category_amount
UNION ALL SELECT category, sub_category, amount FROM total_amount
;
```

```
    category    | sub_category | amount
----------------+--------------+--------
 ladys_fashion | bag          |   17600
 ladys_fashion | jacket       |   12800
 mens_fashion  | jacket       |   14700
 mens_fashion  | shoes        |   30900
 ladys_fashion | all          |   30400
 mens_fashion  | all          |   45600
 all           | all          |   76000
```

上記の結果に示す通り、1つのクエリ結果として、カテゴリ別の小計や総計を同時に出力することが可能です。

別解 本項で紹介したUNION ALLを用いてテーブルを連結する方法は、データを複数回ロードしたり、最終的にデータを結合するコストが発生するため、残念ながらパフォーマンスは良好ではありません。しかし、SQL99で導入された「ROLLUP」を実装しているPostgreSQLやHive, SparkSQLでは、同様の結果を簡潔なクエリで取得できます。

下記にROLLUPを用いてカテゴリ別の売上と小計を同時に取得するクエリを示します。ROLLUPを用いると、小計に相当するレコードの集計キーがNULLとなるので、COALESCE関数を用いて文字列“all”に変換します。このように抽出することで、全体とカテゴリ別の結果を効率的に取得できます。

コード4.2.1.2：ROLLUPを用いてカテゴリ別の売上と小計を同時に取得するクエリ

PostgreSQL Hive SparkSQL

```
SELECT
    COALESCE(category, 'all')     AS category
  , COALESCE(sub_category, 'all') AS sub_category
  , SUM(price) AS amount
FROM
  purchase_detail_log
GROUP BY
  ROLLUP(category, sub_category)
  -- Hiveの場合は下記を使用
  -- category, sub_category WITH ROLLUP
;
```

```
    category    | sub_category | amount
----------------+--------------+--------
 ladys_fashion | bag           |   17600
 ladys_fashion | jacket        |   12800
 ladys_fashion | all           |   30400
 mens_fashion  | jacket        |   14700
 mens_fashion  | shoes         |   30900
 mens_fashion  | all           |   45600
 all           | all           |   76000
```

ワンポイント

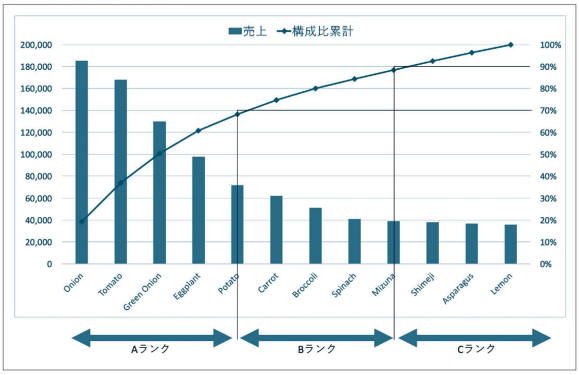
レポートを作成するツールには、小計を計算してくれるものもあるので、最小粒度で集計すれば後からでも小計の計算は可能です。しかし、売上ではなくユニークユーザーを扱うケースでは、最小粒度のユニークユーザーを合計したものが全体のユニークユーザーにはならないので、後から計算で算出することは不可能です。こうしたシーンでは重宝するSQLです。

4-2-2 ABC分析で売れ筋を判別する

SQL SUM(～) OVER(ORDER BY～)　　分析 構成比累計、ABC分析

ABC分析は在庫管理などで用いられる分析手法で、売上における重要度によって商品をレベル分けして、それに応じた戦略を考える際に利用されます。例えば、次図に示す通り、担当するサービスの売上がどのように構成されているかを把握する際にも効果的です。

図4.2.2.1：ABC分析イメージ図



上図を例にすると、売上の70%は5商品で構成されていることが一目瞭然です。一般的に上位70%を「ランクA」と呼びます。分析目的に応じて変わりますが、下記のランク分けが一般的です。

- ランクA：上位0%～70%まで
- ランクB：上位70%～90%まで
- ランクC：上位90%～100%まで

ABC分析に利用するデータ例を下表に示します。

表4.2.2.1：ABC分析におけるデータの形式

カテゴリ	売上	構成比	構成比累計	ランク
ladys_fashion	185,000	19%	19%	A
mens_fashion	168,000	18%	37%	A
book	130,000	14%	50%	A
game	98,000	10%	61%	A
dvd	72,000	8%	68%	A
food	62,000	6%	75%	B
…	…	…	…	…
supplement	36,000	4%	100%	C
	957,000	100%		

データの作成手順は下記の通りです。

1. 売上の高い順にデータを並べます。
2. 売上の合計額を集計します。
3. 売上の合計額から、各データの売上に占める割合を計算し、構成比を求めます。
4. 計算するカテゴリの構成比とそれまでの構成比累計を足して、構成比累計を計算します（カテゴリの売上とそれまでの累計を別途計算し、総売上で割ることで構成比累計を計算することも可能です）。

ランクをSQLとして記述すると、ランク分けの変更に伴いSQLを修正して再実行する必要があるため、出力結果を表示するレポート側で判定することが望ましく、必ずしもSQLとして記述する必要はありません。出力結果を表示するツールやシステムの構成、レポートを受け取る側にABC分析の知識の有無など、状況に応じてSQLへの記述有無を検討しましょう。

下記のコード例に、2015年12月の1ヶ月間の購入ログに対して、売上構成比累計とABCランクを計算するクエリを示します。まず、購入ログを対象1ヶ月間のログに絞り込み、商品カテゴリごとの項目別売上を計算します。続いて、全体の売上に対する項目別売上の構成比と構成比累計を計算します。最後に、構成比累計を上位「0%～70%」「70%～90%」「90%～100%」のランクで分類します。

コード4.2.2.1：売上構成比累計とABCランクを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
monthly_sales AS (
  SELECT
      category1
      -- 項目別売上を計算
    , SUM(amount) AS amount
  FROM
    purchase_log
  -- 対象1ヶ月間のログに絞り込む
  WHERE
    dt BETWEEN '2015-12-01' AND '2015-12-31'
  GROUP BY
    category1
)
, sales_composition_ratio AS (
  SELECT
      category1
    , amount

    -- 構成比: 100.0 * 項目別売上 / 全体売上
    , 100.0 * amount / SUM(amount) OVER() AS composition_ratio

    -- 構成比累計: 100.0 * 項目別累計売上 / 全体売上
    , 100.0 * SUM(amount) OVER(ORDER BY amount DESC)
```

```
      / SUM(amount) OVER() AS cumulative_ratio
  FROM
    monthly_sales
)
SELECT
    *
  -- 構成比累計の範囲に応じてランク付け
  , CASE
      WHEN cumulative_ratio BETWEEN  0 AND  70 THEN 'A'
      WHEN cumulative_ratio BETWEEN 70 AND  90 THEN 'B'
      WHEN cumulative_ratio BETWEEN 90 AND 100 THEN 'C'
    END AS abc_rank
FROM
  sales_composition_ratio
ORDER BY
  amount DESC
;
```

```
    category    | amount |  composition_ratio  |   cumulative_ratio    | abc_rank
----------------+--------+---------------------+-----------------------+----------
 ladys_fashion | 185000 | 19.3312434691745037 |  19.3312434691745037 | A
 mens_fashion  | 168000 | 17.5548589341692790 |  36.8861024033437827 | A
 book          | 130000 | 13.5841170323928945 |  50.4702194357366771 | A
 game          |  98000 | 10.2403343782654127 |  60.7105538140020899 | A
 dvd           |  72000 |  7.5235109717868339 |  68.2340647857889237 | A
 food          |  62000 |  6.4785788923719958 |  74.7126436781609195 | B
...
 supplement    |  36000 |  3.7617554858934169 | 100.0000000000000000 | C
```

構成比や構成比累計などの比率を計算するために、「3-2-3 2つの値の比率を計算する」(P.42) で紹介した、整数を実数に変換して割合を求める手法を用いています。なお、上記のコード例では、分母が全体売上や項目別累計売上など、必ず0より大きくなる値を用いるため、0除算のチェックを省略しています。また、項目別累計売上を求める際には、SUMウィンドウ関数を利用しています。

ワンポイント

本項では商品を対象にABC分析を行いましたが、商品のカテゴリ別で集計すると、また違った発見があります。もちろん、ABC分析以外でも構成比と構成比累計はデータ分布を把握するのに有効です。しっかり習得しましょう。

4-2-3 ファンチャートで商品の売れ行きの伸び率を見る

SQL FIRST_VALUEウィンドウ関数　　分析 ファンチャート

ファンチャートとは、ある基準となる時点を100%として、以降の数値の変動を見るグラフです。例えば、商品またはカテゴリ別の売上金額を状況を判断する場合、売上金額が大きいものはその傾向の判断は容易ですが、小さい金額では読み解くことが困難で、トレンドの変化や急成長している分野を見逃しがちです。

しかし、本節で解説するファンチャートを利用することで、金額の大小ではなく基準からの変化が百分率で示されるため、小さな変化も見逃すことなく、状況を判断できます。下図にファンチャートの例を示します。

図4.2.3.1：ファンチャート

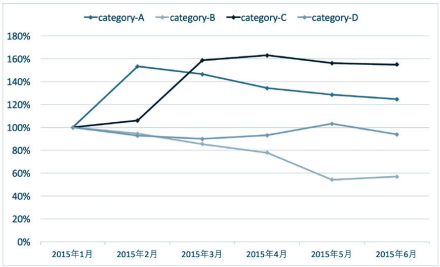
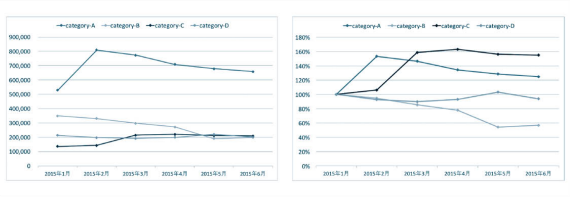


図4.2.3.2：月次の売上推移(左)とファンチャート(右)の比較



前図左の月次売上推移では、金額の少ない項目はその変化を読み取ることは困難ですが、右のファンチャートでは、変化率がグラフで表示されるため、成長や衰退を容易に把握できます。

ファンチャートを作成するためには、下表の例に示すデータを集計する必要があります。本項では、2015年1月の各カテゴリの売上を基準とし、それに対する以降の月の売上の変化率を集計します。

表4.2.3.1：ファンチャート作成に必要なデータ

日付	カテゴリ	売上	Rate
2015年1月	ladys_fashion	527,890	100.0%
2015年1月	mens_fashion	349,012	100.0%
2015年1月	book	135,020	100.0%
2015年1月	game	213,263	100.0%
2015年2月	ladys_fashion	809,698	153.4%
2015年2月	mens_fashion	330,281	94.6%
2015年2月	book	143,322	106.1%
2015年2月	game	197,948	92.8%
…	…	…	…

上表の結果を取得するためのSQLを、下記のコード例に示します。

まず、日付データから年と月の値だけを取り出し、年と月の単位で売上を求めます。続いて、求めた売上を時系列順に並べ替え、ファンチャートを作成するための基準となる月の売上に対する割合を求めます。今回、基準となる売上は、時系列順に並べ替えたときの最初の月の売上になるので、FIRST_VALUEウィンドウ関数を使用して値を求めています。

コード4.2.3.1：ファンチャート作成に必要なデータを取得するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
daily_category_amount AS (
 SELECT
     dt
   , category
   -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合は下記
   -- ■ BigQueryの場合、substringをsubstrに修正
   , substring(dt, 1, 4) AS year
   , substring(dt, 6, 2) AS month
   , substring(dt, 9, 2) AS date
   , SUM(price) AS amount
 FROM purchase_detail_log
 GROUP BY dt, category
)
, monthly_category_amount AS (
 SELECT
     concat(year, '-', month) AS year_month
     -- ■ Redshiftの場合はconcat関数を組み合わせるか、||演算子を用いる
```

```
    -- concat(concat(year, '-'), month) AS year_month
    -- year || '-' || month AS year_month
  , category
  , SUM(amount) AS amount
FROM
  daily_category_amount
GROUP BY
  year, month, category
)
SELECT
  year_month
, category
, amount
, FIRST_VALUE(amount)
    OVER(PARTITION BY category ORDER BY year_month, category ROWS UNBOUNDED PRECEDING)
  AS base_amount
, 100.0
  * amount
  / FIRST_VALUE(amount)
      OVER(PARTITION BY category ORDER BY year_month, category ROWS UNBOUNDED PRECEDING)
  AS rate
FROM
 monthly_category_amount
ORDER BY
 year_month, category
;
```

```
year_month |    category    | amount | base_amount |  rate
-----------+----------------+--------+-------------+--------
2014-01    | ladys_fashion | 263945 |      263945 | 100.00
2014-01    | mens_fashion  | 349012 |      349012 | 100.00
2014-01    | book          | 135020 |      135020 | 100.00
2014-01    | game          | 213263 |      213263 | 100.00
2014-02    | ladys_fashion | 404849 |      263945 | 153.38
2014-02    | mens_fashion  | 188667 |      349012 |  54.05
2014-02    | book          | 143322 |      135020 | 106.14
2014-02    | game          | 197948 |      213263 |  92.81
2014-03    | ladys_fashion | 386651 |      263945 | 146.48
2014-03    | mens_fashion  | 263634 |      349012 |  75.53
2014-03    | book          | 214231 |      135020 | 158.66
2014-03    | game          | 191777 |      213263 |  89.92
2014-04    | ladys_fashion | 321938 |      263945 | 121.97
2014-04    | mens_fashion  | 303950 |      349012 |  87.08
2014-04    | book          | 227336 |      135020 | 168.37
2014-04    | game          | 111065 |      213263 |  52.07
...
```

前述の実行結果（リスト4.2.3.1）を見ると、base_amountカラムには、FIRST_VALUEウィンドウ関数を用いて求めた2014年1月の売上が取得され、そのbase_amountに対する伸び率がrateカラムに計算されています。

ワンポイント

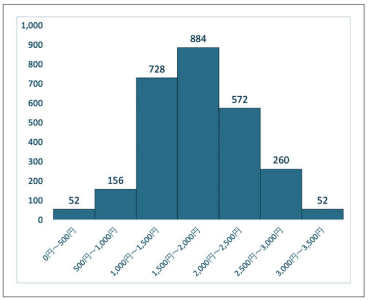
ファンチャート作成時に注意すべきは、どの時点での売上金額を基準点として採択するかで、成長傾向であるか減退傾向であるかの判断が大きく異なることです。例えば、8月に売上が毎年伸びる商品があるケースで8月を基準にすると、その商品は以降減退傾向のグラフになってしまうため、8月に伸びる事象がグラフに現れなくなります。したがって、この場合は季節変動の少ない平均的な月を基準点として選択します。7月～8月の売上変化を確認する、明確な目的がある場合は、直前の6月を基準点として採択します。この通り、必ず根拠をもって基準点を採択することが望ましいです。

4-2-4 ヒストグラムで購入価格帯を集計する

SQL width_bucket関数　　分析 度数分布表、ヒストグラム

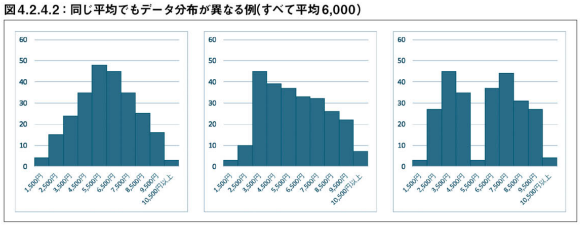
商品の価格に着目して、データの分布を確認できるヒストグラムを作成する方法を紹介します。ヒストグラムは、横軸に階級（データの範囲）と縦軸に度数（データの個数）を示すグラフで、データがどのように分散しているのかが一目で分かります。また、データの山が一番高い部分を「最頻値」と呼びます。

図4.2.4.1：ヒストグラムの例



データを取り扱う際に、平均値を用いて「ユーザーの平均購入額が最頻値」と誤って解釈して進めてしまうケースがありますが、実際はデータの分布によっては、最頻値は平均値に近い値にならないことがあります。下記に示す図はすべて同じ平均値となるデータですが、ヒストグラムを確認すると、データの分布は異なることが分かります。ヒストグラムを作成してデータ分布を確認することは重要です。

図4.2.4.2：同じ平均でもデータ分布が異なる例(すべて平均6,000)



- 左図： 正規分布に近い山なりのグラフ
  - 平均、最頻値付近にグラフの一番高い山が現れる
- 中央： 高い階級にデータが多く裾野が広いグラフ
  - 最頻値よりも平均が高くなる
- 右図： 山が割れているグラフ
  - 平均付近にデータがない

ヒストグラムの作成方法

ヒストグラムを作成するには、まず下表に示す度数分布表を作成します。

1. 最大値、最小値、範囲（最大値-最小値）を求める
2. 範囲を元に、いくつの階級に分けるか決めて、各階級下限、階級上限を求める
3. 各階級に入るデータの個数（度数）を調べ、その結果を整理して表にまとめる

表4.2.4.1：度数分布表

価格帯下限(以上)	価格帯上限(未満)	度数
0	500	52
500	1,000	156
1,000	1,500	728
1,500	2,000	884
2,000	2,500	572
2,500	3,000	260
3,000	3,500	52

作成した度数分布表をグラフ化したものがヒストグラムです。ヒストグラムの作法として、各棒グラフの間にスペースは設けません。本来、棒グラフは独立した項目を比較するために用いますが、ヒストグラムは連続しているデータであり、高さではなく面積でデータの分布を比較することから、グラフ間にスペースを設けずに表現します。

任意の階級数でヒストグラムを作成する

本項では購入されている商品の価格帯を対象に、ヒストグラムを作成するためのデータを抽出します。取得したいデータの形式を下表に示します。売上額はヒストグラムを作成する際には不要ですが、別途必要になった際に改めて取得するのは効率が悪いため、一緒に計算しておきます。

表4.2.4.2：ヒストグラムを作成する際に抽出したいデータ

価格帯下限(以上)	価格帯上限(未満)	度数	売上額
0	500	52	10,400
500	1,000	156	109,200
1,000	1,500	728	910,000
1,500	2,000	884	1,560,000
2,000	2,500	572	1,294,800
2,500	3,000	260	691,600
3,000	3,500	52	182,000

続いて、本項の購入ログサンプルを対象に、売上商品の最大、最小価格を求め、その範囲を10等分してヒストグラムを作成しましょう。

購入ログ内での売上金額の最大値（max_amount）と最小値（min_amount）、金額範囲（range_amount）を求めます。下記のコード例に示す通り、WITH句を用いてこれらの値を計算したテーブルをstatsとして定義します。金額範囲はあとで割り算を適用するため、numeric型で定義すると便利です。また、階級数（bucket_num）もstatsテーブル中に直接定義します。

コード4.2.4.1：最大値、最小値、範囲を求めるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
stats AS (
SELECT
    -- 金額の最大値
    MAX(price) AS max_price
    -- 金額の最小値
  , MIN(price) AS min_price
    -- 金額の範囲
  , MAX(price) - MIN(price) AS range_price
    -- 階級数
  , 10 AS bucket_num
```

```
FROM
  purchase_detail_log
)
SELECT *
FROM stats
;
```

```
 max_price | min_price | range_price | bucket_num
-----------+-----------+-------------+------------
      3500 |       200 |        3300 |         10
```

上記コードで定義したstatsテーブルを、もとのpurchase_logとCROSS JOINし、purchase_logの各売上金額に対して階級を判定していきます。

次のステップ 下記に示すコード例に示す通り、最小金額から最大金額の範囲を階級数で等分したいので、まずはすべての売上金額から最小金額を引き算し、階級を判定するための正規化金額（diff）を計算します。1階級の範囲（bucket_range）は全体の金額範囲を階級数で割ることで求められます。最後に、正規化金額を階級範囲で割り算し、FLOOR関数を用いて小数以下を切り捨てれば、その売上金額がどの階級に含まれるか判定できます。

ちなみに、多くのSQL処理系には、ヒストグラムを作成する関数が標準で用意されています。例えば、PostgreSQLでは、上記の手順をwidth_bucket関数で一度に計算できます。

コード4.2.4.2：各データに対する階級を求めるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
stats AS (
  -- コード4.2.4.1参照
)
, purchase_log_with_bucket AS (
  SELECT
      price
    , min_price
      -- 正規化金額：対象の金額から最小金額を引く
    , price - min_price AS diff
      -- 階級範囲：金額範囲を階級数で割る
    , 1.0 * range_price / bucket_num AS bucket_range

    -- 階級の判定：FLOOR(正規化金額 / 階級範囲)
    , FLOOR(
        1.0 * (price - min_price)
        / (1.0 * range_price / bucket_num)
        -- indexを1から始めるために1を足す
```

```
      ) + 1 AS bucket

    -- PostgreSQLの組み込み関数を使う場合
    -- , width_bucket(price, min_price, max_price, bucket_num) AS bucket
  FROM
    purchase_detail_log, stats
)
SELECT *
FROM purchase_log_with_bucket
ORDER BY amount
;
```

```
price | min_price | diff | bucket_range | bucket | width_bucket
------+-----------+------+--------------+--------+--------------
  200 |       200 |    0 |       330.00 |      1 |            1
  500 |       200 |  300 |       330.00 |      1 |            1
  800 |       200 |  600 |       330.00 |      2 |            2
  800 |       200 |  600 |       330.00 |      2 |            2
 1000 |       200 |  800 |       330.00 |      3 |            3
 1000 |       200 |  800 |       330.00 |      3 |            3
 1100 |       200 |  900 |       330.00 |      3 |            3
 1200 |       200 | 1000 |       330.00 |      4 |            4
 1200 |       200 | 1000 |       330.00 |      4 |            4
 1200 |       200 | 1000 |       330.00 |      4 |            4
 1200 |       200 | 1000 |       330.00 |      4 |            4
 1200 |       200 | 1000 |       330.00 |      4 |            4
 1400 |       200 | 1200 |       330.00 |      4 |            4
 1400 |       200 | 1200 |       330.00 |      4 |            4
 1400 |       200 | 1200 |       330.00 |      4 |            4
 1400 |       200 | 1200 |       330.00 |      4 |            4

(省略)

 2500 |       200 | 2300 |       330.00 |      7 |            7
 2600 |       200 | 2400 |       330.00 |      8 |            8
 2600 |       200 | 2400 |       330.00 |      8 |            8
 2700 |       200 | 2500 |       330.00 |      8 |            8
 2900 |       200 | 2700 |       330.00 |      9 |            9
 3500 |       200 | 3300 |       330.00 |     11 |           11
```

なお、上記の出力結果では、階級範囲を10としたはずなのに、最大値である3500の階級は「11」と判定されています。これは階級の判定ロジックに「階級下限以上〜階級上限未満」を使っているため、階級10の範囲が3,500未満となり、最大値のレコードが含まれなくなってしまうためです。

次のステップ すべてのレコードが指定した階級内に納まるように、クエリを改良しましょう。statsテーブルの定義で、下記のコード例に示す通り、階級上限を金額の最大値+1とすることで、すべてのレコードが必ず階級上限未満に含まれます。

コード4.2.4.3：階級上限の値を調整したクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
stats AS (
SELECT
    -- 金額の最大値 + 1
    MAX(price) + 1 AS max_price
    -- 金額の最小値
  , MIN(price) AS min_price
    -- 金額の範囲 + 1（実数）
  , MAX(price) + 1 - MIN(price) AS range_price
    -- 階級数
  , 10 AS bucket_num
FROM
  purchase_detail_log
)
purchase_log_with_bucket AS (
  -- コード4.2.4.2参照
)
SELECT *
FROM purchase_log_with_bucket
ORDER BY price
;
```

```
price | min_price | diff | bucket_range | bucket | width_bucket
------+-----------+------+--------------+--------+--------------
  200 |       200 |    0 |       330.10 |      1 |            1
  500 |       200 |  300 |       330.10 |      1 |            1
  800 |       200 |  600 |       330.10 |      2 |            2
  800 |       200 |  600 |       330.10 |      2 |            2
 1000 |       200 |  800 |       330.10 |      3 |            3
 1000 |       200 |  800 |       330.10 |      3 |            3
 1100 |       200 |  900 |       330.10 |      3 |            3
 1200 |       200 | 1000 |       330.10 |      4 |            4

 （省略）

 2500 |       200 | 2300 |       330.10 |      7 |            7
 2600 |       200 | 2400 |       330.10 |      8 |            8
 2600 |       200 | 2400 |       330.10 |      8 |            8
 2700 |       200 | 2500 |       330.10 |      8 |            8
 2900 |       200 | 2700 |       330.10 |      9 |            9
 3500 |       200 | 3300 |       330.10 |     10 |           10
```

この通り、階級上限値を調整したクエリ結果では、金額範囲に小数が含まれてしまっていますが、最大値のレコードも含めて1〜10の階級に収まっています。

次のステップ 最後に、求めた階級にしたがって度数を計算します。各階級の下限と上限を同時に表示する場合、下記コード例のlower_limitとupper_limitに示す手順で階級の下限と上限を計算します。

コード4.2.4.4：ヒストグラムを求めるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
stats AS (
  -- コード4.2.4.3参照
)
, purchase_log_with_bucket AS (
  -- コード4.2.4.2参照
)
SELECT
    bucket
    -- 階級の下限と上限を計算する
  , min_price + bucket_range * (bucket - 1) AS lower_limit
  , min_price + bucket_range * bucket AS upper_limit
    -- 度数をカウントする
  , COUNT(price) AS num_purchase
    -- 合計金額を計算する
  , SUM(price) AS total_amount
FROM
  purchase_log_with_bucket
GROUP BY
  bucket, min_price, bucket_range
ORDER BY bucket
;
```

```
bucket | lower_limit | upper_limit | num_purchase | total_amount
-------+-------------+-------------+--------------+--------------
     1 |      200.00 |      530.10 |            2 |          700
     2 |      530.10 |      860.20 |            2 |         1600
     3 |      860.20 |     1190.30 |            3 |         3100
     4 |     1190.30 |     1520.40 |           12 |        15900
     5 |     1520.40 |     1850.50 |           11 |        19000
     6 |     1850.50 |     2180.60 |            7 |        13700
     7 |     2180.60 |     2510.70 |           10 |        23200
     8 |     2510.70 |     2840.80 |            3 |         7900
     9 |     2840.80 |     3170.90 |            1 |         2900
    10 |     3170.90 |     3501.00 |            1 |         3500
```

任意の階級幅でヒストグラムを作成する

本項で説明した通り、価格の上限下限は計算で最適な価格帯を求めることも可能です。しかし、計算して求めた結果、小数点刻みで階級を区切るレポートとなると直感的ではありません。レポートを受け取る側にも納得感がある階級の刻みにすべきでしょう。

前述のコード例（コード4.2.4.3）のstatsテーブルで求めた、金額の最大値や最小値、金額範囲のように固定値に置き換えることで実現できます。下記のコード例は、0〜5,000の範囲を10階級に等分する場合のクエリです。出力結果には、500円刻みの階級でヒストグラムのデータが作成されていることが分かります。

コード4.2.4.5：ヒストグラムの上限と下限を手動で調整したクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

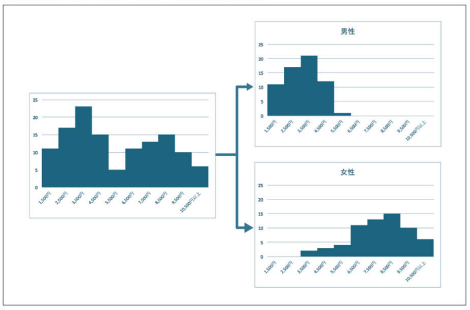
```
WITH
stats AS (
SELECT
    -- 金額の最大値
    5000 AS max_price
    -- 金額の最小値
  , 0 AS min_price
    -- 金額の範囲
  , 5000 AS range_price
    -- 階級数
  , 10 AS bucket_num
FROM
  purchase_detail_log
)
, purchase_log_with_bucket AS (
  -- コード4.2.4.2参照
)
SELECT
    bucket
    -- 階級の下限と上限を計算する
  , min_price + bucket_range * (bucket - 1) AS lower_limit
  , min_price + bucket_range * bucket AS upper_limit
    -- 度数をカウントする
  , COUNT(price) AS num_purchase
    -- 合計金額を計算する
  , SUM(price) AS total_amount
FROM
  purchase_log_with_bucket
GROUP BY
  bucket, min_price, bucket_range
ORDER BY bucket
;
```

```
 bucket | lower_limit | upper_limit | num_purchase | total_amount
--------+-------------+-------------+--------------+--------------
      1 |           0 |         500 |           52 |        10400
      2 |         500 |        1000 |          156 |       109200
      3 |        1000 |        1500 |          728 |       910000
      4 |        1500 |        2000 |          884 |      1560000
      5 |        2000 |        2500 |          572 |      1294800
      6 |        2500 |        3000 |          260 |       691600
      8 |        3500 |        4000 |           52 |       182000
```

ヒストグラムが割れてしまった時は

ヒストグラムの山が2つに割れてしまうことがあるかもしれません。もしかしたら、異なる母集団が存在し、異なる傾向が1つにまとまっていることも考えられます。このようなシーンに遭遇したら、データをさまざまな条件でフィルタして調査してみましょう。

図 4.2.4.2：ヒストグラムが2つの山に割れた場合



まとめ

売上の上昇・下降の原因を調査するように指示された際、「直近の売上」と「過去の売上」の2つのヒストグラムを作成し、2つの期間の差はどのようなところに現れているのか確認すると、その原因を探る糸口になります。例えば、ヒストグラムが全体的に下がっているのか、特定の範囲のみが下がっているのか、それだけでも明らかになれば、その後の調査範囲が絞られてきます。データの分布を確認するのに有効なのがヒストグラムなのです。

Chapter 5

ユーザーを把握するための データ抽出

どのようなユーザーがサービスを利用しているかを
ユーザー属性別に集計するなど、
ユーザーの行動パターンから属性を定義する手法や
サービスを継続して利用してもらえているのか、
サービスの実態を探る指標を紹介します。

5-1 ユーザ全体の特徴・傾向を見つける

サービスを提供することは、ユーザーに対して何かしらの価値を提供することです。サービス運用にあたり、サービス提供側がユーザーに関して知りたい情報は、下記の2つに大きく分類できます。

- ユーザーの属性（年齢、性別、居住地）
- ユーザーの行動（購入する商品、利用する機能、利用する頻度）

「どのような属性のユーザーが利用しているか」と「どのように利用しているのか」を把握せずに、サービスの改善を検討することはまさに暗中模索そのものです。

本節では、ユーザーの属性と行動に関する情報を集計し、ユーザー行動の調査やサービス改善の糸口となるレポートと集計するSQLを紹介します。

サンプルデータ

本節では、下記データ例の2つのテーブルを元に解説します。ECサイトでサービスの利用にはユーザー登録とログイン機能を有するものを想定しています。

ユーザー登録時には、プロフィール情報として性別と誕生日が入力され、同時に登録した日付と登録したデバイスが自動的に取得されます。また、サービス退会時には、退会日が保存され、ユーザーの情報自体は削除されません。

データ5.1.0.1: ユーザーマスタ(mst_users)テーブル

```
 user_id | sex | birth_date | register_date | register_device | withdraw_date
---------+-----+------------+---------------+-----------------+---------------
 U001    | M   | 1977-06-17 | 2016-10-01    | pc              |
 U002    | F   | 1953-06-12 | 2016-10-01    | sp              | 2016-10-10
 U003    | M   | 1965-01-06 | 2016-10-01    | pc              |
 U004    | F   | 1954-05-21 | 2016-10-05    | pc              |
 U005    | M   | 1987-11-23 | 2016-10-05    | sp              |
 U006    | F   | 1950-01-21 | 2016-10-10    | pc              | 2016-10-10
 U007    | F   | 1950-07-18 | 2016-10-10    | app             |
 U008    | F   | 2006-12-09 | 2016-10-10    | sp              |
 U009    | M   | 2004-10-23 | 2016-10-15    | pc              |
 U010    | F   | 1987-03-18 | 2016-10-16    | pc              |
```

データ5.1.0.2:アクションログ(action_log)テーブル

session	user_id	action	category	products	amount	stamp
989004ea	U001	view				2016-11-03 18:00:00
989004ea	U001	favorite	drama	D001		2016-11-03 18:05:31
989004ea	U001	add_cart	drama	D001		2016-11-03 18:07:34
989004ea	U001	add_cart	drama	D002		2016-11-03 18:09:51
989004ea	U001	purchase	drama	D001,D002	2000	2016-11-03 18:19:09
989004ea	U001	review	drama	D003		2016-11-03 18:23:09
762afcd3		view				2016-11-03 18:31:29
47db0370	U002	add_cart	drama	D001		2016-11-03 19:04:01
47db0370	U002	purchase	drama	D001	1000	2016-11-03 20:11:15
47db0370	U002	add_cart	drama	D002		2016-11-03 20:30:43
87b5725f	U002	view				2016-11-03 21:59:50
87b5725f	U002	add_cart	action	A004		2016-11-04 22:03:12
87b5725f	U002	add_cart	action	A005		2016-11-04 22:05:25
87b5725f	U002	add_cart	action	A006		2016-11-04 22:01:19
9afaf87c		view				2016-11-04 22:05:27
a836bca5	U003	view				2016-11-04 22:06:51

ページ閲覧時には、お気に入り登録やカート追加、購入、商品レビューなどの各アクションに応じて、それぞれfavorite、add_cart、purchase、reviewなどのアクション名を出力します。また、ログインユーザーはuser_idをログに出力します。user_idが空のレコードは非ログインユーザーの行動を示します。

サービス側で持つデータベースには、お気に入り登録やカート追加、購入、コメントで個々のテーブルが存在しますが、アクションログテーブルでは、これらを1つのテーブルにまとめて保存することで、UNIONせずにデータを取り扱えます。アクションログ設計の参考にしてください。

5-1-1 ユーザーのアクション数を集計する

SQL COUNT関数、COUNT(DISTINCT ～)、ROLLUP句

分析 UU、利用率、一人あたり

ユーザーがサービス内で提供している機能をどの程度利用しているかを集計する作業は、ユーザーの行動パターンを把握する上でも、実施する施策の効果を計る上でも、重要かつ高頻度で行う作業です。

アクションに関する指標を集計する

サービス内で提供している機能が、ある集計期間にユーザーにどれぐらい使われているのかを集計します。アクションUUをサービスを利用した全体のUUで割ったものを「利用率」とし、利用したユーザーが何回利用したかを示す「1人当たりのアクション数」を集計します。

下記コード例に、次表を集計するクエリを示します。

表5.1.1.1：アクション数の集計済みイメージ

アクション	アクションUU	アクション数	利用率	一人あたりアクション数
view	4,365	52,083	100%	11.9
add_cart	381	978	8.7%	2.6
purchase	234	819	5.4%	3.5

コード5.1.1.1：アクション数と割合を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
stats AS (
  -- ログ全体のユニークユーザー数を求める
  SELECT COUNT(DISTINCT session) AS total_uu
  FROM action_log
)
SELECT
    l.action
  -- アクションUU
  , COUNT(DISTINCT l.session) AS action_uu
  -- アクション数
  , COUNT(1) AS action_count
  -- 全体UU
  , s.total_uu
  -- 利用率: アクションUU / 全体UU
  , 100.0 * COUNT(DISTINCT l.session) / s.total_uu AS usage_rate
  -- 一人あたりアクション数: アクション数 / アクションUU
  , 1.0 * COUNT(1) / COUNT(DISTINCT l.session) AS count_per_user
FROM
    action_log AS l
  -- ログ全体のユニークユーザー数を全レコードに結合する
  CROSS JOIN
    stats AS s
GROUP BY
  l.action, s.total_uu
;
```

▼

```
  action  | action_uu | action_count | total_uu | usage_rate | count_per_user
----------+-----------+--------------+----------+------------+----------------
 view     |      4365 |        52083 |     4365 |     100.00 |          11.93
 add_cart |       381 |          978 |     4365 |       8.72 |           2.56
 purchase |       234 |          819 |     4365 |       5.36 |           3.50
```

上記のコード例では、別テーブルで全体UUを求め、その値をCROSS JOINでアクションログに結合して、アクションごとに集約した指標を求めています。

ログインユーザーと非ログインユーザーを分けて集計する

完全な会員向けのサービスではなく、ログインしていないユーザーや非会員ユーザーでも一部を閲覧・利用できるサービスの場合は、会員と非会員は分けて集計すべきです。サービスに対してロイヤリティが高いユーザーと低いユーザーで異なる傾向を発見できます。

ログイン、非ログイン、会員、非会員を判断するには、ログデータにsessionを付与し保存します。ログインした際に出力されるユーザーIDが空のレコードを非ログインとして判定し、sessionを用いてその数を集計します。まず、ログイン状態の区分を、user_idが含まれているかどうかで判別してみましょう。

コード5.1.1.2: ログイン状態を判別するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_log_with_status AS (
  SELECT
      session
    , user_id
    , action
    -- user_idがNULLまたは空文字でない場合はloginと判定
    , CASE WHEN COALESCE(user_id, '') <> '' THEN 'login' ELSE 'guest' END
      AS login_status
  FROM
    action_log
)
SELECT *
FROM
  action_log_with_status
;
```

```
 session | user_id |  action  | login_status
---------+---------+----------+--------------
 a284bc  | U001    | view     | login
 a284bc  | U001    | add_cart | login
 fadffs  | U002    | view     | login
 fadffs  | U002    | purchase | login
 fadffs  | U002    | view     | login
 fadffs  | U002    | comment  | login
 htyhty  |         | view     | guest
 htyhty  |         | add_cart | guest
 kefcwg  | U003    | view     | login
 kefcwg  | U003    | view     | login
 kefcwg  |         | view     | guest
 kefcwg  |         | add_cart | guest
```

user_idに値が含まれていればlogin_statusはlogin、含まれていない場合はguestとなるように加工します。続いて、このlogin_statusにしたがってアクション数とUUを集計します。

login_statusがguestとloginの場合、さらにログイン・非ログイン問わず全体（all）を同時に集計するため、ROLLUP句を使用します。

なお、ROLLUP句に対応していないRedshiftとBigQueryでは、「4-2-1 カテゴリ別の売上と小計を計算する」(P.113) で紹介したUNION ALLを用いて、同様の結果を得ることが可能です。

コード5.1.1.3:ログイン状態によるアクション数の集計クエリ

PostgreSQL Hive SparkSQL

```
WITH
action_log_with_status AS (
  -- コード5.1.1.2参照
)
SELECT
    COALESCE(action, 'all') AS action
  , COALESCE(login_status, 'all') AS login_status
  , COUNT(DISTINCT session) AS action_uu
  , COUNT(1) AS action_count
FROM
  action_log_with_status
GROUP BY
  -- ■ PostgreSQL, SparkSQLの場合は下記
  ROLLUP(action, login_status)
  -- ■ Hiveの場合は下記
  -- action, login_status WITH ROLLUP
;
```

```
  action  | login_status | action_uu | action_count
----------+--------------+-----------+--------------
 add_cart | guest        |        89 |          348
 add_cart | login        |       301 |          630
 add_cart | all          |       381 |          978
 purchase | login        |       234 |          819
 purchase | all          |       234 |          819
 view     | guest        |      3425 |        48922
 view     | login        |      1034 |         3161
 view     | all          |      4365 |        52083
 all      | all          |      4365 |        53880
```

上記コード例の出力結果は、ログデータであることから、ログイン状態の時のみuser_idが格納されます。そのため、非ログイン状態のユーザーがログインした場合はguestとlogin各々に1が計上され、allはsessionのユニーク数を集計していることから、guest＋login＝all とはなりません。

会員と非会員を分けて集計する

あくまでも非ログイン状態とログイン状態では機能が異なり、それぞれ抽出したいと言う明確な理由があれば、前述のコード5.1.1.3で問題ありません。しかし、一度でもログインしたことがあるユーザーは、ログインしていなくても会員として計上する場合は、ログデータを加工する必要があります。

その場合は、session_log_with_statusの定義を下記コード例に示す通りに変更することで、会員状態を計算できます。

コード5.1.1.4: 会員状態を判別するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_log_with_status AS (
  SELECT
      session
    , user_id
    , action
    -- ログをタイムスタンプ順に並べ、一度でもログインしたことのあるセッションの場合、
    -- それ以降のログのステータスをmemberとする
    , CASE
        WHEN
          COALESCE(MAX(user_id)
            OVER(PARTITION BY session ORDER BY stamp
              ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
            , '') <> ''
          THEN 'member'
        ELSE 'none'
      END AS member_status
    , stamp
  FROM
    action_log
)
SELECT *
FROM
  action_log_with_status
;
```

```
 session | user_id |  action  | member_status |        stamp
---------+---------+----------+---------------+---------------------
 8ms2kl  |         | view     | none          | 2016-10-16 14:00:00
 8ms2kl  | U004    | view     | member        | 2016-10-17 14:00:00
 8ms2kl  |         | view     | member        | 2016-10-18 14:00:00
 a284bc  | U001    | view     | member        | 2016-10-16 23:25:10
 a284bc  | U001    | add_cart | member        | 2016-10-16 23:25:10
 fadffs  | U002    | view     | member        | 2016-10-17 12:00:00
 fadffs  | U002    | purchase | member        | 2016-10-17 12:00:00
 fadffs  | U002    | purchase | member        | 2016-10-17 12:10:00
```

```
fadffs  | U002    | view     | member        | 2016-10-17 12:10:00
htyhty  |         | add_cart | none          | 2016-10-17 12:20:00
htyhty  |         | view     | none          | 2016-10-17 12:20:00
jd4duq  |         | view     | none          | 2016-10-17 15:00:00
jd4duq  | U005    | purchase | member        | 2016-10-18 12:00:00
jd4duq  | U005    | view     | member        | 2016-10-18 12:00:00
kefcwg  | U003    | view     | member        | 2016-10-17 13:00:00
kefcwg  | U003    | view     | member        | 2016-10-17 13:10:00
kefcwg  |         | view     | member        | 2016-10-18 12:30:00
kefcwg  |         | add_cart | member        | 2016-10-18 12:30:00
```

ウィンドウ関数でsessionごとにパーティションを設定することで、そのsessionで一度でもログインしたことがあれば、MAX(user_id)でユーザーIDが取得できます。さらに、初めてログイン状態になる前は非会員として扱いたい場合には、ORDER BY stampをウィンドウ関数内で指定することで実現できます。

上記コード例の出力結果に示す通り、session=8ms2klのユーザーは、10月16日時点ではログインしたことがないため、member_statusはnoneです。しかし、10月18日に改めて未ログイン状態でアクセスした際は、10月17日にuser_id=U004としてログインしたログがあるため、member_statusはmemberと判定されています。

ワンポイント

本項で紹介したクエリでは、未ログイン状態におけるユーザーIDカラムの値が、空文字またはNULLの場合を考慮し、COALESCE関数を用いて空文字に寄せています。

ただし、未ログイン時のユーザーIDが空文字で格納されていると、COUNT(DISTINCT user_id)の結果に未ログインユーザーのカウントが1件余分にカウントされます。COUNT(DISTINCT user_id)で正しくユニークユーザー数を取得するには、未ログイン時のユーザーIDはNULLの方が都合が良いでしょう。

空の値としてNULLを用いるか、空文字などの特殊な値を用いるかは、クエリによって最適な方法が変わるため、COALESCE関数やNULLIF関数を用いて相互に変換する術を覚えておくと役に立ちます。

関連項目	3-1-5 欠損値をデフォルト値に置き換える 3-3-1 グループの特徴を捉える 6-1-1 日次の訪問回数・訪問者数・ページビューを集計する

5-1-2 年齢別区分を集計する

SQL CASE式、CAST　　分析 年齢別区分

会員情報を保持するサービスでは、どのようなユーザーに利用されているのか把握して、ユーザーの利用実態がサービス提供側の意図通りであるか確認する場面があります。また、広告のデザインやキャッチフレーズを検討するため、ユーザー属性を集計する場面もあります。ユーザーの属性を定義し、集計することで、さまざまなレポートが作成可能になります。

本項では視聴率の分析に用いられるM1層、F1層（以降年齢別区分）と呼ばれる属性を、性別と年齢から導き出して設定し、その人数を集計する方法を紹介します。年齢別区分は、下記の表にそれぞれの性別と年齢を記載します。

表5.1.2.1:年齢別区分一覧

年齢別区分	内訳
C層　※1	4-12歳の男女
T層　※2	13-19歳の男女
M1層　※3	20-34歳の男性
M2層	35-49歳の男性
M3層	50歳以上の男性
F1層　※4	20-34歳の女性
F2層	35-49歳の女性
F3層	50歳以上の女性

※1：Cは子供を表すChildの意味
※2：Tは Teenagerの意味
※3：Mは男性を表すMaleの意味
※4：Fは女性を表すFemaleの意味

上記の表から、ユーザーの年齢別区分を集計します。年齢そのものの保存では、登録日から日数が経過すると、データベースに保存されている年齢と実年齢が合致しなくなります。そのため、通常は誕生日のみ保存し、年齢は表示時や集計時に計算して出力します。

年齢は「3-2-5 日付・時刻を計算する」(P.47) で紹介した通り、誕生日と特定の日付を整数型で表現し、その差の値を10000で割る方法が簡単です。

誕生日を元に、特定の日付（2017年1月1日）時点における年齢を計算するクエリを、下記のコード例に示します。誕生日と特定の日付を整数型で表現した結果に「mst_users_with_int_birth_date」と名付け、誕生日情報を付与したユーザーマスタに、「mst_users_with_age」と名付けておきます。

コード5.1.2.1：ユーザーの誕生日を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_users_with_int_birth_date AS (
  SELECT
      *
    -- 特定の日付(2017年1月1日)の整数表現
    , 20170101 AS int_specific_date
    -- 文字列の生年月日を、日付の整数表現に変換
```

```
    -- ■ PostgreSQL, Redshiftの場合は下記
    , CAST(replace(substring(birth_date, 1, 10), '-', '') AS integer) AS int_birth_date
    -- ■ BigQueryの場合は下記
    -- , CAST(replace(substr(birth_date, 1, 10), '-', '') AS int64) AS int_birth_date
    -- ■ Hive, SparkSQLの場合は下記
    -- , CAST(regexp_replace(substring(birth_date, 1, 10), '-', '') AS int)
    --   AS int_birth_date
  FROM
    mst_users
)
, mst_users_with_age AS (
  SELECT
    *
    -- 特定の日付(2017年1月1日)における年齢
    , floor((int_specific_date - int_birth_date) / 10000) AS age
  FROM
    mst_users_with_int_birth_date
)
SELECT
    user_id, sex, birth_date, age
FROM
  mst_users_with_age
;
```

```
 user_id | sex | birth_date | age
---------+-----+------------+-----
 U001    | M   | 1977-06-17 |  39
 U002    | F   | 1953-06-12 |  63
 U003    | M   | 1965-01-06 |  51
 U004    | F   | 1954-05-21 |  62
 U005    | M   | 1987-11-23 |  29
 U006    | F   | 1950-01-21 |  66
 U007    | F   | 1950-07-18 |  66
 U008    | F   | 2006-12-09 |  10
 U009    | M   | 2004-10-23 |  12
 U010    | F   | 1987-03-18 |  29
```

次のステップ 上記の出力結果を元に、年齢別区分をユーザーに付与するクエリを、下記のコード例に示します。

コード5.1.2.2：性別と年齢から年齢別区分を計算するクエリ

```
WITH
mst_users_with_int_birth_date AS (
 -- コード5.1.2.1参照
)
, mst_users_with_age AS (
```

```
 -- コード5.1.2.1参照
)
, mst_users_with_category AS (
  SELECT
      user_id
    , sex
    , age
    , CONCAT(
        CASE
          WHEN 20 <= age THEN sex
          ELSE ''
        END
      , CASE
          WHEN age BETWEEN 4  AND 12 THEN 'C'
          WHEN age BETWEEN 13 AND 19 THEN 'T'
          WHEN age BETWEEN 20 AND 34 THEN '1'
          WHEN age BETWEEN 35 AND 49 THEN '2'
          WHEN age >= 50 THEN '3'
        END
      ) AS category
  FROM
    mst_users_with_age
)
SELECT *
FROM
  mst_users_with_category
;
```

```
 user_id | sex | age | category
---------+-----+-----+----------
 U001    | M   |  39 | M2
 U002    | F   |  63 | F3
 U003    | M   |  51 | M3
 U004    | F   |  62 | F3
 U005    | M   |  29 | M1
 U006    | F   |  66 | F3
 U007    | F   |  66 | F3
```

上記コード例のcategoryカラムで、年齢別区分を計算しています。まず、年齢が20歳以上の場合に、性別の接頭辞として「M」または「F」を出力します。次に、年齢区分に従って、「C」「T」「1」「2」「3」のコードを出力し、最後に両者をCONCAT関数で結合しています。なお、年齢が3歳以下のユーザーの場合は、年齢区分のコードがNULLとなり、CONCATの結果もNULLとなります。

次のステップ 整形後のデータを用いて、COUNT関数でその数を集計するコード例を下記に示します。

コード5.1.2.3：年齢別区分ごとの人数を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_users_with_age AS (
 -- コード5.1.2.1参照
)
, mst_users_with_category AS (
 -- コード5.1.2.1参照
)
SELECT
    category
  , COUNT(1) AS user_count
FROM
  mst_users_with_category
GROUP BY
  category
;
```

```
 category | user_count
----------+------------
 M3       |        235
 M2       |        567
 M1       |        895
 F3       |        357
 F2       |        534
 F1       |        704
 T        |       1268
 C        |        345
```

ワンポイント

年齢を単純に集計すると、デモグラフィックとしては粒度が細かすぎて、特徴を捉えることは困難です。ただし、趣味嗜好が多様化するサービスでは、M1層やF1層などのユーザー属性が適切ではないと判断できた場合は、独自に新たな属性を定義しましょう。年齢幅が広いと判断できるのであれば、10代、20代、30代などと定義し直してCASE式を修正しましょう。

関連項目	3-1-1 コード値をラベルに置き換える 3-2-5 日付・時刻を計算する

5-1-3 年齢別区分ごとの特徴を抽出する

SQL JOIN、GROUP BY　　分析 年齢別区分

サービスの利用形態がユーザー属性で異なる点があることが分かれば、商品や記事をユーザー属性に合致させる、クリックされやすくする、購入を促進するなどの目的で、オススメする記事や商品を変更する施策を検討できます。

前項で使用した年齢別区分を用いて、それぞれの購入した商品のカテゴリを集計した下図のレポートを作成するSQLを、下記のコード例に示します。

図 5.1.3.1：カテゴリ別×年齢別区分

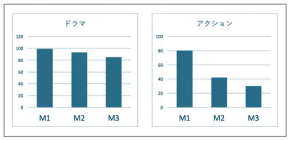
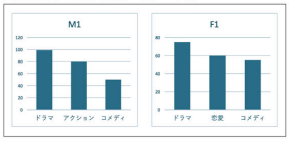


図 5.1.3.2：年齢区分別×カテゴリ別



コード 5.1.3.1：年齢別区分とカテゴリを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

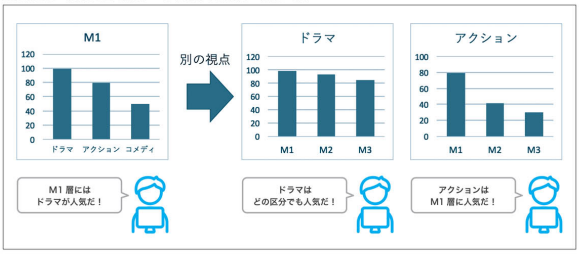
```
WITH
mst_users_with_int_birth_date AS (
  -- コード5.1.2.1参照
)
, mst_users_with_age AS (
  -- コード5.1.2.1参照
)
, mst_users_with_category AS (
  -- コード5.1.2.2参照
)
SELECT
    p.category AS product_category
  , u.category AS user_category
  , COUNT(*) AS purchase_count
FROM
    action_log AS p
  JOIN
    mst_users_with_category AS u
  ON p.user_id = u.user_id
WHERE
  -- 購入ログに絞り込む
  action = 'purchase'
GROUP BY
  p.category, u.category
```

```
ORDER BY
  p.category, u.category
;
```

```
 product_category | user_category | purchase_count
------------------+---------------+----------------
 action           | M1            |             81
 action           | M2            |             40
 drama            | M1            |            102
...
```

上記の出力結果があれば、図5.1.3.1に示す、横軸にカテゴリごとに年齢別区分を集計することも、図5.1.3.2に示す、年齢区分ごとにカテゴリを集計することも可能です。単独のグラフのみでは、特徴を捉えづらいこともあるため、双方のグラフを確認して特徴を捉えるように心掛けましょう。

図5.1.3.3：複数の図を見ることで異なる判断が可能になる



ワンポイント

前述の「4-2-2 ABC分析で売れ筋を判別する」(P.115) で説明した、ABC分析や構成比累計を本項記載のレポートに加えることで、より視認性が向上します。

関連項目 | 4-2-2 ABC分析で売れ筋を判別する

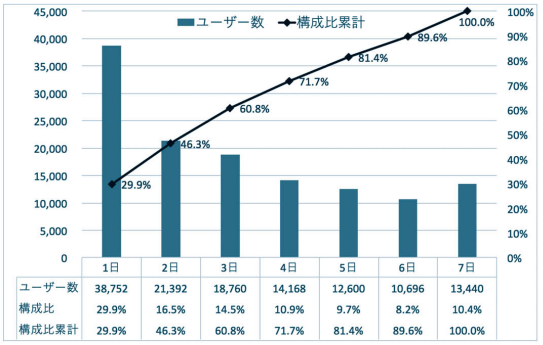
5-1-4 ユーザーの訪問頻度を集計する

SQL SUMウィンドウ関数　　分析 訪問頻度

ユーザーが1週間もしくは月間で何日サービスを利用しているかを知ることは、以降の分析業務に役立ちます。例えば、ニュースサイトで週に1回しか訪問しないユーザーと毎日訪問するユーザーでは利用する機能やアクセスする時間、行動パターンなどに相違があります。

本項では、サービスを週に何日利用しているユーザーが何人いるか抽出する方法を紹介します。下図に示す、「一週間におけるユーザーの利用日数と構成比」を作成する手順を説明しましょう。

図 5.1.4.1：一週間におけるユーザーの利用日数と構成比



	1日	2日	3日	4日	5日	6日	7日
ユーザー数	38,752	21,392	18,760	14,168	12,600	10,696	13,440
構成比	29.9%	16.5%	14.5%	10.9%	9.7%	8.2%	10.4%
構成比累計	29.9%	46.3%	60.8%	71.7%	81.4%	89.6%	100.0%

次に示すコード例に、週に何日利用しているかを集計するクエリを示します。

ユーザーID、アクションとその日時が記録されているaction_logテーブルに対して、ユーザーIDごとに日付をDISTINCTを用いて集計することで、起動日数を集計します。なお、時刻まで記録されている場合は、日付のみにデータを加工して集計します。

コード5.1.4.1：週に何日利用しているかを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_log_with_dt AS (
  SELECT *
    -- タイムスタンプから日付を抽出
    -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、substringで日付部分を抽出する
    , substring(stamp, 1, 10) AS dt
    -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、substrが利用できる
    -- , substr(stamp, 1, 10) AS dt
  FROM action_log
)
, action_day_count_per_user AS (
  SELECT
      user_id
    , COUNT(DISTINCT dt) AS action_day_count
  FROM
    action_log_with_dt
  WHERE
    -- 2016年11月1日～11月7日の1週間分を対象とする
    dt BETWEEN '2016-11-01' AND '2016-11-07'
  GROUP BY
    user_id
)
SELECT
    action_day_count
  , COUNT(DISTINCT user_id) AS user_count
FROM
  action_day_count_per_user
GROUP BY
  action_day_count
ORDER BY
  action_day_count
;
```

```
 action_day_count | user_count
------------------+------------
                1 |      38752
                2 |      21392
                3 |      18760
...
```

上記のコード例を基本にして、本節で紹介した構成比と構成比累計を算出するクエリを、次のコード例に示します。

コード5.1.4.2：構成比と構成比累計を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_day_count_per_user AS (
  -- コード5.1.4.1 参照
)
SELECT
    action_day_count
  , COUNT(DISTINCT user_id) AS user_count

  -- 構成比
  , 100.0
    * COUNT(DISTINCT user_id)
    / SUM(COUNT(DISTINCT user_id)) OVER()
    AS composition_ratio

  -- 構成比累計
  , 100.0
    * SUM(COUNT(DISTINCT user_id))
        OVER(ORDER BY action_day_count
          ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    / SUM(COUNT(DISTINCT user_id)) OVER()
    AS cumulative_ratio
FROM
  action_day_count_per_user
 GROUP BY
   action_day_count
ORDER BY
  action_day_count
;
```

```
action_day_count | user_count | composition_ratio | cumulative_ratio
-----------------+------------+-------------------+-----------------
               1 |      38752 |           29.8533 |          29.8533
               2 |      21392 |           16.4797 |          46.3330
               3 |      18760 |           14.4521 |          60.7851
...
```

まとめ

本項の応用版として、「5-2-5 利用日数に応じた定着率を集計する」（P.199）では、この利用日数に応じて、どのような違いがあるかを集計する方法を紹介します。

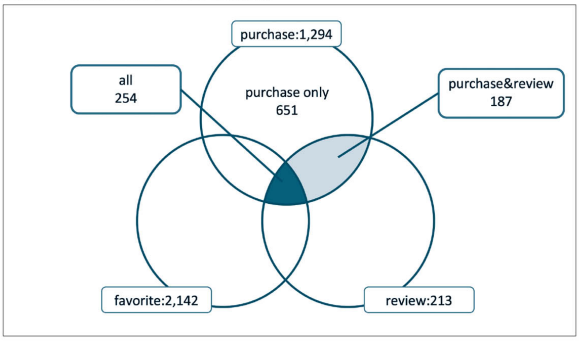
関連項目	5-2-5 利用日数に応じた定着率を集計する

5-1-5 ベン図でユーザーのアクションを集計する

SQL SIGN関数、SUM関数、CASE式、CUBE句　　分析 ベン図

サービス内で複数の機能を提供していても、その機能を漏れなく使っているユーザーは多くはありません。各種機能の利用状況を調べ、提供している機能がユーザーに受け入れられているか、想定している通りに使ってもらえているのかなどを確認する必要があります。本項では、下記に示すベン図を作成するSQLを紹介します。

図5.1.5.1：3つのアクションをベン図で表現した図



まず、ユーザー単位でログを集約し、purchase、review、favoriteの3アクションを行ったログが存在するかどうかを、0と1のフラグで付与します。

コード5.1.5.1：ユーザーごとのアクションフラグを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_action_flag AS (
  -- ユーザーごとに、行ったアクションについて0, 1のフラグを付与する
  SELECT
      user_id
    , SIGN(SUM(CASE WHEN action = 'purchase' THEN 1 ELSE 0 END)) AS has_purchase
    , SIGN(SUM(CASE WHEN action = 'review'   THEN 1 ELSE 0 END)) AS has_review
    , SIGN(SUM(CASE WHEN action = 'favorite' THEN 1 ELSE 0 END)) AS has_favorite
  FROM
```

```
    action_log
  GROUP BY
    user_id
)
SELECT *
FROM user_action_flag;
```

```
 user_id | has_purchase | has_review | has_favorite
---------+--------------+------------+--------------
 U001    |            1 |          1 |            1
 U002    |            1 |          0 |            1
 U003    |            1 |          0 |            0
 U004    |            1 |          0 |            0
...
```

上記コード例の出力結果で、has_purchase、has_review、has_favoriteカラムはそれぞれ、購入、レビュー、お気に入り登録の各アクションを実行したことがある場合は1、実行していない場合は0となります。

なお、レポートで利用するツールには、上記のデータをファイルとして入力すると、ベン図が作成可能なものもあります。本項では、ベン図の作成に必要なデータの集計もSQLで実装します。

次のステップ 上記コード例の出力結果を単純に3つのアクションフラグで集約するだけでは、ベン図を作成するために必要なすべての値を求めることはできません。例えば、「購入アクションを行ったユーザー数」や「購入とレビューを行ったユーザー数」など、1つのアクションや2つのアクションの組み合わせに対するユーザー数などが必要です。

標準SQLに定義されているCUBE句を用いると、これらの組み合わせをすべて考慮した集計を簡単に実現できます。下記コード例に、CUBE句を用いて全アクションの組み合わせでユーザー数をカウントするクエリを示します。ただし、CUBE句を実装しているミドルウェアは、本書で扱うミドルウェアの中ではPostgreSQLのみです。

コード5.1.5.2：すべてのアクションの組み合わせについてユーザー数をカウントするクエリ

PostgreSQL

```
WITH
user_action_flag AS (
  -- コード5.1.5.1参照
)
, action_venn_diagram AS (
  -- CUBEを用いて、アクションのすべての組み合わせを求める
  SELECT
      has_purchase
    , has_review
    , has_favorite
```

```
    , COUNT(1) AS users
  FROM
    user_action_flag
  GROUP BY
    CUBE(has_purchase, has_review, has_favorite)
)
SELECT *
FROM action_venn_diagram
ORDER BY
  has_purchase, has_review, has_favorite
;
```

```
 has_purchase | has_review | has_favorite | users
--------------+------------+--------------+-------
            0 |          0 |            0 |    15
            0 |          0 |            1 |     2
            0 |          0 |              |    17
            0 |          1 |            0 |     6
            0 |          1 |              |     6
            0 |            |            0 |    21
            0 |            |            1 |     2
            0 |            |              |    23
            1 |          0 |            0 |     5
            1 |          0 |            1 |     2
            1 |          0 |              |     7
            1 |          1 |            0 |     4
            1 |          1 |              |     4
            1 |            |            0 |     9
            1 |            |            1 |     2
            1 |            |              |    11
              |          0 |            0 |    20
              |          0 |            1 |     4
              |          0 |              |    24
              |          1 |            0 |    10
              |          1 |              |    10
              |            |            0 |    30
              |            |            1 |     4
              |            |              |    34
```

上記の出力結果で、has_purchase、has_review、has_favoriteカラムの値が空（NULL）になっているレコードは、そのカラムの値を問わないことを示します。

別解 上記コード例のクエリを、CUBE句を用いずに標準SQLの構文を用いて書き換えた場合は、下記コード例に示すクエリとなります。

コード5.1.5.3：CUBE句を使用せず、標準SQLの構文でCUBE句の結果を表すクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_action_flag AS (
  -- コード5.1.5.1参照
)
, action_venn_diagram AS (
  -- すべてのアクションの組み合わせについて個別に計算し、UNION ALLで結果をまとめる

  -- 3つのアクションの組み合わせについて集計
    SELECT has_purchase, has_review, has_favorite, COUNT(1) AS users
    FROM user_action_flag
    GROUP BY has_purchase, has_review, has_favorite

  -- 3つのアクションのうち2つのアクションの組み合わせについて集計
  UNION ALL
    SELECT NULL AS has_purchase, has_review, has_favorite, COUNT(1) AS users
    FROM user_action_flag
    GROUP BY has_review, has_favorite
  UNION ALL
    SELECT has_purchase, NULL AS has_review, has_favorite, COUNT(1) AS users
    FROM user_action_flag
    GROUP BY has_purchase, has_favorite
  UNION ALL
    SELECT has_purchase, has_review, NULL AS has_favorite, COUNT(1) AS users
    FROM user_action_flag
    GROUP BY has_purchase, has_review

  -- 3つのアクションのうち1つのアクションについて集計
  UNION ALL
    SELECT NULL AS has_purchase, NULL AS has_review, has_favorite, COUNT(1) AS users
    FROM user_action_flag
    GROUP BY has_favorite
  UNION ALL
    SELECT NULL AS has_purchase, has_review, NULL AS has_favorite, COUNT(1) AS users
    FROM user_action_flag
    GROUP BY has_review
  UNION ALL
    SELECT has_purchase, NULL AS has_review, NULL AS has_favorite, COUNT(1) AS users
    FROM user_action_flag
    GROUP BY has_purchase

  -- アクションを問わず、すべてのユーザーについて集計
  UNION ALL
    SELECT
      NULL AS has_purchase, NULL AS has_review, NULL AS has_favorite, COUNT(1) AS users
    FROM user_action_flag
)
```

```
SELECT *
FROM action_venn_diagram
ORDER BY
  has_purchase, has_review, has_favorite
;
```

上記のコード例はすべてのミドルウェアで動作しますが、UNION ALLを多用するため、パフォーマンスはあまり良くはありません。

別解 HiveとBigQuery、SparkSQLでは、前述の「3-3-4 横持ちのデータを縦持ちに変換する」（P.70）で紹介した手法を応用して、各アクションフラグのカラムに擬似的にNULLを含むレコードを作成することで、CUBE句と同様の結果を得ることができます。

コード5.1.5.4：擬似的にNULLを含むレコードを追加し、CUBE句と同様の結果を得るクエリ

Hive BigQuery SparkSQL

```
WITH
user_action_flag AS (
  -- コード5.1.5.1参照
)
, action_venn_diagram AS (
  SELECT
      mod_has_purchase AS has_purchase
    , mod_has_review   AS has_review
    , mod_has_favorite AS has_favorite
    , COUNT(1) AS users
  FROM
      user_action_flag
    -- 各カラムにNULLを含むレコードを擬似的に追加する
    -- ■ BigQueryの場合、CROSS JOINとunnest関数を用いる
     CROSS JOIN unnest(array[has_purchase, NULL]) AS mod_has_purchase
     CROSS JOIN unnest(array[has_review,   NULL]) AS mod_has_review
     CROSS JOIN unnest(array[has_favorite, NULL]) AS mod_has_favorite
    -- ■ Hive, SparkSQLの場合、LATERAL VIEWとexplode関数を用いる
    -- LATERAL VIEW explode(array(has_purchase, NULL)) e1 AS mod_has_purchase
    -- LATERAL VIEW explode(array(has_review,   NULL)) e2 AS mod_has_review
    -- LATERAL VIEW explode(array(has_favorite, NULL)) e3 AS mod_has_favorite
  GROUP BY
    mod_has_purchase, mod_has_review, mod_has_favorite
)
SELECT *
FROM action_venn_diagram
ORDER BY
  has_purchase, has_review, has_favorite
;
```

次のステップ 前述のコード5.1.5.2に示した結果でもベン図を書くことはできますが、可読性が高い形式になるように、クエリ結果を整形してみましょう。また、各アクションの組み合わせでのユーザー数の構成比も一緒に計算します。

コード5.1.5.5：ベン図を作成するためのデータを整形するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_action_flag AS (
  -- コード5.1.5.1参照
)
, action_venn_diagram AS (
  -- コード5.1.5.2~5.1.5.4参照
)
SELECT
  -- 0, 1のフラグを文字列に整形
    CASE has_purchase
      WHEN 1 THEN 'purchase' WHEN 0 THEN 'not purchase' ELSE 'any'
    END AS has_purchase
  , CASE has_review
      WHEN 1 THEN 'review' WHEN 0 THEN 'not review' ELSE 'any'
    END AS has_review
  , CASE has_favorite
      WHEN 1 THEN 'favorite' WHEN 0 THEN 'not favorite' ELSE 'any'
    END AS has_favorite
  , users
    -- 全ユニークユーザー数に対する割合を求める
  , 100.0 * users
    / NULLIF(
    -- すべてのアクションについてNULLであるユーザー数が、全ユニークユーザー数となるので、
    -- その行のユーザー数をWindow関数で求める
        SUM(CASE WHEN has_purchase IS NULL
                  AND has_review   IS NULL
                  AND has_favorite IS NULL
                 THEN users ELSE 0 END) OVER()
      , 0)
    AS ratio
FROM
  action_venn_diagram
ORDER BY
  has_purchase, has_review, has_favorite
;
```

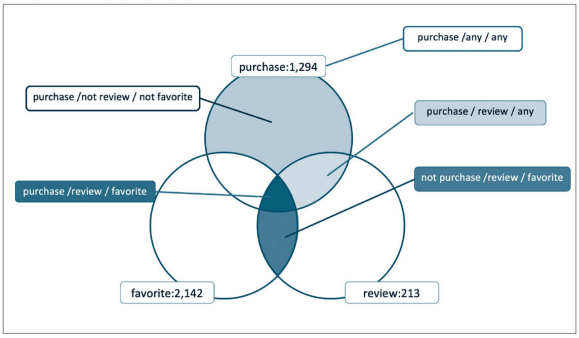
```
 has_purchase | has_review | has_favorite | users |   ratio
--------------+------------+--------------+-------+-----------
 any          | any        | any          |    34 | 100.0000
 any          | any        | favorite     |     4 |  11.7647
 any          | any        | not favorite |    30 |  88.2352
 any          | review     | any          |    10 |  29.4117
```

```
any          | review     | not favorite |    10 |  29.4117
any          | not review | any          |    24 |  70.5882
any          | not review | favorite     |     4 |  11.7647
any          | not review | not favorite |    20 |  58.8235
not purchase | any        | any          |    23 |  67.6470
not purchase | any        | favorite     |     2 |   5.8823
not purchase | any        | not favorite |    21 |  61.7647
not purchase | review     | any          |     6 |  17.6470
not purchase | review     | not favorite |     6 |  17.6470
not purchase | not review | any          |    17 |  50.0000
not purchase | not review | favorite     |     2 |   5.8823
not purchase | not review | not favorite |    15 |  44.1176
purchase     | any        | any          |    11 |  32.3529
purchase     | any        | favorite     |     2 |   5.8823
purchase     | any        | not favorite |     9 |  26.4705
purchase     | review     | any          |     4 |  11.7647
purchase     | review     | not favorite |     4 |  11.7647
purchase     | not review | any          |     7 |  20.5882
purchase     | not review | favorite     |     2 |   5.8823
purchase     | not review | not favorite |     5 |  14.7058
```

上記コード例の出力結果で、アクションの組み合わせが示す範囲を図示したものが、下図です。

例えば、「has_purchase／has_review／has_favorite」が「purchase／any／any」となっているレコードは投稿したユーザー数を表し、「purchase／not review／any」のレコードは、投稿してコメントしなかったユーザー数を表します。また、「any／any／any」は全ユニークユーザー数を表します。

図 5.1.5.2：データの示す範囲について



ワンポイント

本項の例はECサイトですが、例えば、SNSなどのサイトでは下記に挙げるユーザーが存在します。

- 投稿をせずに閲覧をメインに行うユーザー
- 投稿を積極的に行うユーザー
- 投稿はしないが、コメント機能を中心に利用するユーザー
- 投稿もコメントも積極的に利用するユーザー

実施する施策で効果のあるユーザー層がどれぐらい存在するかをベン図で確認して、ターゲットがズレていないか確認しましょう。

関連項目	3-4-3 条件のフラグを0と1で表現する

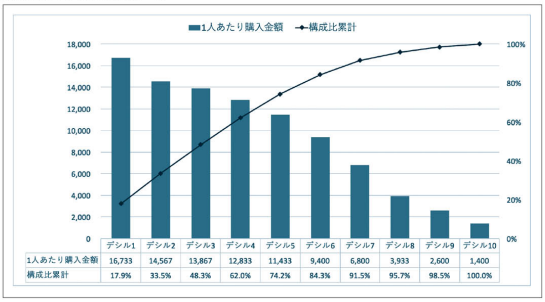
5-1-6 デシル分析でユーザーを10段階のグループに分ける

SQL NTILEウィンドウ関数　　分析 デシル分析

ユーザーの特徴を分析する際に、性別や年齢などのデータがあれば属性に分けて特徴を調べられます。

デモグラフィックなデータが存在しない場合は、ユーザーのアクションで属性を定義できます。属性を与えることで、各属性の特徴を分析して、よりサービスの利用を推し進めるため、属性に適した施策を検討することが可能になります。本項では「デシル分析」と呼ばれる、データを10分割して重要度を把握する手法を紹介します（デシル分析の「デシ」は10分の1を意味します）。

図 5.1.6.1：デシル分析イメージ



	デシル1	デシル2	デシル3	デシル4	デシル5	デシル6	デシル7	デシル8	デシル9	デシル10
1人あたり購入金額	16,733	14,567	13,867	12,833	11,433	9,400	6,800	3,933	2,600	1,400
構成比累計	17.9%	33.5%	48.3%	62.0%	74.2%	84.3%	91.5%	95.7%	98.5%	100.0%

本項では、ユーザーの購入金額に応じてランク分けして、重要度を把握するレポートを作成します。デシル分析の手順は以下の通りです。

1. ユーザーを購入金額の多い順番に並べ替える。
2. 並び替えたユーザーの上位から10%ずつデシル1からデシル10までのグループに割り当てる。
3. 各グループの購入金額の合計を集計する。
4. 全体の購入金額に対して、各デシルでの購入金額の割合（構成比）を計算する。
5. 上位から累積でどの程度の比率を占めるかの構成比累計を集計する。

上記の手順に従い、下表のレポートを作成するためのSQLをコード例に示します。

表5.1.6.1：デシル分析

デシル	購入金額合計	構成比	構成比累計	1人あたり購入金額
デシル1	50,200	17.9%	17.9%	16,733
デシル2	43,700	15.6%	33.5%	14,567
デシル3	41,600	14.8%	48.3%	13,867
デシル4	38,500	13.7%	62.0%	12,833
デシル5	34,300	12.2%	74.2%	11,433
デシル6	28,200	10.0%	84.3%	9,400
デシル7	20,400	7.3%	91.5%	6,800
デシル8	11,800	4.2%	95.7%	3,933
デシル9	7,800	2.8%	98.5%	2,600
デシル10	4,200	1.5%	100.0%	1,400

最初にユーザーを購入金額が多い順に並べ替え、並び替えたユーザーの上位から10%ずつデシル1からデシル10までのグループに割り当てます。等間隔にデータをグルーピングするには、NTILEウィンドウ関数を使用します。

コード5.1.6.1：購入額の多い順にユーザーグループを10分割するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_purchase_amount AS (
  SELECT
      user_id
    , SUM(amount) AS purchase_amount
  FROM
    action_log
  WHERE
    action = 'purchase'
  GROUP BY
    user_id
)
, users_with_decile AS (
```

```
  SELECT
      user_id
    , purchase_amount
    , ntile(10) OVER (ORDER BY purchase_amount DESC) AS decile
  FROM
    user_purchase_amount
)
SELECT *
FROM users_with_decile
;
```

```
 user_id | purchase_amount | decile
---------+-----------------+--------
 U019    |           17800 |      1
 U006    |           16700 |      1
 U013    |           15700 |      1
 U021    |           14700 |      2
 U029    |           14600 |      2
 U004    |           14400 |      2
...
 U005    |            3200 |      9
 U012    |            2500 |      9
 U002    |            2100 |      9
 U016    |            2100 |     10
 U023    |            2000 |     10
 U018    |             100 |     10
```

次のステップ 続いて、各グループの合計・平均購入金額と、累計の購入金額、全体の購入金額などの集約を計算します。GROUP BYでデシルごとに集約し、集約関数とウィンドウ関数を組み合わせて、一度にこれらの値を計算できます。

コード5.1.6.2：10分割したデシルごとの集約を行うクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_purchase_amount AS (
  -- コード5.1.6.1参照
)
, users_with_decile AS (
  -- コード5.1.6.1参照
)
, decile_with_purchase_amount AS (
  SELECT
    decile
    , SUM(purchase_amount) AS amount
    , AVG(purchase_amount) AS avg_amount
    , SUM(SUM(purchase_amount)) OVER (ORDER BY decile) AS cumulative_amount
```

```
    , SUM(SUM(purchase_amount)) OVER () AS total_amount
  FROM
    users_with_decile
  GROUP BY
    decile
)
SELECT *
FROM
  decile_with_purchase_amount
;
```

```
 decile | amount | avg_amount | cumulative_amount | total_amount
--------+--------+------------+-------------------+--------------
      1 |  50200 |   16733.33 |             50200 |       280700
      2 |  43700 |   14566.66 |             93900 |       280700
      3 |  41600 |   13866.66 |            135500 |       280700
      4 |  38500 |   12833.33 |            174000 |       280700
      5 |  34300 |   11433.33 |            208300 |       280700
      6 |  28200 |    9400.00 |            236500 |       280700
      7 |  20400 |    6800.00 |            256900 |       280700
      8 |  11800 |    3933.33 |            268700 |       280700
      9 |   7800 |    2600.00 |            276500 |       280700
     10 |   4200 |    1400.00 |            280700 |       280700
```

次のステップ 最後に、構成比と構成比累計を計算します。

コード5.1.6.3：購入額の多いデシル順に構成比と構成比累計を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_purchase_amount AS (
  -- コード5.1.6.1参照
)
, users_with_decile AS (
  -- コード5.1.6.1参照
)
, decile_with_purchase_amount AS (
  -- コード5.1.6.2参照
)
SELECT
    decile
  , amount
  , avg_amount
  , 100.0 * amount / total_amount AS total_ratio
  , 100.0 * cumulative_amount / total_amount AS cumulative_ratio
FROM
  decile_with_purchase_amount;
```

```
 decile | amount | avg_amount | total_ratio | cumulative_ratio
--------+--------+------------+-------------+------------------
      1 |  50200 |   16733.33 |       17.88 |            17.88
      2 |  43700 |   14566.66 |       15.56 |            33.45
      3 |  41600 |   13866.66 |       14.82 |            48.27
      4 |  38500 |   12833.33 |       13.71 |            61.98
      5 |  34300 |   11433.33 |       12.21 |            74.20
      6 |  28200 |    9400.00 |       10.04 |            84.25
      7 |  20400 |    6800.00 |        7.26 |            91.52
      8 |  11800 |    3933.33 |        4.20 |            95.72
      9 |   7800 |    2600.00 |        2.77 |            98.50
     10 |   4200 |    1400.00 |        1.49 |           100.00
```

ワンポイント

本項で紹介したデシル分析を行った上で、各デシルの特徴を他の分析手法で細かく調査して傾向を見出すと、より詳細なユーザー像を把握できます。

例えば、デシル7～10はまだ定着していないカスタマーなので、メルマガでリテンションを高めるなどの施策で成長を促すなど、高い精度の施策を計画立案する際に役立ちます。また、既存のメルマガ配信ツールと絡めて、デシル7～10に対して送信することが可能になると、よりデータが活用されていくことになるでしょう。なお、他のシステムとの連携では集計済みのデータではなく、コード5.1.6.1の状態を利用することになります。

関連項目 | 3-2-2 複数の値を比較する

5-1-7 RFM分析でユーザーを3つの視点でグループに分ける

SQL CASE式、generate_series関数　　分析 RFM分析

前項のデシル分析では、ユーザーの合計購入金額を10のグループに分ける方法を紹介しました。

しかし、デシル分析ではデータの検索対象が長期間になると、過去は優良顧客としてサービスを利用していたが、現在は他のサービスを利用中で休眠しているユーザーも含まれてしまいます。また、逆に検索対象が短期間では、得られるデータが少なかったり、定期的に購入する安定顧客が含まれない、該当期間で一時的に大きな買い物をしたユーザーが優良顧客と扱われてしまいます。

本項で紹介するRFM分析は、デシル分析よりも詳細にユーザーをグループ分けすることが可能です。

RFM分析の3つの指標を集計する

RFM分析では、下記の3つの指標を元にユーザーをグループ化します。

Recency：最新購入日
最近購入に至ったユーザーほど優良顧客として扱う。

Frequency：購入回数
ユーザーが購入した回数をカウントし、回数が多いユーザーほど優良顧客として扱う。

Monetary：購入金額合計
ユーザーの購入金額の合計を集計し、金額が多いユーザーほど優良顧客として扱う。

上記のRecencyとFrequency、Monetaryの3つの頭文字を取って、RFM分析と呼びます。デシル分析では1回の購入で高額な買い物をしたユーザーと、定期的に高額ではない商品を購入しているユーザーが同一グループと判定されることがありますが、RFM分析では上記の3視点からグループを分けるため、同じグループに所属することはありません。

まずは、下記のコード例に、ユーザーごとにRFMを集計するクエリを示します。

コード5.1.7.1: ユーザー毎にRFMを集計するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
purchase_log AS (
  SELECT
      user_id
    , amount
    -- タイムスタンプから日付を抽出
    -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、substringで日付部分を抽出する
    , substring(stamp, 1, 10) AS dt
    -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、substrが利用できる
    -- , substr(stamp, 1, 10) AS dt
  FROM
    action_log
  WHERE
    action = 'purchase'
)
, user_rfm AS (
  SELECT
      user_id
    , MAX(dt) AS recent_date
    -- ■ PostgreSQL, Redshiftの場合、日付型同士の引き算ができる
    , CURRENT_DATE - MAX(dt::date) AS recency
    -- ■ BigQueryの場合、date_diff関数が利用できる
```

```
    -- , date_diff(CURRENT_DATE, date(timestamp(MAX(dt))), day) AS recency
    -- ■ Hive, SparkSQLの場合.datediff関数が利用できる
    -- , datediff(CURRENT_DATE(), to_date(MAX(dt))) AS recency
    , COUNT(dt) AS frequency
    , SUM(amount) AS monetary
  FROM
    purchase_log
  GROUP BY
    user_id
)
SELECT *
FROM
  user_rfm
;
```

```
 user_id | recent_date | recency | frequency | monetary
---------+-------------+---------+-----------+----------
 U040    | 2016-12-03  |       2 |         7 |    22340
 U140    | 2016-11-23  |      12 |        11 |    27640
 U128    | 2016-11-23  |      12 |         6 |    18550
 U203    | 2016-11-06  |      29 |         3 |    10430
 U058    | 2016-12-03  |       2 |         2 |     2970
 U144    | 2016-10-31  |      35 |         5 |    13360
 U213    | 2016-11-29  |       6 |         4 |    15300
 U014    | 2016-11-03  |      32 |         6 |    17160
 U154    | 2016-10-25  |      41 |         6 |    15230
 U120    | 2016-10-31  |      35 |         4 |    10020
 U251    | 2016-10-06  |      60 |         2 |     6580
...
```

RFMランクを定義する

RFM分析では、3つの指標をそれぞれ5つのグループに分けるのが一般的です。したがって、125（＝5×5×5）のグループでユーザーを捉えることができます。

上記コード例の出力結果でデータの分布を見ながら、各項目を定義付けします。本項では、次表に示す通りに、RFMランクを定義します。

表5.1.7.1：RFMランクの定義表

ランク	R：最新購入日	F：累計購入回数	M：累計購入金額
5	14日以内	20回以上	30万円以上
4	28日以内	10回以上	10万円以上
3	60日以内	5回以上	3万円以上
2	90日以内	2回以上	5000円以上
1	91日以上	1回のみ	5000円未満

コード5.1.7.2：ユーザー毎のRFMランクを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_rfm AS (
  -- コード5.1.7.1参照
)
, user_rfm_rank AS (
  SELECT
      user_id
    , recent_date
    , recency
    , frequency
    , monetary
    , CASE
        WHEN recency < 14 THEN 5
        WHEN recency < 28 THEN 4
        WHEN recency < 60 THEN 3
        WHEN recency < 90 THEN 2
        ELSE 1
      END AS r
    , CASE
        WHEN 20 <= frequency THEN 5
        WHEN 10 <= frequency THEN 4
        WHEN  5 <= frequency THEN 3
        WHEN  2 <= frequency THEN 2
        WHEN  1  = frequency THEN 1
      END AS f
    , CASE
        WHEN 300000 <= monetary THEN 5
        WHEN 100000 <= monetary THEN 4
        WHEN  30000 <= monetary THEN 3
        WHEN   5000 <= monetary THEN 2
        ELSE 1
      END AS m
  FROM
    user_rfm
)
SELECT *
FROM
  user_rfm_rank
;
```

```
 user_id | recent_date | recency | frequency | monetary | r | f | m
---------+-------------+---------+-----------+----------+---+---+---
 U040    | 2016-12-03  |       2 |         7 |    22340 | 5 | 3 | 2
 U140    | 2016-11-23  |      12 |        11 |    27640 | 5 | 4 | 2
 U128    | 2016-11-23  |      12 |         6 |    18550 | 5 | 3 | 2
 U203    | 2016-11-06  |      29 |         3 |    10430 | 3 | 2 | 2
 U058    | 2016-12-03  |       2 |         2 |     2970 | 5 | 2 | 1
 U144    | 2016-10-31  |      35 |         5 |    13360 | 3 | 3 | 2
```

```
 U213    | 2016-11-29  |       6 |         4 |    15300 | 5 | 2 | 2
 U014    | 2016-11-03  |      32 |         6 |    17160 | 3 | 3 | 2
 U154    | 2016-10-25  |      41 |         6 |    15230 | 3 | 3 | 2
 U120    | 2016-10-31  |      35 |         4 |    10020 | 3 | 2 | 2
 U251    | 2016-10-06  |      60 |         2 |     6580 | 2 | 2 | 2
 U043    | 2016-11-22  |      13 |         2 |     3200 | 5 | 2 | 1
 U129    | 2016-09-21  |      75 |         2 |     8830 | 2 | 2 | 2
 U057    | 2016-11-22  |      13 |         6 |    22480 | 5 | 3 | 2
 U085    | 2016-11-13  |      22 |         6 |    15660 | 4 | 3 | 2
 U290    | 2016-09-20  |      76 |         2 |     8490 | 2 | 2 | 2
...
```

上記の出力結果を元に、前表「RFMランクの定義表」における各グループの人数を確認するクエリを下記のコード例に示します。

コード5.1.7.3：各グループの人数を確認するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

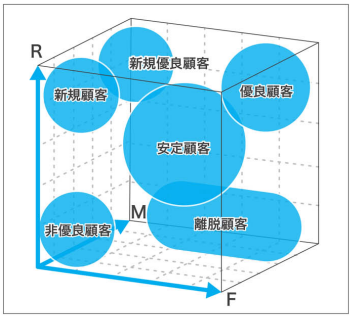
```
WITH
user_rfm AS (
  -- コード5.1.7.1参照
)
, user_rfm_rank AS (
  -- コード5.1.7.2参照
)
, mst_rfm_index AS (
  -- 1から5までの連番テーブルを作成する
  -- PostgreSQLのgenerate_series関数等でも代用可能
          SELECT 1 AS rfm_index
  UNION ALL SELECT 2 AS rfm_index
  UNION ALL SELECT 3 AS rfm_index
  UNION ALL SELECT 4 AS rfm_index
  UNION ALL SELECT 5 AS rfm_index
)
, rfm_flag AS (
  SELECT
      m.rfm_index
    , CASE WHEN m.rfm_index = r.r THEN 1 ELSE 0 END AS r_flag
    , CASE WHEN m.rfm_index = r.f THEN 1 ELSE 0 END AS f_flag
    , CASE WHEN m.rfm_index = r.m THEN 1 ELSE 0 END AS m_flag
  FROM
      mst_rfm_index AS m
    CROSS JOIN
      user_rfm_rank AS r
)
SELECT
    rfm_index
  , SUM(r_flag) AS r
  , SUM(f_flag) AS f
```

```
  , SUM(m_flag) AS m
FROM
  rfm_flag
GROUP BY
  rfm_index
ORDER BY
  rfm_index DESC
;
```

```
 rfm_index |  r  |  f  |  m
-----------+-----+-----+-----
         5 | 100 |   1 |   2
         4 |  77 |  13 |   2
         3 |  84 | 159 |   6
         2 |  23 | 108 | 259
         1 |  12 |  15 |  27
```

極端に少ないユーザー数のグループができた場合は、RFMランクの定義を修正しましょう。この結果を3次元で表現したものが下図です。各ユーザーのRFMランク別に、どのような施策を実施するか検討します。

図5.1.7.1：RFM分析のイメージ



ただし、本項のRFMランクの定義表に従うと、125のグループが作成されます。これを管理するのは大変なので、さらに少ないグループに分けて管理する方法を紹介します。

1次元でユーザーを捉える

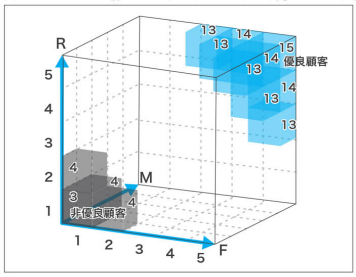
RFM分析を3次元で表現すると、125のグループで管理が大変です。本項では、RFMの各ランクを合計したものを総合ランクとして、13のグループに分けて管理する方法を紹介します。

下表に示す通り、R+F+Mの値を総合ランクとして計算するクエリを、下記のコード例に示します。

表5.1.7.2：総合ランク別のユーザー数

総合ランク	R	F	M	ユーザー数
15	5	5	5	9
14	5	5	4	13
	5	4	5	15
	4	5	5	16
13	5	5	3	26
	5	4	4	31
…	…	…	…	…
3	1	1	1	842

図5.1.7.2：RFM分析において、1次元でユーザーを捉えるイメージ図



コード5.1.7.4：総合ランクを算出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_rfm AS (
  -- コード5.1.7.1参照
)
, user_rfm_rank AS (
  -- コード5.1.7.2参照
)
SELECT
    r + f + m AS total_rank
  , r , f , m
  , COUNT(user_id)
FROM
  user_rfm_rank
GROUP BY
  r, f, m
ORDER BY
  total_rank DESC, r DESC, f DESC, m DESC;
```

```
 total_rank | r | f | m | count
------------+---+---+---+-------
         15 | 5 | 5 | 5 |     1
         14 | 5 | 5 | 4 |     1
         14 | 5 | 4 | 5 |     1
         12 | 5 | 4 | 3 |     4
         11 | 5 | 4 | 2 |     3
         11 | 4 | 4 | 3 |     1
         10 | 5 | 3 | 2 |    69
         10 | 4 | 4 | 2 |     2
...
```

上記コード例で計算した総合ランク別で集計し、各総合ランクのユーザー数を確認します。コード例を下記に示します。

コード5.1.7.5：総合ランク別にユーザー数を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_rfm AS (
  -- コード5.1.7.1参照
)
, user_rfm_rank AS (
  -- コード5.1.7.2参照
)
SELECT
    r + f + m AS total_rank
  , COUNT(user_id)
FROM
  user_rfm_rank
GROUP BY
  -- ■ PostgreSQL, Redshift, BigQueryの場合、
  --   SELECT句で定義した別名をGROUP BYに指定できる
   total_rank
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、
  --   SELECT句で別名を指定する前の式をGROUP BYに指定できる
  -- r + f + m
ORDER BY
  total_rank DESC;
```

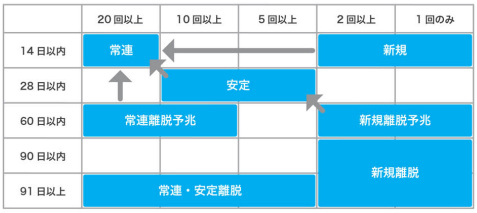
```
 total_rank | count
------------+-------
         15 |     1
         14 |     2
         12 |     4
         11 |     4
         10 |    72
          9 |    69
```

```
8 |   65
7 |   41
6 |   21
5 |    5
4 |    6
3 |    7
```

2次元でユーザーを捉える

RFM指標の2つを利用してユーザー層を定義する方法もあります。下図に示す通り、各ユーザー層に対してどのようなマーケティング施策を実施するのか、各ユーザー層をより上位のユーザー層へどう育てていくのかを考えることができます。本項では例として「R」と「F」を用いて集計します。

図5.1.7.3：RとFを用いて、2次元でユーザーを捉える



上図の各セルにどの程度のユーザーがいるか算出するクエリを、下記のコード例に示します。ユーザー数を集計したら、高いランクへとユーザーを育てるにはどのような施策が適切なのか検討しましょう。

コード5.1.7.6：RとFを用いた2次元でユーザー数を集計するクエリ

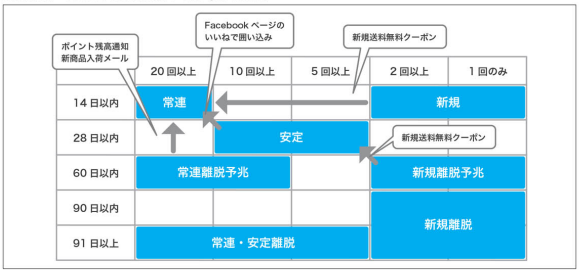
PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
user_rfm AS (
  -- コード5.1.7.1参照
)
, user_rfm_rank AS (
  -- コード5.1.7.2参照
)
SELECT
    CONCAT('r_', r) AS r_rank
  -- ■ BigQueryの場合、CONCAT関数の引数を明示的にstringに変換する
  -- CONCAT('r_', CAST(r AS string)) AS r_rank
```

```
  , COUNT(CASE WHEN f = 5 THEN 1 END) AS f_5
  , COUNT(CASE WHEN f = 4 THEN 1 END) AS f_4
  , COUNT(CASE WHEN f = 3 THEN 1 END) AS f_3
  , COUNT(CASE WHEN f = 2 THEN 1 END) AS f_2
  , COUNT(CASE WHEN f = 1 THEN 1 END) AS f_1
FROM
  user_rfm_rank
GROUP BY
  r
ORDER BY
  r_rank DESC;
```

```
 r_rank | f_5 | f_4 | f_3 | f_2 | f_1
--------+-----+-----+-----+-----+-----
 r_5    |   1 |   8 |  69 |  21 |   1
 r_4    |   0 |   3 |  48 |  25 |   1
 r_3    |   0 |   2 |  39 |  39 |   4
 r_2    |   0 |   0 |   3 |  18 |   2
 r_1    |   0 |   0 |   0 |   5 |   7
```

図 5.1.7.4：どのような施策を打つのか考えてみよう



まとめ

RFM分析の各指標、Recency、Frequency、Monetaryによってユーザーの属性を定義し、1次元、2次元、3次元で表現する方法を身に付けましょう。サービス改善の検討、ユーザー属性別にメルマガの文面を最適化するなど、さまざまな用途で活用しましょう。

関連項目 | 3-1-1 コード値をラベルに置き換える

5-2

ユーザー全体の時系列による状態変化を見付ける

ユーザーのサービス利用開始から日時が経過すると、ロイヤリティが高いユーザーに成長する、あるいは利用を中止して退会、登録は残っているものの再び利用することがない状態（休眠）になるなど、ユーザーの状態に変化が生じます。

- ユーザーが継続して利用する（リピート）
- ユーザーが利用を中止する（退会・休眠）

サービスを運営する側としては、継続して利用してもらうことを願っています。そのためには、まずどの程度のユーザーが継続して利用しているのかを把握して、目標に対する乖離をどのように埋めるか検討する必要があります。また、しばらくサービスの利用がなかったり、休眠の傾向があるユーザーに対しては、継続してもらうための工夫を常に考えたいと思っているはずです。

退会ユーザーに対する施策は実施が困難ですが、休眠ユーザーに対しては、メルマガやCM、広告などの施策で、利用を再開してもらいたいところです。

本節では、ユーザーのサービス利用に対する時系列の変化を数値化し可視化することで、現状の把握、施策の効果、今後の計画に役立つ糸口となるレポートと集計するSQLを紹介します。

サンプルデータ

本節では、下記に示す2つのテーブル、ユーザーマスタとアクションログを元に解説します。SNSに代表されるコミュニティサービスを想定してください。

データ5.2.0.1:ユーザーマスタ(mst_users)

```
 user_id | sex | birth_date | register_date | register_device | withdraw_date
---------+-----+------------+---------------+-----------------+---------------
 U001    | M   | 1977-06-17 | 2016-10-01    | pc              |
 U002    | F   | 1953-06-12 | 2016-10-01    | sp              | 2016-11-10
```

データ5.2.0.2:アクションログ(action_log)

```
 session | user_id |  action  |        stamp
---------+---------+----------+---------------------
 037f2   | U001    | view     | 2016-11-03 18:00:00
 037f2   | U001    | post     | 2016-11-03 18:03:12
 037f2   | U001    | view     | 2016-11-03 18:09:44
 037f2   | U001    | follow   | 2016-11-03 18:12:15
 037f2   | U001    | view     | 2016-11-03 18:15:31
 037f2   | U001    | like     | 2016-11-03 18:16:58
 037f2   | U001    | like     | 2016-11-03 18:17:09
 037f2   | U001    | comment  | 2016-11-03 18:21:37
 c07d1   | U002    | view     | 2016-11-03 18:22:03
 c07d1   | U002    | comment  | 2016-11-03 18:23:59
 c07d1   | U002    | follow   | 2016-11-03 18:25:20
 a9338   | U001    | view     | 2016-11-04 09:31:51
 a9338   | U001    | like     | 2016-11-04 09:33:01
 6ceab   | U002    | view     | 2016-11-04 11:27:30
 6ceab   | U002    | comment  | 2016-11-04 11:29:13
```

5-2-1 登録数の推移と傾向を見る

SQL COUNT(DISTINCT～)　　分析 登録数、昨対比

ユーザー登録が必要なサービスで、登録数は重要な指標の１つです。登録者が減少傾向にあれば、サービスをより活性化させることは難しく、登録者が増加傾向にある場合は、登録ユーザーを囲い込めているかを分析して、サービスの活性化に繋げていくことが重要です。ユーザーを囲い込めているかを分析する前に、まず現状の登録数を把握するSQLを紹介します。

日別登録数の推移

表5.2.1.1：日別登録数の推移

日付	登録数
2016/1/1	417
2016/1/2	424
2016/1/3	413

ユーザー数の集計は、ユーザーを一意に識別するID（ユーザーID）で重複を省きます（DISTINCT）。会員テーブル（mst_users）では、ユーザーIDが重複して登録されることはシステム上ないと考えられますが、本書では登録数も登録したユーザー数として扱うため、定義上重複を省いて集計します。

下記コード例に、ユーザー登録数を日別に集計するSQLを示します。

コード5.2.1.1：日次で登録者数の推移を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    register_date
  , COUNT(DISTINCT user_id) AS register_count
FROM
  mst_users
GROUP BY
  register_date
ORDER BY
  register_date
;
```

```
 register_date | register_count
---------------+----------------
 2016-10-01    |            423
 2016-10-02    |            465
 2016-10-03    |            434
 2016-10-04    |            421
 2016-10-05    |            416
 2016-10-06    |            474
 2016-10-07    |            436
 2016-10-08    |            399
 2016-10-09    |            423
 2016-10-10    |            413
 2016-10-11    |            384
...
 2016-11-28    |            446
 2016-11-29    |            456
 2016-11-30    |            471
```

上記のコード例を登録数集計の基本形として、以降いくつかの切り口で登録数を把握してみましょう。

月別登録数の推移

月別に登録者数を集計し、本節までに紹介したSQLを用いて、下表に示す各月と比較する先月比を集計します。まず、月別を集計するため、日付型のデータから年と月のデータだけを抽出し、year_monthカラムに格納します。このyear_monthごとに集約し登録数を計算します。先月の登録数と比較するためには、LAGウィンドウ関数を使用します。

表5.2.1.2：月別登録者数と先月比

年月	登録数	先月比
2016年1月	12,935	
2016年2月	13,169	101.8%
2016年3月	12,831	97.4%

コード5.2.1.2：各月の登録数と先月比を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_users_with_year_month AS (
  SELECT
      *
    -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、substringで年-月部分を抽出する
    , substring(register_date, 1, 7) AS year_month
    -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、substrが利用できる
    -- , substr(register_date, 1, 7) AS year_month
  FROM
    mst_users
)
SELECT
    year_month
  , COUNT(DISTINCT user_id) AS register_count
  , LAG(COUNT(DISTINCT user_id)) OVER(ORDER BY year_month)
  -- ■ SparkSQLの場合は下記を使用
  -- , LAG(COUNT(DISTINCT user_id))
  --     OVER(ORDER BY year_month ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
    AS last_month_count
  , 1.0
    * COUNT(DISTINCT user_id)
    / LAG(COUNT(DISTINCT user_id)) OVER(ORDER BY year_month)
  -- ■ SparkSQLの場合は下記を使用
  -- / LAG(COUNT(DISTINCT user_id))
  --     OVER(ORDER BY year_month ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
    AS month_over_month_ratio
FROM
  mst_users_with_year_month
GROUP BY
  year_month
;
```

```
 year_month | register_count | last_month_count | month_over_month_ratio
------------+----------------+------------------+------------------------
 2016-10    |          12895 |                  |
 2016-11    |          11436 |            12895 |                 0.8868
```

登録デバイス別の推移

月別登録者の集計後は、登録したレコード内に保持されている情報を利用して、その内訳を集計します。本項では登録したデバイスが何であるかを示す、mst_usersのdeviceカラムを使用して、登録数の内訳を集計します。

登録デバイスと同様に、登録と同時に付加されている情報が他にもあれば、下記コード例のdeviceを該当するカラムに変更することで、その情報に関する内訳を集計できます（例:性別や都道府県など）。

なお、以降の項では、登録ユーザーの行動を詳細に探るための切り口を紹介します。

表5.2.1.3：デバイス別登録数

年月	登録数	PC登録数	SP登録数	アプリ登録数
2016年1月	12,935	3049	4,382	5,504
2016年2月	13,169	3,593	4,492	5,084
2016年3月	12,831	4,108	4,792	3,931

コード5.2.1.3：デバイスごとの登録数を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_users_with_year_month AS (
    -- コード5.2.1.2参照
)
SELECT
    year_month
  , COUNT(DISTINCT user_id) AS register_count
  , COUNT(DISTINCT CASE WHEN register_device = 'pc'  THEN user_id END) AS register_pc
  , COUNT(DISTINCT CASE WHEN register_device = 'sp'  THEN user_id END) AS register_sp
  , COUNT(DISTINCT CASE WHEN register_device = 'app' THEN user_id END) AS register_app
FROM
  mst_users_with_year_month
GROUP BY
  year_month
;
```

```
 year_month | register_count | register_pc | register_sp | register_app
------------+----------------+-------------+-------------+--------------
 2016-10    |          12895 |        3485 |        6439 |         2971
 2016-11    |          11436 |        2962 |        6712 |         1762
```

ワンポイント

登録したデバイスでユーザー行動が異なることがあります。後述する分析を登録デバイス別に実施してみるのもよいでしょう。また、どの程度のユーザーが登録デバイス以外で利用しているかも分析してみましょう（マルチデバイスユーザー）。

関連項目	3-3-2 グループの中での順序を扱う 4-1-4 月別売上の昨対比を求める

5-2-2 継続率と定着率を算出する

SQL COUNT(DISTINCT～)　　分析 継続率、定着率

ユーザーが登録してもサービスを継続的に利用しなければ、いくらユーザーを獲得してもサービスの活性化には繋がりません。ユーザーが継続的に利用しないと、Twitterに代表されるSNSであれば、ユーザーをフォローしてもフィードには何も流れなくなり、ソーシャルゲームであれば、競争相手がいなくなり、継続意欲が薄れていきます。登録から短期間でユーザーが継続して利用しているかを調査する際に、継続率と定着率を用いることで傾向を調べることができます。

本項では、継続率と定着率と呼ばれる指標を下記の通り定義し、ユーザーが登録してから継続的に利用しているかを把握する手法を紹介します。

継続率・定着率の定義

継続率と定着率はその名称は類似していますが、計算方法は若干異なります。各指標の用途と計算方法をまず整理しましょう。

継続率：登録日を基準として、その後の指定日においてユーザーのサービス利用を示す指標

下表（左）に示す通り、6月12日の登録ユーザーが、その翌日もサービスを利用していたら翌日継続者、2日後に利用していたら2日継続者として扱います。また、下表（右）に示す通り、翌日に利用していなくとも、毎日利用していなくとも、その判定日付に利用していたら継続者として扱います。

表5.2.2.1：継続率の判定日付

指標名	6/12	6/13	6/14	6/15
翌日継続	●	○		
2日継続	●		○	
3日継続	●			○

●登録日　○利用日

表5.2.2.2：いずれも3日継続者として扱う

指標名	6/12	6/13	6/14	6/15
3日継続	●	○	○	○
3日継続	●		○	○
3日継続	●			○

●登録日　○利用日

継続率の算出方法は、下表に示す登録者数と利用者数を集計し、「利用者数」÷「登録数」で求めます。6月12日の翌日継続率は80%、2日継続率は60%となります。

表5.2.2.3：日次の登録数と、n日継続者と継続率

登録日	登録数	6月13日 利用者数	6月14日 利用者数	6月15日 利用者数	翌日 継続率	2日 継続率	3日 継続率
6月12日	100	80	60	40	80%	60%	40%
6月13日	200	-	150	100	75%	50%	-
6月14日	150	-	-	75	50%	-	-

定着率：登録日を基準として、その後の特定の7日間において、ユーザーのサービス利用を示す指標

定着は継続とは異なり、7日間の期間に一度でもサービス利用をしたら定着者として扱います。下図に定着率の判定期間を示します。

図 5.2.2.1：定着率の判定期間

7日定着率では、登録翌日から7日目までを定着率を算出する判定期間とします。14日定着率以降は基準日から過去7日間を判定期間とします。例えば、14日定着率は、登録翌日から8日目～14日目が判定期間です。また、7日間のサービス利用日が1日でも3日でも、定着数は1として扱います。

定着率の算出方法は、下表に示す登録者数と利用者数を集計し、「利用者数」÷「登録数」で求めます。例えば、6月12日の7日定着率は80%、14日定着率は60%となります。

表 5.2.2.5：日次の登録数と、n日定着者と定着率

登録日	登録数				
6月12日	100	6/13～6/19 利用者数	66/20～6/26 利用者数	7日 定着率	14日 定着率
		80	60	80%	60%
6月13日	200	6/14～6/20 利用者数	6/21～6/27 利用者数	7日 定着率	14日 定着率
		150	100	75%	50%

継続率と定着率の使い分け

ユーザーに期待する利用サイクルは、サービスによって異なります。

例えば、ソーシャルゲームやニュースサイトなどでは隙間時間で毎日の利用を期待したいものです。ECサイトやクチコミ投稿サイトなどでは、毎日買い物したり、クチコミを投稿したくなる体験を毎日しているユーザーは少なく、何かの目的が発生したタイミング、体験したタイミングで利用するものと考えます。

サービスの目的や用途に応じて、どちらをより重要視するかを検討した上で、定期的に確認してみましょう。

表5.2.2.6：継続率と定着率を使い分けるポイント

指標	用途
継続率	ユーザーが習慣として、毎日利用してもらいたいサービス 例：ニュースサイト、ソーシャルゲーム、SNSなど
定着率	ユーザーに何らかの目的が発生した際に利用してもらいたいサービス 例：ECサイト、クチコミサイト、Q&Aサイト、写真投稿サイトなど

継続率に関するレポート

継続率に関するレポートを作成するためのSQLを紹介します。

日次のn日継続率推移

継続率を向上させる施策の効果が意図通り反映されているかを確認するため、継続率を日次で集計するレポートを作成します。登録の翌日ですらサービスを利用してもらえないと、以降習慣として利用してもらうことは難しいでしょう。サービスをより活発にするには、まずは翌日継続率を向上させることを考えましょう。

表5.2.2.7：日次の登録数とn日継続率

日付	登録数	翌日 継続率	2日 継続率	3日 継続率	…	7日 継続率
2016/6/1	1,632	74.2%	62.4%	47.3%	…	12.1%
2016/6/2	1,579	75.9%	62.1%	49.8%	…	13.0%
2016/6/3	1,601	73.6%	63.1%	51.0%	…	13.2%

上記のレポートを作成するには、まずは「翌日継続率」の計算方法を考えます。翌日継続率の定義は、指定日に登録したユーザーの中で、その翌日にサービスを利用したユーザーの割合です。各ユーザー数をカウントして、割り算で割合を計算しても構いませんが、指定日の翌日に利用したユーザーに1、利用しなかったユーザーに0のフラグを割り振り、最後にAVG関数で平均を取る方法が簡単です。

ここで注意すべきことがあります。翌日継続率を計算するには、翌日のログデータが既に取得済みであることが必須です。登録日翌日にアクションしたユーザーが本当に0人であるケースと、ログがまだ収集できず結果的に0件になるケースは、厳密に区別する必要があります。本項では、ログの集計期間の最新日を取得し、その最新日を越える期間の継続率はNULLとしています。

下記コード例に、翌日継続率計算の準備として、「ログの最新日付」と「ユーザーごとの登録日の翌日の日付」を計算するクエリを示します。

コード5.2.2.1：「ログの最新日付」と「ユーザーごとの登録日の翌日の日付」を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_log_with_mst_users AS (
  SELECT
      u.user_id
    , u.register_date
    -- アクションの日付と、ログ全体の最新日付を日付型として取得
    , CAST(a.stamp AS date) AS action_date
    , MAX(CAST(a.stamp AS date)) OVER() AS latest_date
    -- ■ BigQueryの場合、一度タイムスタンプ型に変換してから日付型に変換
    -- , date(timestamp(a.stamp)) AS action_date
    -- , MAX(date(timestamp(a.stamp))) OVER() AS latest_date

    -- 登録日の1日後の日付を計算する
    -- ■ PostgreSQLの場合は下記を使用
    , CAST(u.register_date::date + '1 day'::interval AS date)
    -- ■ Redshiftの場合は下記を使用
    -- , dateadd(day, 1, u.register_date::date)
    -- ■ BigQueryの場合は下記を使用
    -- , date_add(CAST(u.register_date AS date), interval 1 day)
    -- ■ Hive, SparkSQLの場合は下記を使用
    -- , date_add(CAST(u.register_date AS date), 1)
      AS next_day_1
  FROM
      mst_users AS u
    LEFT OUTER JOIN
      action_log AS a
    ON u.user_id = a.user_id
)
SELECT *
FROM
  action_log_with_mst_users
ORDER BY
  register_date
;
```

```
 user_id | register_date | action_date | latest_date | next_day_1
---------+---------------+-------------+-------------+------------
 U001    | 2016-10-01    | 2016-10-01  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-10-01  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-10-02  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-10-02  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-10-03  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-10-03  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-10-07  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-10-07  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-11-01  | 2016-11-28  | 2016-10-02
 U001    | 2016-10-01    | 2016-11-01  | 2016-11-28  | 2016-10-02
```

```
...
 U023    | 2016-11-04    | 2016-11-04  | 2016-11-28  | 2016-11-05
 U024    | 2016-11-05    | 2016-11-05  | 2016-11-28  | 2016-11-06
 U025    | 2016-11-05    | 2016-11-05  | 2016-11-28  | 2016-11-06
 U026    | 2016-11-10    | 2016-11-10  | 2016-11-28  | 2016-11-11
 U027    | 2016-11-10    | 2016-11-10  | 2016-11-28  | 2016-11-11
 U028    | 2016-11-15    | 2016-11-15  | 2016-11-28  | 2016-11-16
 U029    | 2016-11-28    | 2016-11-28  | 2016-11-28  | 2016-11-29
 U030    | 2016-11-28    | 2016-11-28  | 2016-11-28  | 2016-11-29
```

次のステップ 続いて、指定日の翌日にアクションしたかどうかのフラグを0と1で表現します。ただし、「指定日の翌日」が、ログの最新日付を超えている場合はフラグをNULLとします。

コード5.2.2.2：ユーザーのアクションフラグを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_log_with_mst_users AS (
  -- コード5.2.2.1参照
)
, user_action_flag AS (
  SELECT
      user_id
    , register_date
      -- 4. 登録日の1日後にアクションをしたかどうかを0,1のフラグで表す
    , SIGN(
        -- 3. 登録日の1日後にアクションをした回数をユーザごとに合計する
        SUM(
          -- 1. 登録日の1日後が、ログの最新日付以前に含まれるか確認
          CASE WHEN next_day_1 <= latest_date THEN
            -- 2. 登録日の1日後の日付にアクションをしている場合は1、それ以外は0を返す
            CASE WHEN next_day_1 = action_date THEN 1 ELSE 0 END
          END
        )
      ) AS next_1_day_action
  FROM
    action_log_with_mst_users
  GROUP BY
    user_id, register_date
)
SELECT *
FROM
  user_action_flag
ORDER BY
  register_date, user_id
;
```

```
 user_id | register_date | next_1_day_action
---------+---------------+-------------------
 U001    | 2016-10-01    |                 1
 U002    | 2016-10-01    |                 1
 U003    | 2016-10-01    |                 1
 U004    | 2016-10-05    |                 0
 U005    | 2016-10-05    |                 0
 U006    | 2016-10-10    |                 0
...
 U027    | 2016-11-10    |                 0
 U028    | 2016-11-15    |                 0
 U029    | 2016-11-30    |
 U030    | 2016-11-30    |
...
```

本項で使用したログの最新日付は「2016-11-30」であるため、登録日が「2016-11-30」のユーザーに関しては、翌日のアクションフラグはNULLとなります。

ユーザーのアクションフラグが0と1で表現できれば、フラグの値に100.0を掛けてAVG関数で平均を取ることで、パーセント表記の割合を計算できます。

コード5.2.2.3:翌日継続率を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_log_with_mst_users AS (
  -- コード5.2.2.1参照
)
, user_action_flag AS (
  -- コード5.2.2.2参照
)
SELECT
    register_date
  , AVG(100.0 * next_1_day_action) AS repeat_rate_1_day
FROM
  user_action_flag
GROUP BY
  register_date
ORDER BY
  register_date
;
```

```
 register_date | repeat_rate_1_day
---------------+-------------------
 2016-10-01    |           43.2354
 2016-10-02    |           41.3456
 2016-10-03    |           36.3289
 2016-10-04    |           38.9874
```

```
 2016-10-05    |           40.8762
 2016-10-06    |           39.7642
 2016-10-07    |           37.9614
 2016-10-08    |           38.9691
 2016-10-09    |           40.7982
 2016-10-10    |           41.9623
 2016-10-11    |           42.8791
...
 2016-11-28    |           37.2457
 2016-11-29    |           38.6923
 2016-11-30    |
```

ちなみに、2日目以降の継続率を計算する場合も、上記コード例のaction_log_with_mst_usersテーブルの中で、登録のn日後の日付を計算し、同様の操作を行うことで計算できます。

別解 ただし、各継続率の値をテーブルの列で表現すると、冗長なクエリとなって管理が難しくなるケースがあります。本項では、計算する指標のマスタを管理する一時テーブルを用いて、各指標を縦持ちで表現する方法を紹介します。

まず、前述のコード例では列に保持していた指標の情報を、下記コード例に示す通り、index_nameとinverval_dateを持つテーブルに行として保持します。index_nameは指標の名前、interval_dateは、登録後何日目の指標であるかを表しています。

コード5.2.2.4：継続率の指標を管理するマスタテーブルを作成するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval(index_name, interval_date) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、UNION ALLなどで代用可能
  --   ※3-4-5:擬似的なテーブルを作成する参照
  VALUES
    ('01 day repeat', 1)
  , ('02 day repeat', 2)
  , ('03 day repeat', 3)
  , ('04 day repeat', 4)
  , ('05 day repeat', 5)
  , ('06 day repeat', 6)
  , ('07 day repeat', 7)
)
SELECT *
FROM repeat_interval
ORDER BY index_name
;
```

```
  index_name   | interval_date
---------------+---------------
 01 day repeat |             1
 02 day repeat |             2
 03 day repeat |             3
 04 day repeat |             4
 05 day repeat |             5
 06 day repeat |             6
 07 day repeat |             7
```

この指標マスタを用いて、先ほどと同様の手順で継続率を計算するクエリを下記のコード例に示します。なお、列持ちのコード例と異なる箇所を強調して記載しています。

コード5.2.2.5：継続率を縦持ちで集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval AS (
  -- コード5.2.2.4参照
)
, action_log_with_index_date AS (
  SELECT
      u.user_id
    , u.register_date
    -- アクションの日付と、ログ全体の最新日付を日付型として取得
    , CAST(a.stamp AS date) AS action_date
    , MAX(CAST(a.stamp AS date)) OVER() AS latest_date
    -- ■ BigQueryの場合、一度タイムスタンプ型に変換してから日付型に変換
    -- , date(timestamp(a.stamp)) AS action_date
    -- , MAX(date(timestamp(a.stamp))) OVER() AS latest_date

    -- 登録日のn日後の日付を計算する
    , r.index_name

    -- ■ PostgreSQLの場合は下記を使用
    , CAST(CAST(u.register_date AS date) + interval '1 day' * r.interval_date AS date)
    -- ■ Redshiftの場合は下記を使用
    -- , dateadd(day, r.interval_date, u.register_date::date)
    -- ■ BigQueryの場合は下記を使用
    -- , date_add(CAST(u.register_date AS date), interval r.interval_date day)
    -- ■ Hive, SparkSQLの場合は下記を使用
    -- , date_add(CAST(u.register_date AS date), r.interval_date)
      AS index_date
  FROM
      mst_users AS u
    LEFT OUTER JOIN
      action_log AS a
    ON u.user_id = a.user_id
```

```
    CROSS JOIN
      repeat_interval AS r
)
, user_action_flag AS (
  SELECT
      user_id
    , register_date
    , index_name
      -- 4. 登録日のn日後にアクションをしたかどうかを0,1のフラグで表す
    , SIGN(
        -- 3. 登録日のn日後にアクションをした回数をユーザごとに合計する
        SUM(
          -- 1. 登録日のn日後が、ログの最新日付以前に含まれるか確認
          CASE WHEN index_date <= latest_date THEN
            -- 2. 登録日のn日後の日付にアクションをしている場合は1、それ以外は0を返す
            CASE WHEN index_date = action_date THEN 1 ELSE 0 END
          END
        )
      ) AS index_date_action
  FROM
    action_log_with_index_date
  GROUP BY
    user_id, register_date, index_name, index_date
)
SELECT
    register_date
  , index_name
  , AVG(100.0 * index_date_action) AS repeat_rate
FROM
  user_action_flag
GROUP BY
  register_date, index_name
ORDER BY
  register_date, index_name
;
```

```
 register_date |  index_name   |   repeat_rate
---------------+---------------+----------------
 2016-10-01    | 01 day repeat |       43.2354
 2016-10-01    | 02 day repeat |       33.3476
 2016-10-01    | 03 day repeat |       21.2456
 2016-10-01    | 04 day repeat |       18.9871
 2016-10-01    | 05 day repeat |       15.2389
 2016-10-01    | 06 day repeat |       12.8972
 2016-10-01    | 07 day repeat |        9.8991
 2016-10-02    | 01 day repeat |       41.3456
 2016-10-02    | 02 day repeat |       35.8972
...
```

```
 2016-11-23    | 06 day repeat |       11.8741
 2016-11-23    | 07 day repeat |
...
```

上記コード例の出力結果をみると、n日継続率を計算するために必要な判定期間のログが存在していない指標は、NULLになっていることが分かります。

定着率に関するレポート

継続率と同様のデータを用いて、定着率に関するレポートとそのレポートを作成するためのSQLを紹介します。

日次のn日定着率推移

継続率同様に、施策の効果が意図通り反映されているかを確認するために、定着率を日次で集計するレポートを作成します。なお、7日定着率が極端に低い場合は、定着率ではなく翌日継続率〜7日継続率までを確認して、問題点を検討しましょう。

表5.2.2.8：日次のn日定着率推移

日付	登録数	7日 定着率	14日 定着率	21日 定着率	28日 定着率
2016/6/1	1,632	85.2%	60.1%	24.6%	18.4%
2016/6/2	1,579	82.5%	60.7%	26.7%	20.8%
2016/6/3	1,601	83.3%	61.4%	26.2%	19.9%

定着率を算出する場合は、対象となる期間が複数の日付に跨がるため、前述のコード5.2.2.4（P.180）で示したrepeat_intervalテーブルのinterval_dateを、interval_begin_dateとinterval_end_dateに拡張します。下記にコード例を示します。

コード5.2.2.6：定着率の指標を管理するマスタテーブルを作成するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval(index_name, interval_begin_date, interval_end_date) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、UNION ALLなどで代用可能
  --   ※3-4-5：擬似的なテーブルを作成する参照
  VALUES
    ('07 day retention',  1,  7)
  , ('14 day retention',  8, 14)
  , ('21 day retention', 15, 21)
  , ('28 day retention', 22, 28)
)
```

```
SELECT *
FROM repeat_interval
ORDER BY index_name
;
```

```
    index_name    | interval_begin_date | interval_end_date
------------------+---------------------+-------------------
 07 day retention |                   1 |                 7
 14 day retention |                   8 |                14
 21 day retention |                  15 |                21
 28 day retention |                  22 |                28
```

次のステップ あとは、継続率の場合と同様に、各指標の定着率を計算します。

コード5.2.2.7：定着率を縦持ちで集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval AS (
  -- コード5.2.2.6参照
)
, action_log_with_index_date AS (
  SELECT
      u.user_id
    , u.register_date
    -- アクションの日付と、ログ全体の最新日付を日付型として取得
    , CAST(a.stamp AS date) AS action_date
    , MAX(CAST(a.stamp AS date)) OVER() AS latest_date
    -- ■ BigQueryの場合、一度タイムスタンプ型に変換してから日付型に変換
    -- , date(timestamp(a.stamp)) AS action_date
    -- , MAX(date(timestamp(a.stamp))) OVER() AS latest_date
    , r.index_name

    -- 指標の対象期間の開始日と終了日を計算する
    -- ■ PostgreSQLの場合は下記を使用
    , CAST(u.register_date::date + '1 day'::interval * r.interval_begin_date AS date)
      AS index_begin_date
    , CAST(u.register_date::date + '1 day'::interval * r.interval_end_date AS date)
      AS index_end_date

    -- ■ Redshiftの場合は下記を使用
    -- , dateadd(day, r.interval_begin_date, u.register_date::date) AS index_begin_date
    -- , dateadd(day, r.interval_end_date  , u.register_date::date) AS index_end_date
    -- ■ BigQueryの場合は下記を使用
    -- , date_add(CAST(u.register_date AS date), interval r.interval_begin_date day)
    --   AS index_begin_date
    -- , date_add(CAST(u.register_date AS date), interval r.interval_end_date day)
    --   AS index_end_date
```

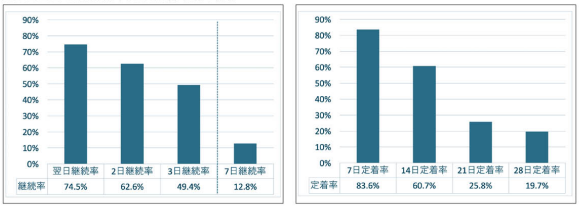
```
    -- ■ Hive, SparkSQLの場合は下記を使用
    -- , date_add(CAST(u.register_date AS date), r.interval_begin_date)
    --   AS index_begin_date
    -- , date_add(CAST(u.register_date AS date), r.interval_end_date)
    --   AS index_end_date
  FROM
      mst_users AS u
    LEFT OUTER JOIN
      action_log AS a
    ON u.user_id = a.user_id
    CROSS JOIN
      repeat_interval AS r
)
, user_action_flag AS (
  SELECT
      user_id
    , register_date
    , index_name
      -- 4. 指標の対象期間にアクションをしたかどうかを0,1のフラグで表す
    , SIGN(
        -- 3. 指標の対象期間にアクションをした回数をユーザごとに合計する
        SUM(
          -- 1. 指標の対象期間の終了日が、ログの最新日付以前に含まれるか確認
          CASE WHEN index_end_date <= latest_date THEN
            -- 2. 指標の対象期間にアクションをしている場合は1、それ以外は0を返す
            CASE WHEN action_date BETWEEN index_begin_date AND index_end_date
                 THEN 1 ELSE 0
            END
          END
        )
      ) AS index_date_action
  FROM
    action_log_with_index_date
  GROUP BY
    user_id, register_date, index_name, index_begin_date, index_end_date
)
SELECT
    register_date
  , index_name
  , AVG(100.0 * index_date_action) AS index_rate
FROM
  user_action_flag
GROUP BY
  register_date, index_name
ORDER BY
  register_date, index_name
;
```

```
register_date |    index_name    | index_rate
--------------+------------------+------------
 2016-10-01   | 07 day retention |    72.2136
 2016-10-01   | 14 day retention |    56.2363
 2016-10-01   | 21 day retention |    21.2367
 2016-10-01   | 28 day retention |     8.2378
 2016-10-02   | 07 day retention |    70.2169
 2016-10-02   | 14 day retention |    52.2374
 2016-10-02   | 21 day retention |    20.9871
...
 2016-11-10   | 07 day retention |    68.9871
 2016-11-10   | 14 day retention |    55.1467
 2016-11-10   | 21 day retention |
 2016-11-10   | 28 day retention |
...
```

n日継続率、n日定着率別の推移

下図のレポートに示す「n日継続率」と「n日定着率」ごとに集計することで、登録後どのぐらいの日数でユーザーが安定して利用するようになるのか、何日後を目処にサービス利用を辞めるユーザーが急に増えるのか分かります。もし、継続率や定着率が極端に下がるタイミングがあれば、その前後に何らかのお知らせを送信したり、n日継続して利用したユーザーにボーナスを与える施策を実施するなど、継続率と定着率が安定する日数まで、ユーザーが利用を続けるような計画を立案しましょう。

図5.2.2.2：n日継続率、n日定着率別の推移



また、定着率を計算するために設けたrepeat_intervalテーブルのフォーマットで、継続率を計算することも可能です。定着率のフォーマットに合わせることで、継続率と定着率の計算を同じクエリで実現できます。

下記コード例に、定着率のフォーマットに修正したクエリを示します。

コード5.2.2.8：継続率の指標を管理するマスタテーブルを定着率のフォーマットに修正したクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval(index_name, interval_begin_date, interval_end_date) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、UNION ALLなどで代用可能
  --   ※3-4-5：擬似的なテーブルを作成する参照
  VALUES
    ('01 day repeat'   ,  1,  1)
  , ('02 day repeat'   ,  2,  2)
  , ('03 day repeat'   ,  3,  3)
  , ('04 day repeat'   ,  4,  4)
  , ('05 day repeat'   ,  5,  5)
  , ('06 day repeat'   ,  6,  6)
  , ('07 day repeat'   ,  7,  7)
  , ('07 day retention',  1,  7)
  , ('14 day retention',  8, 14)
  , ('21 day retention', 15, 21)
  , ('28 day retention', 22, 28)
)
SELECT *
FROM repeat_interval
ORDER BY index_name
;
```

```
    index_name    | interval_begin_date | interval_end_date
------------------+---------------------+-------------------
 01 day repeat    |                   1 |                 1
 02 day repeat    |                   2 |                 2
 03 day repeat    |                   3 |                 3
 04 day repeat    |                   4 |                 4
 05 day repeat    |                   5 |                 5
 06 day repeat    |                   6 |                 6
 07 day repeat    |                   7 |                 7
 07 day retention |                   1 |                 7
 14 day retention |                   8 |                14
 21 day retention |                  15 |                21
 28 day retention |                  22 |                28
```

本項では例として、repeatに関する行のみ残して実行したコードを下記に示します。

コード5.2.2.9：n日継続率ごとの集計クエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval AS (
  -- コード5.2.2.8参照
)
, action_log_with_index_date AS (
```

```
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
SELECT
    index_name
  , AVG(100.0 * index_date_action) AS repeat_rate
FROM
  user_action_flag
GROUP BY
  index_name
ORDER BY
  index_name
;
```

```
  index_name   | repeat_rate
---------------+-------------
 01 day repeat |     42.1346
 02 day repeat |     30.2389
 03 day repeat |     22.9713
 04 day repeat |     18.9873
 05 day repeat |     15.1246
 06 day repeat |     12.9712
 07 day repeat |      9.2362
```

ワンポイント

継続率と定着率は共に、登録日を基準にn日後の行動を集計するため、登録日からn日経過しないと計算できません。したがって、30日継続率や60日継続率など、値を求めるには時間を要する指標を重要な指標に設定せず、翌日継続率や7日継続率、7日定着率など、まずは短期間で施策の結果が反映される指標を重要視することをおすすめします。

また、定着率は7日間の期間で集計するため、実際に何日利用しているのかは分かりません。14日定着期間のうち、70%のユーザーは1日～2日の利用に留まる、などの詳細を探りたい場合は、「5-1-4 ユーザーの訪問頻度を集計する」で前述したSQLを参考に集計しましょう。

関連項目	5-1-4 ユーザーの訪問頻度を集計する 3-1-4 日付やタイムスタンプを扱う 3-4-5 擬似的なテーブルを作成する

5-2-3 継続と定着に影響するアクションを集計する

SQL CROSS JOIN、CASE式、AVG関数　　分析 継続率、定着率

前項「5-2-2 継続率と定着率を算出する」(P.174) で継続率と定着率と呼ばれる指標を紹介しました。それぞれの推移を計算して、ユーザーの利用状況を知ることは重要なことですが、何を行ったら、継続率や定着率が向上するか分からなければ、施策の検討に至ることはできません。

こうした指標とレポートを作成すると、「〜〜率を上げよう」などの新たな目標や課題が課せられ、その課題には何をしたら良いのか、具体的な提言を求められます。まずは「〜〜しているユーザーは、何を達成しているのか」を調べましょう。

翌日継続率を改善するには、登録した当日に何を達成したのか調べることで、何を実施すれば翌日継続率に影響するのか糸口が見付かるはずです。14日定着率を上げたければ、7日定着率の判定期間内にどのような行動をしたのかを調べます。下図に示すレポートを作成し、継続率と定着率に影響すると推測されるアクションを集計しましょう。本項では例として、翌日継続率を改善するため、登録初日のアクションの利用有無で、翌日継続率にどれほど違いがあるかを調査するSQLを紹介します。

図 5.2.3.1：アクションに応じた、利用者、非利用者における翌日継続率比較

アクション	利用者数	利用率	利用者 翌日継続率	非利用者 翌日継続率
profile_regist	2,411	80.1%	52.1%	10.2%
tutorial-step1	2,371	78.7%	53.8%	10.8%
tutorial-step2	2,340	77.7%	55.5%	10.9%
tutorial-end	2,309	76.7%	55.9%	24.4%
post（投稿）	681	22.6%	81.8%	39.2%
comment（コメント）	477	15.8%	67.3%	51.3%
follow（フォロー）	1,691	56.2%	53.2%	42.0%

利用者の翌日継続率が高く、非利用者の翌日継続率が低いアクションが、より翌日継続率に影響すると判断できます。また、その利用率が低ければ、その行動を促すためにチュートリアルに盛り込んだり、アクションの利用を促進する、サイトのデザインや導線を検討しましょう。

各アクションについて、利用者と非利用者の翌日継続率を同時に計算するためには、全ユーザーと全アクションの組み合わせを作成し、ユーザーのアクション実行有無を0と1のフラグとして表現するテーブルを作成すると便利です。

まずは、全ユーザーとアクションの組み合わせを導出します。これは、ユーザーのマスタテーブルと、アクションのマスタテーブルをCROSS JOINで結合することで作成できます。

コード5.2.3.1：すべてのユーザーとアクションの組み合わせを導出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval(index_name, interval_begin_date, interval_end_date) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、SELECT文で代用可能
  --   ※3-4-5:擬似的なテーブルを作成する参照
  VALUES ('01 day repeat',  1,  1)
)
, action_log_with_index_date AS (
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
, mst_actions AS (
            SELECT 'view'    AS action
  UNION ALL SELECT 'comment' AS action
  UNION ALL SELECT 'follow'  AS action
)
, mst_user_actions AS (
  SELECT
      u.user_id
    , u.register_date
    , a.action
  FROM
      mst_users AS u
    CROSS JOIN
      mst_actions AS a
)
SELECT *
FROM mst_user_actions
ORDER BY user_id, action
;
```

```
    index_name    | interval_begin_date | interval_end_date
------------------+---------------------+-------------------
 07 day retention |                   1 |                 7
 14 day retention |                   8 |                14
 21 day retention |                  15 |                21
 28 day retention |                  22 |                28
```

次のステップ 続いて、各ユーザーについて、ユーザー登録日とその翌日におけるアクションの有無を0と1のフラグで表現します。

コード5.2.3.2：ユーザーのアクションログを0, 1のフラグで表現するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval AS (
  -- コード5.2.3.1参照
)
, action_log_with_index_date AS (
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
, mst_actions AS (
  -- コード5.2.3.1参照
)
, mst_user_actions AS (
  -- コード5.2.3.1参照
)
, register_action_flag AS (
  SELECT DISTINCT
      m.user_id
    , m.register_date
    , m.action
    , CASE
        WHEN a.action IS NOT NULL THEN 1
        ELSE 0
      END AS do_action
    , index_name
    , index_date_action
  FROM
      mst_user_actions AS m
    LEFT JOIN
      action_log AS a
      ON m.user_id = a.user_id
      AND CAST(m.register_date AS date) = CAST(a.stamp AS date)
      -- ■ BigQueryの場合、一度タイムスタンプ型に変換してから日付型に変換
      -- AND CAST(m.register_date AS date) = date(timestamp(a.stamp))
      AND m.action = a.action
    LEFT JOIN
      user_action_flag AS f
      ON m.user_id = f.user_id
  WHERE
    f.index_date_action IS NOT NULL
)
SELECT
  *
FROM
```

```
  register_action_flag
ORDER BY
  user_id, index_name, action
;
```

```
 user_id | register_date | action  | do_action |   index_name   | index_date_action
---------+---------------+---------+-----------+----------------+-------------------
 U001    | 2016-10-01    | comment |         0 | 01 day repeat |                 1
 U001    | 2016-10-01    | follow  |         1 | 01 day repeat |                 1
 U001    | 2016-10-01    | view    |         1 | 01 day repeat |                 1
 U002    | 2016-10-01    | comment |         0 | 01 day repeat |                 1
 U002    | 2016-10-01    | follow  |         0 | 01 day repeat |                 1
 U002    | 2016-10-01    | view    |         1 | 01 day repeat |                 1
 U003    | 2016-10-01    | comment |         0 | 01 day repeat |                 1
 U003    | 2016-10-01    | follow  |         0 | 01 day repeat |                 1
 U003    | 2016-10-01    | view    |         1 | 01 day repeat |                 1
...
```

上記コード例の実行結果の読み方は、例えば、ユーザー「U001」を例にすると、ユーザー登録日2016-10-01に「follow」と「view」のアクションを実行しており、翌日継続の判定期間には「comment」と「follow」、「view」の各アクションを実行していることを意味します。

次のステップ ユーザーのアクションログが0と1のフラグで表現できれば、後は前提条件における割合をAVG関数で計算するだけです。各アクションに関して、登録日にそのアクションを実行している利用者の条件は「do_action=1」、実行していない非利用者の条件は「do_action=0」と表現できます。その利用者／非利用者に対して、index_date_actionの平均を計算すれば、翌日継続率が計算できます。

コード5.2.3.3：アクションに応じた継続率、定着率を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval AS (
  -- コード5.2.3.1参照
)
, action_log_with_index_date AS (
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
, mst_actions(action) AS (
  -- コード5.2.3.1参照
)
, mst_user_actions AS (
```

```
  -- コード5.2.3.1参照
)
, register_action_flag AS (
  -- コード5.2.3.2参照
)
SELECT
    action
  , COUNT(1) users
  , AVG(100.0 * do_action) AS usage_rate
  , index_name
  , AVG(CASE do_action WHEN 1 THEN 100.0 * index_date_action END) AS idx_rate
  , AVG(CASE do_action WHEN 0 THEN 100.0 * index_date_action END) AS no_action_idx_rate
FROM
  register_action_flag
GROUP BY
  index_name, action
ORDER BY
  index_name, action
;
```

```
 action   | users | usage_rate |    index_name    | idx_rate  | no_action_idx_rate
----------+-------+------------+------------------+-----------+--------------------
 comment | 28    |    7.1429  | 01 day repeat    |    0.0000 |   15.3846
 follow   | 28    |    7.1429  | 01 day repeat    |   50.0000 |   11.5385
 view     | 28    |  100.0000  | 01 day repeat    |   14.2857 |
```

ワンポイント

特定アクションの実行が継続率と定着率の向上に繋がると判明しても、そのアクションを実行するハードルが極めて高い場合、継続率と定着率の影響が多少低くても、よりアクションが実行しやすいものを選ぶべきです（例：動画を投稿するよりも画像付きの投稿を促すなど）。

また、本項ではアクションの有無を集計しましたが、アクション回数によって違いがあるかもしれません。次項ではアクション回数に着目してその詳細を探ります。

5-2-4 アクション回数に応じた定着率を集計する

SQL CROSS JOIN、CASE式、AVG関数　　分析 定着率

FacebookやTwitterなどに代表されるSNSの事例で、「登録後1週間以内に10人をフォローしたらユーザーはサービスを継続的に利用する」ことを耳にしたことはないでしょうか。このような調査結果から、大きな方針を示すことができれば、「○○さんは友達ではないですか？」、「人気のユーザーをフォローしましょう」などを表示するモジュールを開発したり、チュートリアルに人気ユーザーをフォロー

する導線を設けるなど、サービスの活性化に繋がる具体的な施策を検討できます。

本項では例として、下図に示す通り、登録日と登録日以降の7日以内（7日定着判定期間）に実行したアクション数に応じて、14日定着率に変化があるかを集計します。

図5.2.4.1：今回のアクション集計日と定着率の判定期間

本項では、下図に示す通り、各アクション別に度数分布表を作成し、達成者における14日定着率を集計するクエリを紹介します。

図5.2.4.2：アクション回数に応じた定着率

アクション	階級	達成者	達成者 14日定着率
post	0	2,250	49.4%
	1以上5以下	665	45.1%
	6以上10以下	391	76.7%
	11以上	225	88.9%
follow	0	1,201	43.6%
	1以上5以下	923	56.4%
	6以上10以下	791	59.5%
	11以上	595	67.1%

まず、上図に示したアクションの階級マスタを一時テーブルとして定義し、定着率を求めるためのユーザーアクションフラグとの組み合わせを、CROSS JOINで求めます。そのコード例を下記に示します。

コード5.2.4.1：アクションの階級マスタとユーザーアクションフラグの組み合わせを算出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
repeat_interval(index_name, interval_begin_date, interval_end_date) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、SELECT文で代用可能
  --   ※3-4-5：擬似的なテーブルを作成する参照
  VALUES ('14 day retention',  8, 14)
)
, action_log_with_index_date AS (
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
, mst_action_bucket(action, min_count, max_count) AS (
  -- アクションの階級マスタ
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、SELECT文とUNION ALLで代用可能
  VALUES
    ('comment',  0,    0)
  , ('comment',  1,    5)
  , ('comment',  6,   10)
  , ('comment', 11, 9999) -- 最大値として便宜上9999を入れている
  , ('follow' ,  0,    0)
  , ('follow' ,  1,    5)
  , ('follow' ,  6,   10)
  , ('follow' , 11, 9999) -- 最大値として便宜上9999を入れている
)
, mst_user_action_bucket AS (
  -- ユーザーマスタとアクション階級マスタの組み合わせを算出
  SELECT
      u.user_id
    , u.register_date
    , a.action
    , a.min_count
    , a.max_count
  FROM
      mst_users AS u
    CROSS JOIN
      mst_action_bucket AS a
)
SELECT *
FROM
  mst_user_action_bucket
ORDER BY
  user_id, action, min_count
;
```

```
 user_id | register_date | action  | min_count | max_count
---------+---------------+---------+-----------+-----------
 U001    | 2016-01-01    | comment |         0 |         0
 U001    | 2016-01-01    | comment |         1 |         5
 U001    | 2016-01-01    | comment |         6 |        10
 U001    | 2016-01-01    | comment |        11 |      9999
 U001    | 2016-01-01    | follow  |         0 |         0
 U001    | 2016-01-01    | follow  |         1 |         5
 U001    | 2016-01-01    | follow  |         6 |        10
 U001    | 2016-01-01    | follow  |        11 |      9999
 U002    | 2016-01-01    | comment |         0 |         0
…
```

次のステップ 上記コード例の出力結果に、登録後7日間のログをLEFT JOINし、登録後7日間のアクション度数を集計するクエリを、下記のコード例に示します。

なお、下記のコード例では、登録後7日間のログを結合するため、JOIN句内でBETWEEN句を用いますが、Hiveでは動作しません。別途WHERE句での絞り込みが必要となりますが割愛します。

コード5.2.4.2：登録後7日間のアクション度数を集計するクエリ

PostgreSQL Redshift BigQuery SparkSQL

```
WITH
repeat_interval AS (
  -- コード5.2.4.1参照
)
, action_log_with_index_date AS (
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
, mst_action_bucket AS (
  -- コード5.2.4.1参照
)
, mst_user_action_bucket AS (
  -- コード5.2.4.1参照
)
, register_action_flag AS (
  -- 登録日から7日後までのアクション数をカウントし、
  -- アクション階級と14日定着の達成フラグを算出する
  SELECT
      m.user_id
    , m.action
    , m.min_count
    , m.max_count
    , COUNT(a.action) AS action_count
    , CASE
        WHEN COUNT(a.action) BETWEEN m.min_count AND m.max_count THEN 1
```

```
        ELSE 0
      END AS achieve
    , index_name
    , index_date_action
  FROM
      mst_user_action_bucket AS m
    LEFT JOIN
      action_log AS a
      ON m.user_id = a.user_id
      -- 登録日当日から7日後までのアクションログを結合
      -- ■ PostgreSQL, Redshiftの場合は下記
       AND CAST(a.stamp AS date)
             BETWEEN CAST(m.register_date AS date)
                 AND CAST(m.register_date AS date) + interval '7 days'
      -- ■ BigQueryの場合は下記
      -- AND date(timestamp(a.stamp))
      --       BETWEEN CAST(m.register_date AS date)
      --           AND date_add(CAST(m.register_date AS date), interval 7 day)
      -- ■ SparkSQLの場合は下記
      -- AND CAST(a.stamp AS date)
      --       BETWEEN CAST(m.register_date AS date)
      --           AND date_add(CAST(m.register_date AS date), 7)
      -- ■ Hiveの場合は、JOIN句でBETWEEN句が使用できないため、WHERE句での絞り込みが必要
      AND m.action = a.action
    LEFT JOIN
      user_action_flag AS f
      ON m.user_id = f.user_id
  WHERE
    f.index_date_action IS NOT NULL
  GROUP BY
      m.user_id
    , m.action
    , m.min_count
    , m.max_count
    , f.index_name
    , f.index_date_action
)
SELECT *
FROM
  register_action_flag
ORDER BY
  user_id, action, min_count
;
```

```
-[ RECORD 1 ]-----+----------------
user_id           | U001
action            | comment
min_count         | 0
max_count         | 0
```

```
action_count      | 0
achieve           | 1
index_name        | 14 day retention
index_date_action | 1
-[ RECORD 2 ]-----+----------------
user_id           | U001
action            | comment
min_count         | 1
max_count         | 5
action_count      | 0
achieve           | 0
index_name        | 14 day retention
index_date_action | 1
...
```

次のステップ 最後に、上記コード例で求めたアクション度数別に、14日定着率を集計します。

コード5.2.4.3：登録後7日間のアクション度数別に14日定着率を集計するクエリ

PostgreSQL Redshift BigQuery SparkSQL

```
WITH
repeat_interval AS (
  -- コード5.2.4.1参照
)
, action_log_with_index_date AS (
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
, mst_action_bucket AS (
  -- コード5.2.4.1参照
)
, mst_user_action_bucket AS (
  -- コード5.2.4.1参照
)
, register_action_flag AS (
  -- コード5.2.4.2参照
)
SELECT
    action
  -- ■ PostgreSQL, Redshiftの場合は下記で文字列を連結する
  , min_count || ' ~ ' || max_count AS count_range
  -- ■ BigQueryの場合は下記で文字列を連結する
  -- , CONCAT(CAST(min_count AS string), ' ~ ', CAST(max_count AS string))
  --   AS count_range
  , SUM(CASE achieve WHEN 1 THEN 1 ELSE 0 END) AS achieve
  , index_name
  , AVG(CASE achieve WHEN 1 THEN 100.0 * index_date_action END) AS achieve_index_rate
FROM
```

```
  register_action_flag
GROUP BY
    index_name, action, min_count, max_count
ORDER BY
    index_name, action, min_count;
```

```
 action  | count_range | achieve |    index_name    | achieve_index_rate
---------+-------------+---------+------------------+--------------------
 comment | 0 ~ 0       |      24 | 14 day retention |             4.1666
 comment | 1 ~ 5       |       3 | 14 day retention |            33.3333
 comment | 6 ~ 10      |       0 | 14 day retention |
 comment | 11 ~ 9999   |       0 | 14 day retention |
 follow  | 0 ~ 0       |      25 | 14 day retention |             4.0000
 follow  | 1 ~ 5       |       2 | 14 day retention |            50.0000
 follow  | 6 ~ 10      |       0 | 14 day retention |
 follow  | 11 ~ 9999   |       0 | 14 day retention |
```

本項ではユーザーが行ったアクションに着目して集計していますが、受けたアクション別（いいね、コメント、フォローなど）で集計すると、より活性化させるべきアクションが見えてくるはずです。

関連項目	4-2-4 購入価格帯を集計する

5-2-5 利用日数に応じた定着率を集計する

SQL COUNT関数、SUMウィンドウ関数、AVG関数　分析 定着率

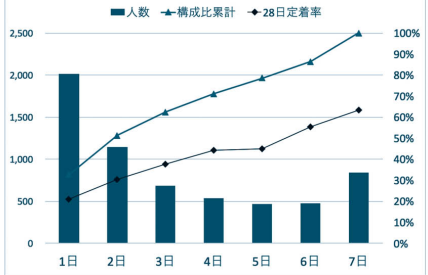
本項では、7日定着期間に何日利用したかで、以降の定着率にどのような影響があるのかを調査するため、ある月の登録者の「翌日から7日以内の利用日数」を集計し、「28日定着率（登録日の21日後から28日後に利用したユーザーの割合）」を計算し、次図に示すレポートを作成する手順を説明します。

このレポートを確認すると、以下のことが分かります。

- 約70%のユーザーは7日定着判定期間（登録翌日から7日以内）に1日～4日しか起動していない
- 7日定着判定期間で1日しか利用しないユーザーの28日定着率は20.8%
- 5日利用したユーザーでも28日定着率は45.0%
- ただし、6日利用すると28日定着率は55.5%となり、3日に比べて10.5pt向上する

毎日利用してもらうことが望ましいのは容易に推測できますが、このレポートに示す通り、数値化して図示することで、その根拠が明らかになります。その結果、定着率改善のためには、登録翌日から7

図 5.2.5.1：登録日の翌日から7日以内の利用日数に応じた28日定着率



利用日数	1日	2日	3日	4日	5日	6日	7日
人数	2,012	1,143	682	536	454	476	839
割合	32.7%	18.6%	11.1%	8.7%	7.5%	7.7%	13.6%
構成比累計	32.7%	51.3%	62.4%	71.1%	78.6%	86.4%	100.0%
28日定着人数	419	348	256	237	209	264	533
28日定着率	20.8%	30.4%	37.5%	44.2%	45.0%	55.5%	63.5%

日以内に6日以上、ユーザーが利用することを目的とした施策を検討することになります。

例えば、ソーシャルゲームであれば、「登録直後のユーザーに対して、1日～5日間特別に連続ログインボーナスを与え、さらに6日目にはより大きなボーナスを用意して、何とか6日間は利用してもらう」などの施策を検討します。

上図に示すレポートの作成方法は、下記の通りです。

1. 登録日の翌日から7日以内の利用日数を集計する
2. 利用日数別に集計したユーザー数の構成比と構成比累計を計算する
3. 利用日数別に集計したユーザー数を母数に28日定着率を集計し、その割合を計算する

まず、定着率の指標マスタを作成して、user_action_flagを作成します。次に、ユーザーマスタに登録日の翌日から7日以内のアクションログを結合し、その期間の利用日数をカウントします。

続いて、利用日数のカウントとuser_action_flagを結合し、両者の関係を確認するためのテーブル(register_action_flag)を作成します。次のコード例にクエリを示します。

コード5.2.5.1：登録日の翌日から7日以内の利用日数と28日定着フラグを算出するクエリ

PostgreSQL Redshift BigQuery SparkSQL

```
WITH
repeat_interval(index_name, interval_begin_date, interval_end_date) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、SELECT文で代用可能
  --   ※3-4-5:擬似的なテーブルを作成する参照
  VALUES ('28 day retention', 22, 28)
)
, action_log_with_index_date AS (
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
, register_action_flag AS (
  SELECT
      m.user_id
    , COUNT(DISTINCT CAST(a.stamp AS date)) AS dt_count
    -- ■ BigQueryの場合は下記
    -- , COUNT(DISTINCT date(timestamp(a.stamp))) AS dt_count
    , index_name
    , index_date_action
  FROM
      mst_users AS m
    LEFT JOIN
      action_log AS a
      ON m.user_id = a.user_id
      -- 登録日翌日から7日以内のアクションログを結合
      -- ■ PostgreSQL, Redshiftの場合は下記
       AND CAST(a.stamp AS date)
             BETWEEN CAST(m.register_date AS date) + interval '1 day'
                 AND CAST(m.register_date AS date) + interval '8 days'
      -- ■ BigQueryの場合は下記
      -- AND date(timestamp(a.stamp))
      --       BETWEEN date_add(CAST(m.register_date AS date), interval 1 day)
      --           AND date_add(CAST(m.register_date AS date), interval 8 day)
      -- ■ SparkSQLの場合は下記
      -- AND CAST(a.stamp AS date)
      --       BETWEEN date_add(CAST(m.register_date AS date), 1)
      --           AND date_add(CAST(m.register_date AS date), 8)
      -- ■ Hiveの場合は、JOIN句でBETWEEN句が使用できないため、WHERE句での絞り込みが必要
    LEFT JOIN
      user_action_flag AS f
      ON m.user_id = f.user_id
  WHERE
    f.index_date_action IS NOT NULL
  GROUP BY
      m.user_id
    , f.index_name
    , f.index_date_action
```

```
)
SELECT *
FROM
  register_action_flag;
```

```
 user_id | dt_count |    index_name    | index_date_action
---------+----------+------------------+-------------------
 U027    |        1 | 28 day retention |                 0
 U021    |        1 | 28 day retention |                 0
 U020    |        1 | 28 day retention |                 0
 U002    |        2 | 28 day retention |                 0
 U003    |        2 | 28 day retention |                 0
 U004    |        2 | 28 day retention |                 0
 U008    |        3 | 28 day retention |                 1
 U015    |        3 | 28 day retention |                 1
 U001    |        4 | 28 day retention |                 0
```

上記コード例の出力結果として、登録日翌日から7日以内の利用日数と、そのユーザーの28日定着フラグのリストが得られます。

次のステップ 上記の結果を利用日数ごとに集約して、利用日数に応じた定着率を計算します。下記コード例に、利用日数に応じた定着率を集計するクエリを示します。

コード5.2.5.2：利用日数に応じた定着率を集計するクエリ

PostgreSQL Redshift BigQuery SparkSQL

```
WITH
repeat_interval AS (
  -- コード5.2.5.1参照
)
, action_log_with_index_date AS (
  -- コード5.2.2.7参照
)
, user_action_flag AS (
  -- コード5.2.2.7参照
)
, register_action_flag AS (
  -- コード5.2.5.1参照
)
SELECT
     dt_count AS dates
   , COUNT(user_id) AS users
   , 100.0 * COUNT(user_id) / SUM(COUNT(user_id)) OVER() AS user_ratio
   , 100.0
     * SUM(COUNT(user_id))
         OVER(ORDER BY index_name, dt_count
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
     / SUM(COUNT(user_id)) OVER() AS cum_ratio
```

```
    , SUM(index_date_action) AS achieve_users
    , AVG(100.0 * index_date_action) AS achieve_ratio
FROM
  register_action_flag
GROUP BY
  index_name, dt_count
ORDER BY
  index_name, dt_count;
```

```
 dates | users | user_ratio | cum_ratio | achieve_users | achieve_ratio
-------+-------+------------+-----------+---------------+---------------
     1 |  2102 |       32.7 |      32.7 |           419 |          20.8
     2 |  1143 |       18.6 |      51.3 |           348 |          30.4
     3 |   682 |       11.1 |      62.4 |           256 |          37.5
     4 |   536 |        8.7 |      71.1 |           237 |          44.2
...
```

ワンポイント

本項では、登録日の翌日から7日以内の利用日数を用いて集計しましたが、利用日数ではなく、SNSサービスの投稿回数やソーシャルゲームのレベルなど、対象を変えて分析してみましょう。

関連項目	5-1-4 ユーザーの訪問頻度を集計する

5-2-6 ユーザーの残存率を集計する

SQL CROSS JOIN、SUM(CASE～)、AVG(CASE～)

分析 残存率

サービス登録の数ヶ月後、どれぐらいの割合のユーザーがサービス利用を継続しているのかを大雑把かつ長期的に把握することで、サービスに対する問題点の洗い出し、過去と現在の比較、未来の目標に対する見込みの検討などが可能になります。

下表に示す通り、横軸に登録月、縦に該当月のサービス利用者数を集計した表を作成します。

表5.2.6.1：ユーザーの残存率

	2016年1月	2016年2月	2016年3月	2016年4月	2016年5月	2016年6月
2016年1月	100(100%)					
2016年2月	90(90%)	150(100%)				
2016年3月	80(80%)	120(80%)	120(100%)			
2016年4月	70(70%)	90(60%)	90(75%)	200(100%)		
2016年5月	60(60%)	60(40%)	60(50%)	150(75%)	250(100%)	
2016年6月	50(50%)	30(20%)	30(25%)	100(50%)	125(50%)	220(100%)

この表を作成して、以下の項目に該当する問題点がないかチェックしましょう。

- 以前に比べてn ヶ月後の残存率が下がっている
  →新規登録者に対して、サービス利用のハードルが上がっているのではないか？
- n ヶ月後に残存率が急に下がる傾向がある
  →サービスを利用するに当たり、一通りの目的を達成するタイミング、
  飽きるタイミングが想定よりも早く迎えるのではないか？
- 古くから利用していたユーザーが特定の月を境に利用しなくなっている
  →競合にユーザーを取られているのではないか？

ユーザーの残存率を月単位で集計するには、まずユーザーのサービス登録月から12ヶ月後までの月を導出するための補助テーブルを定義します。下記にそのコード例を示します。

コード5.2.6.1：12ヶ月後までの月を導出するための補助テーブルを作成するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
mst_intervals(interval_month) AS (
  -- 12ヶ月分の連番を作成する(generate_series等で代用可能)
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、SELECT文とUNION ALLで代用可能
  --   ※3-4-5:擬似的なテーブルを作成する参照
  VALUES (1), (2), (3), (4), (5), (6)
       , (7), (8), (9), (10), (11), (12)
)
SELECT *
FROM mst_intervals
;
```

```
 interval_month
----------------
              1
              2
              3
...
             10
             11
             12
```

上記の補助テーブルを用いて、ユーザーの登録月から12ヶ月後までの月を、ユーザーマスタに付与します。続いて、アクションログのログ日付を月表現に変換し、登録月から12ヶ月後までの月を付与したユーザーマスタに結合し、月単位で残存率を集計します。

下記のコード例に、ユーザーのサービス登録月から12ヵ月後までの残存率を集計するクエリを示します。

コード5.2.6.2：登録月から12ヶ月後までの残存率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_intervals AS (
  -- コード5.2.6.1参照
)
, mst_users_with_index_month AS (
  -- ユーザーマスタに登録月から12ヶ月後までの月を付与する
  SELECT
      u.user_id
    , u.register_date
    -- nヶ月後の日付、登録月、登録月nヶ月後の月を計算
    -- ■ PostgreSQLの場合、下記を使用
    , CAST(u.register_date::date + i.interval_month * '1 month'::interval AS date)
      AS index_date
    , substring(u.register_date, 1, 7) AS register_month
    , substring(CAST(
        u.register_date::date + i.interval_month * '1 month'::interval
      AS text), 1, 7) AS index_month
    -- ■ Redshiftの場合、下記を使用
    -- , dateadd(month, i.interval_month, u.register_date::date) AS index_date
    -- , substring(u.register_date, 1, 7) AS register_month
    -- , substring(
    --     CAST(dateadd(month, i.interval_month, u.register_date::date) AS text), 1, 7)
    --   AS index_month
    -- ■ BigQueryの場合、下記を使用
    -- , date_add(date(timestamp(u.register_date)), interval i.interval_month month)
    --   AS index_date
    -- , substr(u.register_date, 1, 7) AS register_month
    -- , substr(CAST(
    --     date_add(date(timestamp(u.register_date)), interval i.interval_month month)
    --   AS string), 1, 7) AS index_month
    -- ■ Hive, SparkSQLの場合、下記を使用
    -- , add_months(u.register_date, i.interval_month) AS index_date
    -- , substring(u.register_date, 1, 7) AS register_month
    -- , substring(
    --     CAST(add_months(u.register_date, i.interval_month) AS string), 1, 7)
    --   AS index_month
  FROM
      mst_users AS u
    CROSS JOIN
      mst_intervals AS i
)
, action_log_in_month AS (
  -- アクションログのログ日付を月表示に変換する
  SELECT DISTINCT
      user_id
```

```
    , substring(stamp, 1, 7) AS action_month
    -- ■ BigQueryの場合はsubstrを用いる
    -- , substr(stamp, 1, 7) AS action_month
  FROM
    action_log
)
SELECT
  -- ユーザーマスタとアクションログを結合し、月ごとに残存率を集計する
    u.register_month
  , u.index_month

  -- action_monthがNULLでない(アクションを行なった)ユーザーをカウントする
  , SUM(CASE WHEN a.action_month IS NOT NULL THEN 1 ELSE 0 END) AS users
  , AVG(CASE WHEN a.action_month IS NOT NULL THEN 100.0 ELSE 0.0 END)
    AS retension_rate
FROM
    mst_users_with_index_month AS u
  LEFT JOIN
    action_log_in_month AS a
    ON u.user_id = a.user_id
    AND u.index_month = a.action_month
 GROUP BY
   u.register_month, u.index_month
 ORDER BY
   u.register_month, u.index_month
;
```

```
 register_month | index_month | users | retension_rate
----------------+-------------+-------+----------------
 2016-10        | 2016-10     | 12895 |       100.0000
 2016-10        | 2016-11     |  8723 |        67.6463
 2016-10        | 2016-12     |  4897 |        37.9759
...
 2016-11        | 2016-11     | 11436 |       100.0000
 2016-11        | 2016-12     |  7213 |        63.0727
 2016-11        | 2017-01     |  3786 |        33.1059
...
```

ワンポイント

日々チェックするレポートとしては大雑把ですが、長期的なスパンでユーザー登録と継続利用の大局を把握する際には有効です。また、このレポートを作成時は、該当月に実施した施策やキャンペーンなどのイベントを記載すると、数値の変動を読み解く材料となり、より有効に活用できます。

関連項目 | 3-4-5 擬似的なテーブルを作成する

5-2-7 訪問頻度からユーザーの属性を定義し集計する

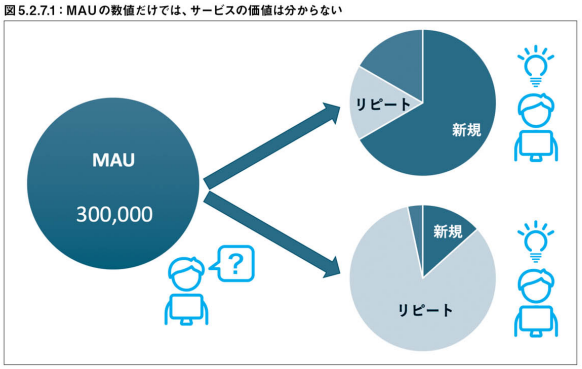
SQL CASE式、NULLIF関数、LAG関数　　分析 MAU、リピート、カムバック

前述の「5-1-4 ユーザーの訪問頻度を集計する」(P.145) では、1週間に何日訪問しているかを集計しましたが、本項では、サービス利用者の訪問頻度を月別で捉えた際に、その頻度に応じていくつかの属性で分類し、その内訳を集計する方法を紹介します。

MAU

月単位でサービスを利用したユーザー数をMAU (Monthly Active Users) と呼びます。「今月のMAUは30万でした」と報告を受けた場合、その30万人はほぼ新規ユーザーで構成されているのか、サービスを常日頃から利用している30万人なのか、その内訳は不明です。その内訳によってサービスの価値も変わります。もちろん、検討すべき施策の内容も変わってきます。

図5.2.7.1：MAUの数値だけでは、サービスの価値は分からない



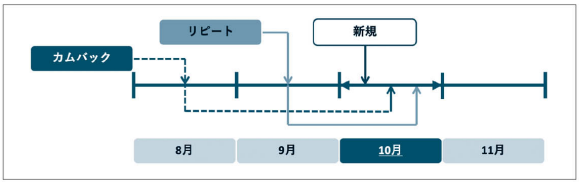
MAUを3つに分解する

上図に示す通り、MAUだけではどのようなユーザーがサービスを利用しているか把握することは困難です。サービスを利用しているユーザーの構成をより詳細に把握するため、本項ではMAUを下記の3つの属性に分解します。

- 新規ユーザー　　：当月に登録した新規ユーザー
- リピートユーザー　：前月も利用しているユーザー
- カムバックユーザー：当月に新規登録していない＆前月利用もしていない
  しばらく利用期間が空いていたユーザー

下図に、10月のMAUを3属性に分解するイメージを示します。

図5.2.7.2：MAUにおけるユーザーの分類



10万のMAUに対して施策を検討すると、どうしても施策が曖昧になってしまいます。新規ユーザー2万人、リピートユーザー7万人、カムバックユーザー1万人で構成されるが、新規ユーザーの2万人を対象に、リピートする施策は何か検討する方が確度も上がります。まずはMAUの内訳を集計しましょう。下記コード例に、上記3分類でMAUの内訳を集計するクエリを示します。

コード5.2.7.1：新規ユーザー数、リピートユーザー数、カムバックユーザー数を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
monthly_user_action AS (
  -- 月ごとにユーザーアクションを集約
  SELECT DISTINCT
      u.user_id
    -- ■ PostgreSQLの場合、下記を使用
    , substring(u.register_date, 1, 7) AS register_month
    , substring(l.stamp, 1, 7) AS action_month
    , substring(CAST(
        l.stamp::date - interval '1 month' AS text
      ), 1, 7) AS action_month_priv
    -- ■ Redshiftの場合、下記を使用
    -- , substring(u.register_date, 1, 7) AS register_month
    -- , substring(l.stamp, 1, 7) AS action_month
    -- , substring(
    --     CAST(dateadd(month, -1, l.stamp::date) AS text), 1, 7)
    --   AS action_month_priv
    -- ■ BigQueryの場合、下記を使用
    -- , substr(u.register_date, 1, 7) AS register_month
```

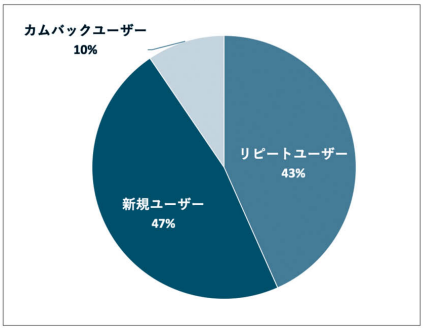
```
    -- , substr(l.stamp, 1, 7) AS action_month
    -- , substr(CAST(
    --     date_sub(date(timestamp(l.stamp)), interval 1 month)
    --   AS string), 1, 7) AS action_month_priv
    -- ■ Hive, SparkSQLの場合、下記を使用
    -- , substring(u.register_date, 1, 7) AS register_month
    -- , substring(l.stamp, 1, 7) AS action_month
    -- , substring(
    --     CAST(add_months(to_date(l.stamp), -1) AS string), 1, 7)
    --   AS action_month_priv

  FROM
      mst_users AS u
    JOIN
      action_log AS l
      ON u.user_id = l.user_id
)
, monthly_user_with_type AS (
  -- 月別のユーザー分類テーブル
  SELECT
      action_month
    , user_id
    , CASE
        -- 登録月がアクションした月に一致する場合は新規ユーザー
        WHEN register_month = action_month THEN 'new_user'
        -- 前月のアクションが存在する場合はリピートユーザー
        WHEN action_month_priv
             = LAG(action_month)

                OVER(PARTITION BY user_id ORDER BY action_month)
                -- ■ SparkSQLの場合は下記を使用
                -- OVER(PARTITION BY user_id ORDER BY action_month
                --   ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)

            THEN 'repeat_user'
        -- 上記以外はカムバックユーザー
        ELSE 'come_back_user'
      END AS c
    , action_month_priv
  FROM
    monthly_user_action
)
SELECT
    action_month

    -- 当月のMAU
  , COUNT(user_id) AS mau

    -- ======================================
    -- new_users:       当月の新規ユーザー数
    -- repeat_users:    当月のリピートユーザー数
```

```
    -- come_back_users: 当月のカムバックユーザー数
    -- ========================================
  , COUNT(CASE WHEN c = 'new_user'       THEN 1 END) AS new_users
  , COUNT(CASE WHEN c = 'repeat_user'    THEN 1 END) AS repeat_users
  , COUNT(CASE WHEN c = 'come_back_user' THEN 1 END) AS come_back_users
FROM
    monthly_user_with_type
GROUP BY
  action_month
ORDER BY
  action_month
;
```

```
 action_month |   mau  | new_users | repeat_users | come_back_users
--------------+--------+-----------+--------------+-----------------
 2016-03      |  21931 |     21931 |            0 |               0
 2016-04      |  35555 |     27755 |         7800 |               0
 2016-05      | 117819 |    106249 |        10634 |             936
 2016-06      | 320684 |    277121 |        40586 |            2977
 2016-07      | 429247 |    314249 |       109213 |            5785
 2016-08      | 293556 |    138463 |       127257 |           27846
```

上記コード例の出力結果から、2016年8月のMAUでの3要素（新規ユーザー・リピートユーザー・カムバックユーザー）の構成を円グラフで表現すると、下図の通りです。

図5.2.7.3：MAUを3つの属性に分類



この通り、構成を図式化すると、「今月のMAUは約30万人です。全体の約50%は新規ユーザーで、リピーターは約40%です。残りの10%は（前月利用していなかったユーザーが再度利用した）カムバックユーザーで構成されています」と分かり、曖昧だったMAUの輪郭が見えてきます。

続いて、さらに詳細に報告できるように説明しましょう。

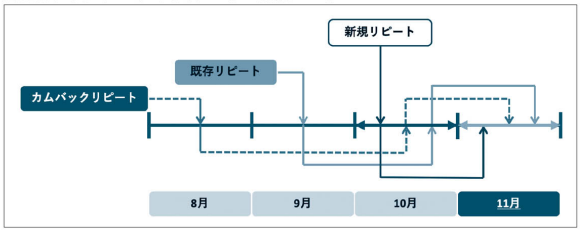
リピートユーザーを3つに分解する

リピートユーザーは前月も利用しているユーザーと定義しています。前月のユーザー状態によって、さらにリピートユーザーを3つに分類します。

- 新規リピートユーザー　：前月は新規ユーザーとして分類されたユーザーが、当月も利用した
- 既存リピートユーザー　：前月もリピートユーザーとして分類されたユーザーが当月も利用した
- カムバックリピートユーザー　：前月カムバックユーザーとして分類されたユーザーが当月も利用した

少々複雑になりますが、前掲の図5.2.7.2に加えて、10月のMAU区分に応じて、11月のリピートユーザーをさらに3つの区分に分解するイメージを下図に示します。

図5.2.7.4：リピートユーザーにおけるユーザーの分類について



リピートユーザーをさらに3要素に分解することで、同じリピートユーザーでもまだ登録したばかりの経験の浅いユーザーで構成されているのか、毎月利用しているユーザーがどれほどいるのかなど、サービスを構成するユーザー層をより詳細に把握できます。

上図にしたがって、リピートユーザーをさらに3つに分類するSQLを下記のコード例に示します。

コード5.2.7.2：リピートユーザーを細分化して集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
monthly_user_action AS (
-- コード5.2.7.1を参照
)
, monthly_user_with_type AS (
-- コード5.2.7.1を参照
)
, monthly_users AS (
  SELECT
      m1.action_month

    , COUNT(m1.user_id) AS mau

    , COUNT(CASE WHEN m1.c = 'new_user'       THEN 1 END) AS new_users
    , COUNT(CASE WHEN m1.c = 'repeat_user'    THEN 1 END) AS repeat_users
    , COUNT(CASE WHEN m1.c = 'come_back_user' THEN 1 END) AS come_back_users

      -- ========================================
      -- new_repeat_users:
      --   当月がリピートユーザーかつ、前月が新規ユーザーであったユーザー数
      -- continuous_repeat_users:
      --   当月がリピートユーザーかつ、前月がリピートユーザーであったユーザー数
      -- come_back_repeat_users:
      --   当月がリピートユーザーかつ、前月がカムバックユーザーであったユーザー数
      -- ========================================
    , COUNT(
        CASE WHEN m1.c = 'repeat_user' AND m0.c = 'new_user' THEN 1 END
      ) AS new_repeat_users
    , COUNT(
        CASE WHEN m1.c = 'repeat_user' AND m0.c = 'repeat_user' THEN 1 END
      ) AS continuous_repeat_users
    , COUNT(
        CASE WHEN m1.c = 'repeat_user' AND m0.c = 'come_back_user' THEN 1 END
      ) AS come_back_repeat_users

  FROM
      -- m1: 当月のユーザー分類テーブル
      monthly_user_with_type AS m1
    LEFT OUTER JOIN
      -- m0: 前月のユーザー分類テーブル
      monthly_user_with_type AS m0
      ON  m1.user_id = m0.user_id
      AND m1.action_month_priv = m0.action_month
  GROUP BY
    m1.action_month
)
SELECT
  *
FROM
```

```
  monthly_users
ORDER BY
  action_month;
```

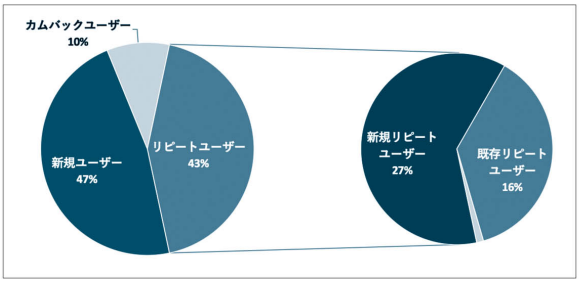


```
-[ RECORD 1 ]-----------+--------
action_month            | 2016-03
mau                     | 21931
new_users               | 21931
repeat_users            | 0
come_back_users         | 0
new_repeat_users        | 0
continuous_repeat_users | 0
come_back_repeat_users  | 0
-[ RECORD 2 ]-----------+--------
action_month            | 2016-04
mau                     | 35555
new_users               | 27755
repeat_users            | 7800
come_back_users         | 0
new_repeat_users        | 7800
continuous_repeat_users | 0
come_back_repeat_users  | 0
-[ RECORD 3 ]-----------+--------
action_month            | 2016-05
mau                     | 117819
new_users               | 106249
repeat_users            | 10634
come_back_users         | 936
new_repeat_users        | 7267
continuous_repeat_users | 3367
come_back_repeat_users  | 0
-[ RECORD 4 ]-----------+--------
action_month            | 2016-06
mau                     | 320684
new_users               | 277121
repeat_users            | 40586
come_back_users         | 2977
new_repeat_users        | 34593
continuous_repeat_users | 5590
come_back_repeat_users  | 403
-[ RECORD 5 ]-----------+--------
action_month            | 2016-07
mau                     | 429247
new_users               | 314249
repeat_users            | 109213
come_back_users         | 5785
new_repeat_users        | 87932
continuous_repeat_users | 20176
come_back_repeat_users  | 1105
```

上記のコード例では、コード量を抑えるために各属性のユーザー数のみをカウントしています。なお、MAUに占める割合を算出するには、100.0×「指標」÷mauを計算します。

上記コードの出力結果を補助円を用いて表現すると、下図の通り、リピートユーザーの内訳が分かります。左側の円グラフはMAUを大きく3つに分けた状態でサービス利用者の割合を示し、右の補助円グラフでリピートユーザーの内訳を示しています。

図5.2.7.5：MAUを3つの属性に分類し、さらにリピートユーザーを3つの属性に分類



MAUの属性ごとのリピート率を計算する

本項では、MAUをいくつかの属性に分類して、ユーザー数を集計し円グラフで構成を表現しましたが、構成比のみでは「先月の新規登録ユーザーの何割がリピートしているのか」、「カムバックユーザーを増やすキャンペーンを先月実施したが、その効果としてリピートしているのか」などは分かりません。

本項での各出力結果を元に、次図に示すレポートを示すことで、上記の質問に対する回答が可能になります。なお、表では各リピート率を算出するため対応する分子と分母を塗り潰しています。

もちろん、コード5.2.7.2で示したSQLの結果を元に、Excelなどの表計算ソフトで計算すれば、上図に示したレポートは作成可能ですが、レポートを作成する時間を短縮するため、MAUの属性ごとのリピート率をSQLで計算するクエリを、次のコード例に示します。

表5.2.7.1：MAUの内訳とMAU属性ごとのリピート率

	2016年3月	2016年4月	2016年5月	2016年6月	2016年7月	2016年8月
MAU	21,931	35,555	117,819	320,684	429,247	293,566
新規MAU	21,931	27,755	106,249	277,121	314,249	138,463
新規リピートMAU		7,800	7,267	34,593	87,932	78,403
新規リピート率		35.6%	26.2%	32.6%	31.7%	24.9%
リピートMAU		7,800	10,634	40,586	109,213	127,257
既存リピートMAU			3,367	5,590	20,176	47,385
既存リピート率			43.2%	52.6%	49.7%	43.4%
カムバックMAU			936	2,977	5,785	27,846
カムバックリピートMAU				403	1,105	1,469
カムバックリピート率				43.1%	37.1%	25.4%

コード5.2.7.3：MAUの内訳とMAU属性ごとのリピート率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
monthly_user_action AS (
-- コード5.2.7.1を参照
)
, monthly_user_with_type AS (
-- コード5.2.7.1を参照
)
, monthly_users AS (
-- コード5.2.7.2を参照
)
SELECT
    action_month
  , mau
  , new_users
  , repeat_users
  , come_back_users
  , new_repeat_users
  , continuous_repeat_users
  , come_back_repeat_users
  -- ■ PostgreSQL, Redshift, BigQueryの場合、下記を使用
  -- ■ Hiveの場合、NULLIFをCASE式で書き換える
  -- ■ SparkSQLの場合、
  --   LAG関数にフレーム指定 ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING を指定する
  -- 前月の新規ユーザーのうち、当月が新規リピートユーザーであるユーザーの割合
  , 100.0 * new_repeat_users
    / NULLIF(LAG(new_users) OVER(ORDER BY action_month), 0)
    AS priv_new_repeat_ratio

  -- 前月のリピートユーザーのうち、当月が継続リピートユーザーであるユーザーの割合
```

```
  , 100.0 * continuous_repeat_users
    / NULLIF(LAG(repeat_users) OVER(ORDER BY action_month), 0)
    AS priv_continuous_repeat_ratio

  -- 前月のカムバックユーザーのうち、当月がカムバックリピートユーザーであるユーザーの割合
  , 100.0 * come_back_repeat_users
    / NULLIF(LAG(come_back_users) OVER(ORDER BY action_month), 0)
    AS priv_come_back_repeat_ratio

FROM
  monthly_users
ORDER BY
  action_month
;
```

```
-[ RECORD 1 ]---------------+--------
action_month                | 2016-03
mau                         | 21931
new_users                   | 21931
repeat_users                | 0
come_back_users             | 0
new_repeat_users            | 0
continuous_repeat_users     | 0
come_back_repeat_users      | 0
priv_new_repeat_ratio       |
priv_continuous_repeat_ratio |
priv_come_back_repeat_ratio  |
-[ RECORD 2 ]---------------+--------
action_month                | 2016-04
mau                         | 35555
new_users                   | 27755
repeat_users                | 7800
come_back_users             | 0
new_repeat_users            | 7800
continuous_repeat_users     | 0
come_back_repeat_users      | 0
priv_new_repeat_ratio       | 35.6
priv_continuous_repeat_ratio |
priv_come_back_repeat_ratio  |
-[ RECORD 3 ]---------------+--------
action_month                | 2016-05
mau                         | 117819
new_users                   | 106249
repeat_users                | 10634
come_back_users             | 936
new_repeat_users            | 7267
continuous_repeat_users     | 3367
come_back_repeat_users      | 0
```

```
priv_new_repeat_ratio        | 26.2
priv_continuous_repeat_ratio | 43.2
priv_come_back_repeat_ratio  |
-[ RECORD 4 ]----------------+--------
action_month                 | 2016-06
mau                          | 320684
new_users                    | 277121
repeat_users                 | 40586
come_back_users              | 2977
new_repeat_users             | 34593
continuous_repeat_users      | 5590
come_back_repeat_users       | 403
priv_new_repeat_ratio        | 32.6
priv_continuous_repeat_ratio | 52.6
priv_come_back_repeat_ratio  | 43.1
-[ RECORD 5 ]----------------+--------
action_month                 | 2016-07
mau                          | 429247
new_users                    | 314249
repeat_users                 | 109213
come_back_users              | 5785
new_repeat_users             | 87932
continuous_repeat_users      | 20176
come_back_repeat_users       | 1105
priv_new_repeat_ratio        | 31.7
priv_continuous_repeat_ratio | 49.7
priv_come_back_repeat_ratio  | 37.1
```

ワンポイント

本項のレポートは月単位で集計している関係で、月初に登録したユーザーは30日前後に利用しないとリピートユーザーにならないのに対して、月末に登録したユーザーは1日〜2日後に利用すればリピートユーザーとなります。判定期間が平等ではありませんが、概略の推移を確認したり、サービス単位での比較には、有用なレポートとなります。判定期間の差が気になる場合は、独自に定義を設定し集計しましょう。

関連項目 | 3-4-3 条件のフラグを0と1で表現する

5-2-8 訪問種別を定義して成長指数を集計する

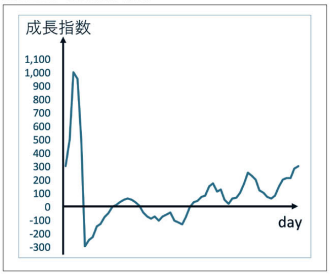
SQL CASE式、LAG関数、SUM(CASE～)　分析 成長指数

サービス運営では、ユーザー登録をはじめ、ユーザーの継続利用やリテンションのための施策を担当し、サービス成長を加速させることを担当するチームがあります。ユーザー獲得チームまたはグロースハックチームなどと呼ばれます。本項では、サービスの成長を指標化したり、グロースハックチームの成果を指標化する方法の１つ、成長指数と呼ばれる指標を紹介します。

成長指数

成長指数はユーザーのサービス利用に関する状態変化を数値化して、サービスが成長しているかを示す指標です。この成長指数が１以上の時はサービスが成長している状態を示し、０を下回るとサービスが衰退している状態を意味します。

図 5.2.8.1：成長指数の推移



下表に、サービスの成長指数を算出するために必要となる、サービス利用に関する状態変化のパターンを示します。

表 5.2.8.1：サービス利用に関する状態変化のパターン

パターン	内容
Signup	新規登録し、利用を開始した
Deactivation	アクティブユーザーが非アクティブユーザーになった
Reactivation	非アクティブユーザーがアクティブユーザーに戻った
Exit	退会し利用を停止した

ユーザーのサービス登録からサービス退会までの利用状況から、成長指数を算出するため、どのように各指標を判定するかを下表に示します。

表5.2.8.2：ユーザーの利用と成長指数を算出する際に用いる状態変化の判定

日付	10/1	10/2	10/3	10/4	10/5	10/6
行動	登録					退会
利用有無	○	×	×	○	×	○
判定	Signup	Deactivation		Reactivation	Deactivation	Exit

10/1に登録したので「signup」と判定します。
10/2は利用していないので、アクティブユーザーから非アクティブになったと言うことで「Deactivation」と判定します。
10/3は引き続き利用しなかったので、状態変化がないので、どの指標にも分類されません。
10/4は利用があったので、非アクティブからアクティブユーザーへ状態変化したことで、「Reactivation」と判定します。
10/5は利用していないので、アクティブユーザーから非アクティブになったと言うことで「Deactivation」と判定します。
10/6は利用していますが、この日に退会したことによって、「Reactivation」ではなく「Exit」と判定します。

上記の判定結果を用いて、成長指数を下記の通りに計算します。

成長指数 = Signup + Reactivation - Deactivation - Exit

(+) 5000	Signup	新規ユーザー数
(+) 3000	Reactivation	非アクティブユーザーがアクティブに戻った数
(-) 4000	Deactivation	アクティブユーザーが非アクティブユーザーになった数
(-) 100	Exit	退会数
(+)3900	Growth index	成長指数

「利用するようになったユーザー」と「去っていったユーザー」を集計して、どちらのユーザーが多いのか数値化するものが成長指数です。利用を継続しているユーザーやずっと利用していないユーザー、当日に登録して当日に退会したユーザーは、成長指数の増減には影響しません。この成長指数を集計して、その推移を記録したものが冒頭に紹介した、前図です。

この通り、サービスの成長指数をシンプルに定義することで、マーケティング以外の担当者も理解可能になります。その結果、成長指数を改善するには、下記2点に注力すれば良いと考えるようになります。

1. SignupとReactivationを増やすことを考える。
2. Deactivationを下げることを考える。

成長指数を集計する

成長指数を算出する目的のユーザー状態として、新規登録したか（is_new）、退会したか（is_exit）、その日サービスにアクセスしたか（is_access）、前日にサービスにアクセスしたか（was_access）を、日別に判定するクエリを、下記のコード例に示します。

コード5.2.8.1: 成長指数を算出するためのユーザー状態を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
unique_action_log AS (
  -- 同じ日付のログを2重にカウントしないよう、アクセスログから日付の重複を排除する
  SELECT DISTINCT
      user_id
    , substring(stamp, 1, 10) AS action_date
    -- ■ BigQueryの場合、下記を用いる
    -- , substr(stamp, 1, 10) AS action_date
  FROM
    action_log
)
, mst_calendar AS (
  -- 集計したい期間のカレンダーテーブルを用意する
  -- generate_series等で動的に作成も可能
           SELECT '2016-10-01' AS dt
  UNION ALL SELECT '2016-10-02' AS dt
  UNION ALL SELECT '2016-10-03' AS dt
  -- 省略
  UNION ALL SELECT '2016-11-04' AS dt
)
, target_date_with_user AS (
  -- ユーザーマスタに対して、カレンダーテーブルの全日付をtarget_dateとして付与する
  SELECT
      c.dt AS target_date
    , u.user_id
    , u.register_date
    , u.withdraw_date
  FROM
      mst_users AS u
    CROSS JOIN
      mst_calendar AS c
)
, user_status_log AS (
  SELECT
      u.target_date
    , u.user_id
    , u.register_date
    , u.withdraw_date
    , a.action_date
    , CASE WHEN u.register_date = a.action_date THEN 1 ELSE 0 END AS is_new
    , CASE WHEN u.withdraw_date = a.action_date THEN 1 ELSE 0 END AS is_exit
```

```
    , CASE WHEN u.target_date    = a.action_date THEN 1 ELSE 0 END AS is_access
    , LAG(CASE WHEN u.target_date = a.action_date THEN 1 ELSE 0 END)
      OVER(
        PARTITION BY u.user_id
        ORDER BY u.target_date
        -- ■ SparkSQLの場合、下記フレーム指定を追記する
        -- ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING
      ) AS was_access
  FROM
      target_date_with_user AS u
    LEFT JOIN
      unique_action_log AS a
      ON u.user_id = a.user_id
      AND u.target_date = a.action_date
  WHERE
    -- 集計期間を登録日以降の日付に絞り込む
    u.register_date <= u.target_date
    -- 退会日が入っている場合は、集計期間を退会日以前の日付に絞り込む
    AND (
         u.withdraw_date IS NULL
      OR u.target_date <= u.withdraw_date
    )
)
SELECT
    target_date
  , user_id
  , is_new
  , is_exit
  , is_access
  , was_access
FROM
  user_status_log
;
```

```
 target_date | user_id | is_new | is_exit | is_access | was_access
-------------+---------+--------+---------+-----------+------------
 2016-10-01  | U001    |      1 |       0 |         1 |
 2016-10-02  | U001    |      0 |       0 |         1 |          1
 2016-10-03  | U001    |      0 |       0 |         1 |          1
 2016-10-04  | U001    |      0 |       0 |         1 |          1
 2016-10-05  | U001    |      0 |       0 |         0 |          1
 2016-10-06  | U001    |      0 |       0 |         0 |          0
...
 2016-10-01  | U002    |      1 |       0 |         1 |
 2016-10-02  | U002    |      0 |       0 |         0 |          1
```

次のステップ 日別のユーザー状態が判定できたら、成長指数を計算するためのパターンを計算し、最終的な成長指数を計算します。下記コード例に、signupとreactivation、deactivation、exit、growth indexの各値を日次で計算するクエリを示します。

コード5.2.8.2：日次の成長指数を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
unique_action_log AS (
  -- コード5.2.8.1 参照
)
, mst_calendar AS (
  -- コード5.2.8.1 参照
)
, target_date_with_user AS (
  -- コード5.2.8.1 参照
)
, user_status_log AS (
  -- コード5.2.8.1 参照
)
, user_growth_index AS (
  SELECT
      *
    , CASE
        -- ある日付に新規登録または退会している場合は、signupまたはexitと判定する
        WHEN is_new + is_exit = 1 THEN
          CASE
            WHEN is_new  = 1 THEN 'signup'
            WHEN is_exit = 1 THEN 'exit'
          END
        -- 新規登録も退会もしていない場合は、reactivationまたはdeactivationと判定する
        -- ただし、reactivation, deactivationの定義に当てはまらない場合はNULLとする
        WHEN is_new + is_exit = 0 THEN
          CASE
            WHEN was_access = 0 AND is_access = 1 THEN 'reactivation'
            WHEN was_access = 1 AND is_access = 0 THEN 'deactivation'
          END
        -- ある日付に新規登録と退会を同時に行っている(is_new + is_exit = 2)場合はNULLとする
      END AS growth_index
  FROM
    user_status_log
)
SELECT
    target_date
  , SUM(CASE growth_index WHEN 'signup'       THEN  1 ELSE 0 END) AS signup
  , SUM(CASE growth_index WHEN 'reactivation' THEN  1 ELSE 0 END) AS reactivation
  , SUM(CASE growth_index WHEN 'deactivation' THEN -1 ELSE 0 END) AS deactivation
  , SUM(CASE growth_index WHEN 'exit'         THEN -1 ELSE 0 END) AS exit
    -- 成長指数の定義に基づき値を集計する
  , SUM(
```

```
      CASE growth_index
        WHEN 'signup'       THEN  1
        WHEN 'reactivation' THEN  1
        WHEN 'deactivation' THEN -1
        WHEN 'exit'         THEN -1
        ELSE 0
      END
    ) AS growth_index
FROM
  user_growth_index
GROUP BY
  target_date
ORDER BY
  target_date
;
```

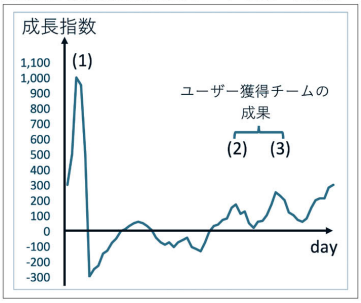
```
 target_date | signup | reactivation | deactivation | exit | growth_index
-------------+--------+--------------+--------------+------+--------------
 2016-10-01  |    234 |            0 |            0 |    0 |          234
 2016-10-02  |    346 |           12 |         -123 |   -3 |          232
 2016-10-03  |    568 |           13 |         -165 |   -2 |          414
 2016-10-04  |    836 |           32 |         -489 |   -3 |          376
 2016-10-05  |    932 |           25 |         -742 |   -5 |          210
 2016-10-06  |    722 |           42 |         -976 |  -12 |         -224
 2016-10-07  |    675 |           37 |         -568 |  -23 |          121
 2016-10-08  |    462 |           24 |         -254 |  -45 |          187
 2016-10-09  |    572 |           57 |         -432 |  -54 |          143
 2016-10-10  |    468 |           63 |         -269 |  -62 |          200
 2016-10-11  |    346 |           24 |         -178 |  -45 |          147
 2016-10-12  |    435 |           37 |         -156 |  -42 |          274
 2016-10-13  |    348 |           39 |         -218 |  -74 |           95
 2016-10-14  |    435 |           26 |         -146 |  -52 |          263
 2016-10-15  |    386 |           28 |         -287 |  -59 |           68
 2016-10-16  |    538 |           31 |         -143 |  -61 |          365
 2016-10-17  |    476 |           28 |         -287 |  -45 |          172
```

上記のコード例では、ユーザー状態から、signup、reactivation、deactivation、exitを判定して、日付ごとに各ユーザー数をカウントしています。ユーザー数をカウントする際、signupとreactivationは正の数、deactivationとexitは負の数としてカウントして、これらの値を合計したものが成長指数となります。

サービスローンチ時からの成長指数の推移

サービスの成長指数がローンチ時からどのように推移するのか、執筆陣の経験を元にまとめます。また、成長指数を向上させるために考えるべきポイントを紹介します。

図 5.2.8.2：成長指数の推移におけるユーザー獲得チームの成果



サービスのローンチ直後は、事前に計画していたプレスリリースや集客施策に加え、新サービスを紹介するメディアに掲載されることなどによって、登録数が一気に伸びます（図①のタイミング）。しかし、しばらくすると8割ぐらいが非アクティブなユーザーとなるでしょう。

ローンチ直後の飛躍的な成長が一段落した時点で、どのような方法でどのようなユーザー層を獲得できるのか、そして、どのようにすれば継続利用を確保できるかを検討します（図②～③のタイミング）。

例えば、ローンチのタイミングに合わせて仕込んだメディアによるユーザー獲得などは、容易に考え付きますが、あくまでも一時的なカンフル剤の役目しか果たせず、成長を継続させるものではないケースが多々あります。

メディアによるユーザー獲得ではなく、サービス（プロダクト）を通じて、ユーザーを獲得し継続して利用してもらうことを重視することで、安定かつ継続した成長が見込めるようになります。継続的に施策を検討し、成長指数を成長させることが、チームにおける目標であると同時に成果となります。

ワンポイント

日常的な利用を期待するサービス（ニュースサイトやソーシャルゲームなど）、何らかの目的が発生した際に利用を期待するサービス（飲食店検索サービスやECサイト）では、利用頻度が異なります。成長指数を計算するために使用する指標の集計は、サービスの特性に応じて、日次や週次、月次などから適切な集計期間を選択しましょう。

関連項目	3-1-1 コード値をラベルに置き換える 3-3-2 グループの中での順序を扱う

5-2-9 指標を改善する手順を身に付ける

本章ではユーザーを把握するためのレポートとSQLを紹介しました。ユーザー把握はゴールではなく、プロセスの1つに過ぎません。サービス提供者にとっては売上やユーザー数の拡大がゴールです。ゴールを達成するための1手段としてユーザー行動を把握して、より確度の高い施策を実施することを検討します。　それでは、指標を向上させるには、どうすればよいのでしょうか。

指標を向上させるためには

下記を行動の基本として考えます。

達成していない人を達成している人に寄せる

その上で、下記の手順で分析を進めます。

1. 達成させたい行動（＝向上させたい指標）を決める
2. 達成する前の行動の中から、着目する行動を決める
3. （2）の注目する行動の有無や回数を集計し、その後に（1）の達成させたい行動を満たしているユーザーの割合を比較する

例えば、翌日継続率を改善したい場合、上記の手順に当てはめると下記になります。

1. 翌日、利用してもらいたい（翌日継続率を改善したい）
2. 翌日利用するかどうかは、その前の登録当日のアクションに着目する
3. 登録当日に行ったアクションの有無で、翌日継続率の割合を比較する

「5-2-3_継続と定着に影響するアクションを集計する」（P.189）では、上記手順を元に集計する方法を紹介しました。達成させたい行動がある場合、「達成する前の行動」に着目するところがポイントです。

デモグラフィック情報から男性より女性の翌日継続率が高いと判明した場合、広告バナーのクリエイティブ、サイトデザイン、サイトコンセプトの見直しなどの施策が検討可能です。しかし、大規模な改修が必要であったり、会員の構成比が変わるだけでは、改善したとは言えません。当然、コンセプトの確認やターゲット層を囲えているかを確認することは重要ですが、指標改善の視点では、やはり「達成する前の行動」に着目することが重要と考えられます。

上記の他にも下記に挙げる視点で集計します。

■ より投稿やコメントしてもらいたい
→フォローされている人数によって違いがあるか
→フォローしている人数によって違いがあるか
→プロフィール画像の登録によって違いがあるか

■ 新規ユーザーのリピート率を改善したい
→登録初月の投稿回数によって違いがあるか
→登録初月にフォローした人数によって違いがあるか
→登録翌日から7日間以内の利用日数に違いがあるか

■ CVRを改善したい
→購入前に詳細ページを見た回数によって違いがあるか
→購入前にお気に入り機能の利用有無によって違いがあるか

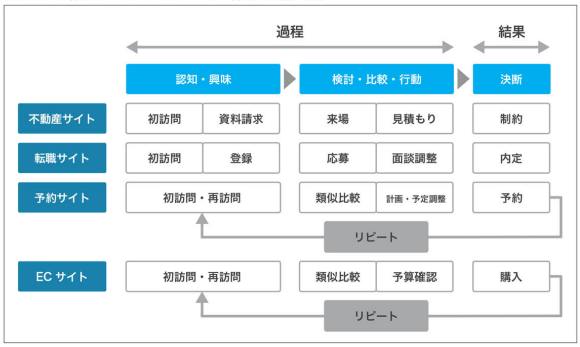
この通り、改善したい指標に対して、さまざまな仮説を立てて検証を進めることで、注力すべき箇所や施策が見えてきます。自ら担当しているサービスに置き換えて考えてみましょう。

5-3
時系列に沿ってユーザー個別のアクションを分析する

本節ではユーザーが起こしたアクションのタイミングに着目して、複数のアクションがどれぐらいの間隔で行われたのか、どれぐらい時間が経過すれば次のアクションに結び付くのかなどを集計します。

例えば、ユーザーが最終的な成果（=結果）に至るまでに、どの程度の検討期間（=過程）を要したのかが分かれば、キャンペーン開始時期の設定に役立てたり、その検討期間が想定以上に長いと判断できたら、より早い判断を促す施策を検討する材料になります。

図 5.3.0.1：各種サービスにおけるユーザー行動の過程と結果



ユーザーのアクションを時間の経過を考慮に入れて可視化することで、ユーザーの成長過程や行動パターンが明らかになり、改善箇所の特定やより精度の高い見積もりが可能になります。

本節では、時間の経過を考慮したレポートやSQLを紹介します。

5-3-1 ユーザーのアクション間隔を集計する

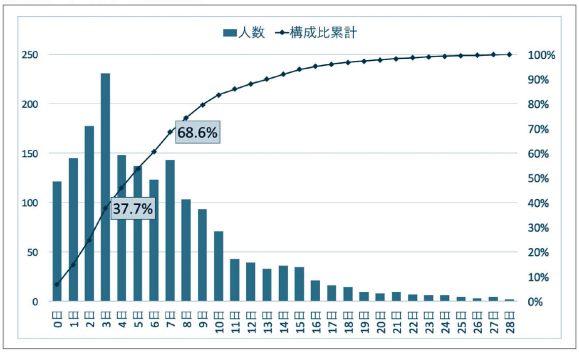
SQL 日付関数、LAG関数　　分析 リードタイム

ユーザーのあるアクションから次のアクションまでの期間を集計することで、問題点を浮き彫りにできたり、新たな施策の検討が可能になります。例えば、資料請求→来場予約→見積もり→成約の流れがあるサービスで、成約までの期間（リードタイム）が想定よりも長期にわたることが分かれば、各シーンの業務フローの課題を整理し、より短い期間で成約まで導くことで売上改善が見込めます。

下図は、ユーザーのリードタイムを集計して、構成比累計を用いてユーザーの行動を示したものです。本項では、元データのパターンに応じて、下図を作成するためにリードタイムを計算する3種類のSQLを紹介します。

なお、下図の作成に当たって、リードタイムを計算した上で、「4-2-2 ABC分析で売れ筋を判別する」（P.115）で説明した構成比累計を計算するSQLを参考にしてください。

図 5.3.1.1：リードタイムと構成比累計



同一レコードにある2つの日付から計算する場合

宿泊施設や飲食店などの予約サイトでは、予約情報を格納するレコードに申込日と宿泊日、訪問日時などが格納されていることがほとんどです。日付同士の引き算やdatediff関数で各日付の差分を計算します。下記コード例に、申込日と宿泊日のリードタイムを計算するクエリを示します。

コード5.3.1.1：申込日と宿泊日のリードタイムを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
reservations(reservation_id, register_date, visit_date, days) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、UNION ALLなどで代用可能
  --    ※3-4-5:擬似的なテーブルを作成する参照
  VALUES
    (1, date '2016-09-01', date '2016-10-01', 3)
  , (2, date '2016-09-20', date '2016-10-01', 2)
  , (3, date '2016-09-30', date '2016-11-20', 2)
  , (4, date '2016-10-01', date '2017-01-03', 2)
  , (5, date '2016-11-01', date '2016-12-28', 3)
)
SELECT
    reservation_id
  , register_date
  , visit_date
  -- ■ PostgreSQL, Redshiftの場合、日付型同士の引き算ができる
  , visit_date::date - register_date::date AS lead_time
  -- ■ BigQueryの場合、date_diff関数が利用できる
  -- , date_diff(date(timestamp(visit_date)), date(timestamp(register_date)), day)
  --   AS lead_time
  -- ■ Hive, SparkSQLの場合、datediff関数が利用できる
  -- , datediff(to_date(visit_date), to_date(register_date)) AS lead_time
FROM
  reservations
;
```

```
 reservation_id | register_date | visit_date | lead_time
----------------+---------------+------------+-----------
              1 | 2016-09-01    | 2016-10-01 |        30
              2 | 2016-09-20    | 2016-10-01 |        11
              3 | 2016-09-30    | 2016-11-20 |        51
              4 | 2016-10-01    | 2017-01-03 |        94
              5 | 2016-11-01    | 2016-12-28 |        57
```

複数のテーブルにある複数の日付から計算する場合

資料請求や見積もり、成約など、複数のステップが存在するケースでは、各データが別テーブルに格納されている場合があります。その際は複数の日付を同一レコードで参照可能にするため、テーブルをJOINして、下記コード例に示す通り、前記のコードと同様に日付の差分を計算します。

コード5.3.1.2：各ステップにおけるリードタイムとトータルのリードタイムを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
requests(user_id, product_id, request_date) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、UNION ALLなどで代用可能
  --    ※3-4-5:擬似的なテーブルを作成する参照
  VALUES
    ('U001', '1', date '2016-09-01')
  , ('U001', '2', date '2016-09-20')
  , ('U002', '3', date '2016-09-30')
  , ('U003', '4', date '2016-10-01')
  , ('U004', '5', date '2016-11-01')
)
, estimates(user_id, product_id, estimate_date) AS (
  VALUES
    ('U001', '2', date '2016-09-21')
  , ('U002', '3', date '2016-10-15')
  , ('U003', '4', date '2016-10-15')
  , ('U004', '5', date '2016-12-01')
)
, orders(user_id, product_id, order_date) AS (
  VALUES
    ('U001', '2', date '2016-10-01')
  , ('U004', '5', date '2016-12-05')
)
SELECT
      r.user_id
    , r.product_id
    -- ■ PostgreSQL, Redshiftの場合、日付型同士の引き算ができる
    , e.estimate_date::date - r.request_date::date  AS estimate_lead_time
    , o.order_date::date    - e.estimate_date::date AS order_lead_time
    , o.order_date::date    - r.request_date::date  AS total_lead_time
    -- ■ BigQueryの場合、date_diff関数が利用できる
    --, date_diff(date(timestamp(e.estimate_date)),
    --    date(timestamp(r.request_date)) , day)
    --  AS estimate_lead_time
    --, date_diff(date(timestamp(o.order_date)),
    --    date(timestamp(e.estimate_date)), day)
    --  AS order_lead_time
    --, date_diff(date(timestamp(o.order_date)),
    --    date(timestamp(r.request_date)) , day)
    --  AS total_lead_time
```

```
    -- ■ Hive, SparkSQLの場合、datediff関数が利用できる
    --, datediff(to_date(e.estimate_date), to_date(r.request_date))
    --  AS estimate_lead_time
    --, datediff(to_date(o.order_date)   , to_date(e.estimate_date))
    --  AS order_lead_time
    --, datediff(to_date(o.order_date)   , to_date(r.request_date))
    --  AS total_lead_time
FROM
    requests AS r
  LEFT OUTER JOIN
    estimates AS e
    ON  r.user_id    = e.user_id
    AND r.product_id = e.product_id
  LEFT OUTER JOIN
    orders AS o
    ON  r.user_id    = o.user_id
    AND r.product_id = o.product_id
;
```

```
 user_id | product_id | estimate_lead_time | order_lead_time | total_lead_time
---------+------------+--------------------+-----------------+-----------------
 U001    | 1          |                    |                 |
 U001    | 2          |                  1 |              10 |              11
 U002    | 3          |                 15 |                 |
 U003    | 4          |                 14 |                 |
 U004    | 5          |                 30 |               4 |              34
```

同一テーブルの異なるレコードにある日付から計算する場合

ECサイトで購入から次回購入までの間隔を知りたい場合、データが同じテーブルに存在するので、LAG関数を用いて日付の差分を計算します。

コード5.3.1.3：前回購入日からの日数を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
purchase_log(user_id, product_id, purchase_date) AS
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、UNION ALLなどで代用可能
  --    ※3-4-5:擬似的なテーブルを作成する参照
  VALUES
    ('U001', '1', '2016-09-01')
  , ('U001', '2', '2016-09-20')
  , ('U002', '3', '2016-09-30')
  , ('U001', '4', '2016-10-01')
  , ('U002', '5', '2016-11-01')
)
```

```
SELECT
    user_id
  , purchase_date
  -- ■ PostgreSQL, Redshiftの場合、日付型同士の引き算ができる
   , purchase_date::date
     - LAG(purchase_date::date)
       OVER(
         PARTITION BY user_id
         ORDER BY purchase_date
         )
     AS lead_time
  -- ■ BigQueryの場合、date_diff関数が利用できる
  -- , date_diff(date(timestamp(purchase_date)),
  --     LAG(date(timestamp(purchase_date)))
  --       OVER(PARTITION BY user_id ORDER BY purchase_date)
  --   , day) AS lead_time
  -- ■ Hiveの場合、datediff関数が利用できる
  -- , datediff(to_date(purchase_date),
  --     LAG(to_date(purchase_date))
  --       OVER(PARTITION BY user_id ORDER BY purchase_date))
  --   AS lead_time
  -- ■ SparkSQLの場合、datediff関数に加え、LAG関数にフレーム指定が必要
  --, datediff(to_date(purchase_date),
  --    LAG(to_date(purchase_date))
  --      OVER(PARTITION BY user_id ORDER BY purchase_date
  --        ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING))
  --  AS lead_time
FROM
  purchase_log
;
```

```
 user_id | purchase_date | lead_time
---------+---------------+-----------
 U001    | 2016-09-01    |
 U001    | 2016-09-20    |        19
 U001    | 2016-10-01    |        11
 U002    | 2016-09-30    |
 U002    | 2016-11-01    |        32
```

ワンポイント

リードタイムを集計したら、次のステップとしてユーザーのデモグラフィック情報を用いて比較しましょう。ECサイトであれば、都心部よりも地方の方が前回購入日からのリードタイムが短かったり、年齢別区分で異なるなど、さまざまな傾向が得られるかもしれません。

関連項目	3-2-5 日付・時刻を計算する 3-4-2 複数のテーブルを横に並べる

5-3-2 カート追加後に購入されているかを把握する

SQL CASE式、SUM関数、AVG関数　　分析 カゴ落ち率、カート放棄率

カートに商品を追加した後に購入せずに離脱し、購入に至っていない状況を「カゴ落ち」または「カート放棄」と呼びます。カートに商品を追加したにも関わらず、購入に至っていないケースは、下記に挙げる理由が考えられます。

- 商品購入までの手続きに何らかの問題がある
- 想定外のコスト（送料や代引き手数料）の提示に難色を示した
- ブックマーク代わりにカートを利用し、他サイトでより安い商品を購入してそのままになっている

カートに商品を追加しているにも関わらず、購入に至らない件数がどれぐらいあるのか、また購入される場合は何時間以内に購入されているのかを把握するためのレポートを下図に示します。

図5.3.2.1：カート追加からの経過時間別購入数の推移

		購入数・率				
日付	カート追加数	1時間以内	6時間以内	24時間以内	48時間以内	カゴ落ち
2016/10/14	4,301	2,988 69.5%	3,101 72.1%	3,129 72.8%	3,316 77.1%	985 22.9%
2016/10/15	4,539	2,987 65.8%	3,098 68.3%	3,125 68.8%	3,413 75.2%	1,126 24.8%

本書では、カートに追加されて48時間以内に購入されていない商品を「カゴ落ち」と呼び、カートに追加された商品数を母数とした割合をカゴ落ち率と定義します。下記データ例に、カゴ落ち率を集計する際に利用するアクションログを示します。

データ5.3.2.1：アクションログ(action_log)

```
     dt     | session | user_id |  action  | products |       stamp
------------+---------+---------+----------+----------+---------------------
 2016-10-14 | A       | U001    | add_cart | 1        | 2016-10-14 18:00:00
 2016-10-14 | A       | U001    | add_cart | 2        | 2016-10-14 18:01:00
 2016-10-14 | A       | U001    | add_cart | 3,4      | 2016-10-14 18:02:00
 2016-10-14 | A       | U001    | purchase | 1,2,3,4  | 2016-10-14 18:10:00
 2016-10-14 | B       | U002    | add_cart | 1        | 2016-10-14 19:00:00
```

```
2016-10-14 | B       | U002    | purchase | 1        | 2016-10-14 20:00:00
2016-10-14 | B       | U002    | add_cart | 2        | 2016-10-14 20:30:00
2016-10-15 | C       | U001    | add_cart | 4        | 2016-10-15 12:00:00
2016-10-15 | C       | U001    | add_cart | 5        | 2016-10-15 12:00:00
2016-10-15 | C       | U001    | add_cart | 6        | 2016-10-15 12:00:00
2016-10-15 | D       | U002    | purchase | 2        | 2016-10-15 13:00:00
2016-10-15 | D       | U001    | purchase | 5,6      | 2016-10-15 15:00:00
```

前記データ例でのsession:Aのレコードは、カートに1商品ずつ2回追加し、その後2つの商品を一度でカートに追加し、カート内の4商品を1回のアクションで購入した状態を表します。

下記コード例では、1レコードにカンマ区切りで複数の商品が含まれているデータを、1レコード1商品に整形して、商品ごとにカート追加された時刻と購入された時刻を算出します。同時に、カート追加から購入までに要した時間も計算しています。

コード5.3.2.1：商品ごとに、カート追加された時刻と購入された時刻を算出するクエリ

PostgreSQL Hive BigQuery SparkSQL

```
WITH
row_action_log AS (
  SELECT
    dt
  , user_id
  , action
  -- カンマ区切りのproduct_idリストを行に展開する
  -- ■ PostgreSQLの場合、regexp_split_to_table関数を利用できる
  , regexp_split_to_table(products, ',') AS product_id
  -- ■ Hive, BigQuery, SparkSQLの場合、FROM句にて展開し、product_idを取得
  -- , product_id
  , stamp
  FROM
    action_log
  -- ■ BigQueryの場合は下記を追加
  -- CROSS JOIN unnest(split(products, ',')) AS product_id
  -- ■ Hive, SparkSQLの場合は下記を追加
  -- LATERAL VIEW explode(split(products, ',')) e AS product_id
)
, action_time_stats AS (
  -- ユーザーと商品の組み合わせごとに、カート追加と購入の時刻を取得する
  SELECT
    user_id
  , product_id
  , MIN(CASE action WHEN 'add_cart' THEN dt END) AS dt
  , MIN(CASE action WHEN 'add_cart' THEN stamp END) AS add_cart_time
  , MIN(CASE action WHEN 'purchase' THEN stamp END) AS purchase_time
  -- ■ PostgreSQLの場合、timestmap型に変換して差分を取り、EXTRACT(epoc ~)で秒数に変換
  , EXTRACT(epoch from
        MIN(CASE action WHEN 'purchase' THEN stamp::timestamp END)
```

```
      - MIN(CASE action WHEN 'add_cart' THEN stamp::timestamp END))
  -- ■ BigQueryの場合､unix_seconds関数で秒単位のUNIX時間に変換して差分を取る
  -- ,   MIN(CASE action WHEN 'purchase' THEN unix_seconds(timestamp(stamp)) END)
  --   - MIN(CASE action WHEN 'add_cart' THEN unix_seconds(timestamp(stamp)) END)
  -- ■ Hive, SparkSQLの場合､Hive, SparkSQLの場合、
  -- unix_timestamp関数で秒単位のUNIX時間に変換して差分を取る
  -- ,   MIN(CASE action WHEN 'purchase' THEN unix_timestamp(stamp) END)
  --   - MIN(CASE action WHEN 'add_cart' THEN unix_timestamp(stamp) END)
    AS lead_time

  FROM
    row_action_log
  GROUP BY
    user_id, product_id
)
SELECT
    user_id
  , product_id
  , add_cart_time
  , purchase_time
  , lead_time
FROM
  action_time_stats
ORDER BY
  user_id, product_id
;
```

```
 user_id | product_id |    add_cart_time    |    purchase_time    | lead_time
---------+------------+---------------------+---------------------+-----------
 U001    | 1          | 2016-10-14 18:00:00 | 2016-10-14 18:10:00 |       600
 U001    | 2          | 2016-10-14 18:01:00 | 2016-10-14 18:10:00 |       540
 U001    | 3          | 2016-10-14 18:02:00 | 2016-10-14 18:10:00 |       480
 U001    | 4          | 2016-10-15 12:00:00 |                     |
 U001    | 5          | 2016-10-15 12:00:00 | 2016-10-15 15:00:00 |     10800
 U001    | 6          | 2016-10-15 12:00:00 | 2016-10-15 15:00:00 |     10800
 U002    | 1          | 2016-10-14 19:00:00 | 2016-10-14 20:00:00 |      3600
 U002    | 2          | 2016-10-14 20:30:00 | 2016-10-15 13:00:00 |     59400
```

次のステップ 最後にカート追加から購入までに要した時間で分類し、購入された商品数と購入率を計算します。

コード5.3.2.2：カート追加後ｎ時間以内に購入された商品数と購入率を計算するクエリ

```
WITH
row_action_log AS (
  -- コード5.3.2.1参照
```

```
)
, action_time_stats AS (
  -- コード5.3.2.1参照
)
, purchase_lead_time_flag AS (
SELECT
    user_id
  , product_id
  , dt
  , CASE WHEN lead_time <=  1 * 60 * 60 THEN 1 ELSE 0 END AS purchase_1_hour
  , CASE WHEN lead_time <=  6 * 60 * 60 THEN 1 ELSE 0 END AS purchase_6_hours
  , CASE WHEN lead_time <= 24 * 60 * 60 THEN 1 ELSE 0 END AS purchase_24_hours
  , CASE WHEN lead_time <= 48 * 60 * 60 THEN 1 ELSE 0 END AS purchase_48_hours
  , CASE
      WHEN lead_time IS NULL OR NOT (lead_time <= 48 * 60 * 60) THEN 1
      ELSE 0
    END AS not_purchase
FROM
  action_time_stats
)
SELECT
    dt
  , COUNT(*) AS add_cart
  , SUM(purchase_1_hour  ) AS purchase_1_hour
  , AVG(purchase_1_hour  ) AS purchase_1_hour_rate
  , SUM(purchase_6_hours ) AS purchase_6_hours
  , AVG(purchase_6_hours ) AS purchase_6_hours_rate
  , SUM(purchase_24_hours) AS purchase_24_hours
  , AVG(purchase_24_hours) AS purchase_24_hours_rate
  , SUM(purchase_48_hours) AS purchase_48_hours
  , AVG(purchase_48_hours) AS purchase_48_hours_rate
  , SUM(not_purchase     ) AS not_purchase
  , AVG(not_purchase     ) AS not_purchase_rate
FROM
  purchase_lead_time_flag
 GROUP BY
   dt
;
```

```
-[ RECORD 1 ]----------+----------------------
dt                     | 2016-10-14
add_cart               | 4301
purchase_1_hour        | 2988
purchase_1_hour_rate   | 69.5
purchase_6_hours       | 3101
purchase_6_hours_rate  | 72.1
purchase_24_hours      | 3129
purchase_24_hours_rate | 72.8
```

```
purchase_48_hours      | 3316
purchase_48_hours_rate | 77.1
not_purchase           | 985
not_purchase_rate      | 22.9
-[ RECORD 2 ]----------+----------------------
dt                     | 2016-10-15
add_cart               | 4539
purchase_1_hour        | 2987
purchase_1_hour_rate   | 65.8
purchase_6_hours       | 3098
purchase_6_hours_rate  | 68.3
purchase_24_hours      | 3125
purchase_24_hours_rate | 68.8
purchase_48_hours      | 3413
purchase_48_hours_rate | 75.2
not_purchase           | 1126
not_purchase_rate      | 24.8
```

上記コード例の結果はカラム数が多いため、カラムを縦に表示する拡張表示形式で出力しています。

ワンポイント

カゴ落ち商品があるユーザーに対して、メルマガやプッシュ通知などで、クーポンや値下げ情報を送信して、購入を促進する工夫を凝らしてみましょう。

関連項目	3-2-5 日付・時刻を計算する

5-3-3 登録からの売上を経過日数別に集計する

SQL CROSS JOIN, CASE式, AVG関数　　分析 ARPU, ARPPU, LTV

サービス運営側はユーザーを獲得するため、リスティング広告やアフィリエイトなど費用を費やして集約を行います。ユーザーの登録からの経過日数に応じて売上を集計することで、集客に適切な費用を見積もったり、1登録に要するコストが妥当かどうか判断することができます。

本項では、ユーザー登録を月別に集計し、n日経過時点での1ユーザー当たりの売上金額を集計する方法を紹介します。

表5.3.3.1：月別登録者と経過日数別の売上

登録月	登録数	一人あたり 30日売上額	一人あたり 45日売上額	一人あたり 60日売上額
2016年1月	12,937	489	783	859
2016年2月	12,101	529	771	900

本項のクエリを設計する基本的な考え方は、これまで説明した継続率や定着率を算出するクエリと同様です。ここでは、0と1のアクションフラグではなく、売上額を集計する点に注意してクエリを考えてみましょう。まず、集計の元となる指標マスタを作成し、ユーザーマスタの登録日と購入ログを結合し、集計に必要なデータを揃えます。

コード5.3.3.1:ユーザーごとに登録からの経過日数別売上を算出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
index_intervals(index_name, interval_begin_date, interval_end_date) AS (
  -- ■ PostgreSQLの場合、VALUES句で一時テーブルを作成できる
  -- ■ Hive, Redshift, BigQuery, SparkSQLの場合、UNION ALLなどで代用可能
  --    ※3-4-5:擬似的なテーブルを作成する参照
  VALUES
    ('30 day sales amount'   ,  0, 30)
  , ('45 day sales amount'   ,  0, 45)
  , ('60 day sales amount'   ,  0, 60)
)
, mst_users_with_base_date AS (
  SELECT
      user_id

    -- 基準日として登録日を用いる
    , register_date AS base_date

    -- 初回購入日を基準にしたい場合は、base_dateの定義を書き換える
    -- , first_purchase_date AS base_date
  FROM
    mst_users
)
, purchase_log_with_index_date AS (
  SELECT
      u.user_id
    , u.base_date
    -- アクションの日付と、ログ全体の最新日付を日付型として取得
    , CAST(p.stamp AS date) AS action_date
    , MAX(CAST(p.stamp AS date)) OVER() AS latest_date
    , substring(p.stamp, 1, 7) AS month
    -- ■ BigQueryの場合、一度タイムスタンプ型に変換してから日付型に変換
    -- , date(timestamp(p.stamp)) AS action_date
    -- , MAX(date(timestamp(p.stamp))) OVER() AS latest_date
    -- , substr(p.stamp, 1, 7) AS month

    , i.index_name
    -- 指標の対象期間の開始日と終了日を計算する
    -- ■ PostgreSQLの場合は下記を使用
    , CAST(u.base_date::date + '1 day'::interval * i.interval_begin_date AS date)
      AS index_begin_date
    , CAST(u.base_date::date + '1 day'::interval * i.interval_end_date   AS date)
      AS index_end_date
```

```
    -- ■ Redshiftの場合は下記を使用
    -- , dateadd(day, r.interval_begin_date, u.base_date::date) AS index_begin_date
    -- , dateadd(day, r.interval_end_date  , u.base_date::date) AS index_end_date
    -- ■ BigQueryの場合は下記を使用
    -- , date_add(CAST(u.base_date AS date), interval r.interval_begin_date day)
    --   AS index_begin_date
    -- , date_add(CAST(u.base_date AS date), interval r.interval_end_date day)
    --   AS index_end_date
    -- ■ Hive, SparkSQLの場合は下記を使用
    -- , date_add(CAST(u.base_date AS date), r.interval_begin_date)
    --   AS index_begin_date
    -- , date_add(CAST(u.base_date AS date), r.interval_end_date)
    --   AS index_end_date
    , p.amount
  FROM
      mst_users_with_base_date AS u
    LEFT OUTER JOIN
      action_log AS p
      ON u.user_id = p.user_id
      AND p.action = 'purchase'
    CROSS JOIN
      index_intervals AS i
)
SELECT *
FROM
  purchase_log_with_index_date
;
```

```
-[ RECORD 1 ]----+-------------------
user_id          | U001
base_date        | 2016-10-01
action_date      | 2016-11-04
latest_date      | 2016-11-04
month            | 2016-11
index_name       | 30 day sales amount
index_begin_date | 2016-10-01
index_end_date   | 2016-10-31
amount           | 1000
-[ RECORD 2 ]----+-------------------
user_id          | U001
base_date        | 2016-10-01
action_date      | 2016-11-03
latest_date      | 2016-11-04
month            | 2016-11
index_name       | 30 day sales amount
index_begin_date | 2016-10-01
index_end_date   | 2016-10-31
amount           | 2000
-[ RECORD 3 ]----+-------------------
```

```
user_id          | U001
base_date        | 2016-10-01
action_date      | 2016-11-03
latest_date      | 2016-11-04
month            | 2016-11
index_name       | 30 day sales amount
index_begin_date | 2016-10-01
index_end_date   | 2016-10-31
amount           | 2000
```

次のステップ 続いて、これまでと同様に、対象期間にログの日付が含まれているかどうかで場合分けをします。下記に示すコード例で、月次レポートとしてまとめるため、登録月と指標で集約し、登録後の経過日数別の売上を集計します。ここで0と1のアクションフラグではなく、購入額を返している点に注意してください。

コード5.3.3.2：月別登録者数と経過日数別の売上を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
index_intervals(index_name, interval_begin_date, interval_end_date) AS (
  -- コード5.3.3.1参照
)
, mst_users_with_base_date AS (
  -- コード5.3.3.1参照
)
, purchase_log_with_index_date AS (
  -- コード5.3.3.1参照
)
, user_purchase_amount AS (
  SELECT
      user_id
    , month
    , index_name
      -- 3. 指標の対象期間に購入した金額をユーザごとに合計する
    , SUM(
        -- 1. 指標の対象期間の終了日が、ログの最新日付以前に含まれるか確認
        CASE WHEN index_end_date <= latest_date THEN
          -- 2. 指標の対象期間に購入している場合は購入額、それ以外は0を返す
          CASE
            WHEN action_date BETWEEN index_begin_date AND index_end_date THEN amount
            ELSE 0
          END
        END
      ) AS index_date_amount
  FROM
    purchase_log_with_index_date
  GROUP BY
```

```
    user_id, month, index_name, index_begin_date, index_end_date
)
SELECT
    month
  -- 登録者数をカウントする
  -- ただし、指標の対象期間の終了日がログの最新日付以前に含まれないユーザーは除く
  , COUNT(index_date_amount) AS users
  , index_name
  -- 指標の対象期間内に購入したユーザー数
  , COUNT(CASE WHEN index_date_amount > 0 THEN user_id END) AS purchase_uu
  -- 指標の対象期間内の合計売上
  , SUM(index_date_amount) AS total_amount
  -- 登録者あたりの平均売上
  , AVG(index_date_amount) AS avg_amount
FROM
  user_purchase_amount
GROUP BY
  month, index_name
ORDER BY
  month, index_name
;
```

```
month   | users |     index_name      | purchase_uu | total_amount | avg_amount
---------+-------+---------------------+-------------+--------------+-----------
 2016-10  | 12937 | 30 day sales amount |        3461 |      6326193 |     489.11
 2016-10  | 12937 | 45 day sales amount |        4211 |     10129671 |     783.23
 2016-10  | 12937 | 60 day sales amount |        4565 |     11112883 |     859.41
 2016-11  | 12101 | 30 day sales amount |        3523 |      6401429 |     529.13
 2016-11  | 11436 | 45 day sales amount |        3983 |      8817156 |     771.61
 2016-11  | 10623 | 60 day sales amount |        4311 |      9560700 |     900.33
```

(注) avg_amountは登録者1人当たりの売上を示す

本項では、広告でのユーザー獲得の費用を検証するため、登録数を母数にして1人あたりの売上を集計していますが、母数を変更することで異なる指標として活用できます。サービスの利用者数を母数にすると、「ユーザー1人あたりの平均売上金額（ARPU：Average Revenue Per User）」となり、課金ユーザー数を母数にすると「課金ユーザー1人あたりの平均売上金額（ARPPU：Average Revenue Per Paid User）」となります。

両者は似た名称ですが、意味合いが違うので注意が必要です。単純に利用者あたりの課金額を示す場合はARPUを使用します。フリーミアムモデル（例：ソーシャルゲーム）で無課金ユーザーと課金ユーザーを区別する必要がある場合は、後者のARPPUを使用します。

LTV(顧客生涯価値)について

本項で算出する「n日経過時点での1ユーザー当たりの売上金額」と類似する指標として、LTV（Life Time Value：顧客生涯価値）と呼ばれるものがあります。LTVは顧客が生涯を通じて、どれぐらい利益に貢献するかを算出したものです。LTVはCPA（Cost Per Acquisition：顧客獲得単価）を設定・管理する上で重要な数値となります。LTVは下記の式で求めるのが一般的です。

LTV（顧客生涯価値）＝年間取引額×収益率×継続年数（滞在期間）

LTVはユーザーごとに算出することも可能ですが、実務では顧客全体をベースとして捉え算出します。また、収益率を厳密に算出できるケースも稀です。インターネットでの各種サービスに携わっている執筆陣がLTVを求められるケースとして、広告を出稿する際の予算設定、適切なCPAであるかどうかのチェックを目的とすることがほとんどです。

インターネットにおけるサービスはサイクルが非常に早く、LTVのように継続年数を5年～10年のスパンで考えるのは適切ではない場合があり、登録後数ヶ月～半年の売上を元にCPAの設定を求められるため、本項記載のレポートを作成しています。

ワンポイント

本稿のSQLはユーザーの登録日基準で1人当たりのn日売上額を集計しているものとなっていますが、アフィリエイトの成果地点を登録ではなく、初回売上時に設定している場合は、SQLのWITH句にあるmst_users_with_base_dateテーブルのbase_dateカラムを、ユーザー別の初回売上日を基準に書き換えることで、より広告担当者の運用に即したレポートになります。

関連項目 | 3-2-5 日付・時刻を計算する

Chapter 6

Webサイトでの行動を把握するためのデータ抽出

Webサイト特有の指標とそのデータを抽出するSQLを紹介します。
Webサイトの分析ではアクセス解析ツールの仕様や機能によっては目的とする分析が不可能なケースもあります。
自らのSQLで各種データを集計する方法を身に付け、さまざまのデータと組み合わせることでより詳細な分析が可能になります。

6-1 サイト全体の特徴・傾向を見つける

本節では、Webサイトにおけるサイト全体の特徴や傾向を見つけるためのレポートとSQLを紹介します。ここで紹介するレポートはアクセス解析ツールを導入しているサービスでは、既に把握している数値かもしれません。しかしながら、そうした状態であってもビッグデータ基盤から同様の数値を取得することには意味があります。

アクセス解析ツールで集計された結果やレポートは多少のフィルタリングは可能かもしれませんが、提供されている機能以外のレポートは作成できません。レポートの設計がアクセス解析ツールの機能に制限されてしまいます。その制限を越えて自由にレポートを設計し集計することが、ビッグデータ基盤を導入する1つのメリットです。

アクセス解析ツールで提供される結果をさらに複雑な条件でドリルダウンして詳細を探るには、アクセス解析ツールが集計する項目は最低限取得できる状態にしておく必要があります。その上で、アクセス解析ツールには用意されていない、ユーザーデータや商品データなどの業務データと組み合わせることで、今までにはないレポートの作成が可能になります。

本節では、ECサイトを例題にしてレポートやクエリを紹介します。

サンプルデータ

本節では、下記に示すサンプルデータ（アクセスログ・購入ログ）のテーブルを元に解説します。

データ6.1.0.1：アクセスログ(access_log／拡張表示)

```
-[ RECORD 1 ]-+------------------------------------------------------------
stamp         | 2016-10-01 12:00:00
short_session | 0CVKaz
long_session  | 1CwlSX
url           | http://www.example.com/?utm_source=google&utm_medium=search
referrer      | http://www.google.co.jp/xxx
-[ RECORD 2 ]-+------------------------------------------------------------
stamp         | 2016-10-01 13:00:00
short_session | 0CVKaz
long_session  | 1CwlSX
url           | http://www.example.com/detail?id=1
referrer      |
-[ RECORD 3 ]-+------------------------------------------------------------
stamp         | 2016-10-01 13:00:00
short_session | 1QceiB
```

```
long_session  | 3JMO2k
url           | http://www.example.com/list/cd
referrer      |
...
```

データ6.1.0.2：購入ログ(purchase_log)

```
         stamp         | short_session | long_session | purchase_id | amount
-----------------------+---------------+--------------+-------------+--------
 2016-10-01 15:00:00 | 0CVKaz         | 1CwlSX       |           1 |   1000
 2016-10-01 16:00:00 | 2is8PX         | 7Dn99b       |           2 |   1000
 2016-10-01 20:00:00 | 2is8PX         | 7Dn99b       |           3 |   1000
 2016-10-02 14:00:00 | 2is8PX         | 7Dn99b       |           4 |   1000
...
```

6-1-1 日次の訪問者数・訪問回数・ページビューを集計する

SQL COUNT関数、COUNT(DISTINCT～)　　分析 訪問者数、訪問回数、ページビュー

Webサイトでは、訪問者数や訪問回数、ページビューの集計は基本です。それぞれの指標が何を集計しているのか理解し、以降の集計に活かしましょう。下表に訪問者数、訪問回数、ページビューそれぞれの集計する定義を示します。

表6.1.1.1：指標の定義

指標	集計する内容
訪問者数	ブラウザを閉じても消えないCookieのユニーク数 ※1人のユーザーが1日で3回サイトを訪問しても1として計算
訪問回数	ブラウザを閉じたら消えるCookieのユニーク数 ※1人のユーザーが1日で3回サイトを訪問したら3として計算
ページビュー	ページを表示したログの行数

上記定義に加えて、本項では「1訪問あたりのページビュー」を日次で集計し、下表に示すデータを取得するSQLを紹介します。

表6.1.1.2：日次のアクセスデータ

日付	訪問者数	訪問回数	ページビュー	1訪問あたりのページビュー
2016/6/1	87,501	102,800	506,497	4.9
2016/6/2	89,928	105,901	508,689	4.8
2016/6/3	83,914	99,713	491,838	4.9
2016/6/4	65,027	80,679	434,543	5.4

コード6.1.1.1：日次のアクセスデータを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、substringで日付部分を抽出する
    substring(stamp, 1, 10) AS dt
  -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、substrが利用できる
  -- substr(stamp, 1, 10) AS dt

  -- 訪問者数を計算
  , COUNT(DISTINCT long_session) AS access_users
  -- 訪問回数を計算
  , COUNT(DISTINCT short_session) AS access_count
  -- ページビューを計算
  , COUNT(*) AS page_view

  -- 1ユーザあたりのページビュー数
  -- ■ PostgreSQL, Redshift, BigQuery, SparkSQLの場合、NULLIF関数が使用できる
   , 1.0 * COUNT(*) / NULLIF(COUNT(DISTINCT long_session), 0) AS pv_per_user
  -- ■ Hiveの場合、NULLIF関数の代わりにCASE式を用いる
  -- , 1.0 * COUNT(*)
  --   / CASE
  --       WHEN COUNT(DISTINCT long_session) <> 0 THEN COUNT(DISTINCT long_session)
  --     END
  -- AS pv_per_user
FROM
  access_log
GROUP BY
  -- ■ PostgreSQL, Redshift, BigQueryの場合、
  --   SELECT句で定義した別名をGROUP BYに指定できる
  dt
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、
  --   SELECT句で別名を指定する前の式をGROUP BYに指定できる
  -- substring(stamp, 1, 10)
ORDER BY
  dt
;
```

```
     dt     | access_users | access_count | page_view | pv_per_user
------------+--------------+--------------+-----------+-------------
 2016-06-01 |        87501 |       102800 |    506497 |        4.92
 2016-06-02 |        89928 |       105901 |    508689 |        4.80
 2016-06-03 |        83914 |        99713 |    491838 |        4.93
```

サイトにログイン機能を提供している場合は、これらに加えて、ログインUUと非ログインUUも同時に集計しましょう。ログインUUと非ログインUUの集計に関しては、前述の「5-1-1 ユーザーのアクション数を集計する」(P.133) を参考にしてください。

ワンポイント

必要とされるレポートがページビューだけであっても、それ以外の指標を集計する際に、別途SQLを新たに作成して実行し、新たにレポートを作成するのでは時間が掛かり面倒です。1つのSQLを実行し予めまとめて取得しておきましょう。

関連項目 | 5-1-1 ユーザーのアクション数を集計する

6-1-2 ページ毎の訪問者数・訪問回数・ページビューを集計する

SQL 正規表現、URL関数

ログデータにはURLが含まれていることが多く、そのURLを集計して、各ページの訪問回数やページビュー等を集計します。

しかし、ログデータのURLには、画面制御用のパラメーターや広告計測用、クリック数計測用等の各種パラメーター（例：http://○○.com?utm_source=hoge）などが含まれるケースが多く、同一ページに複数のURLが存在するため、単純にページビューを算出できません。広告からの流入数を調査するのであれば、各パラメーター別に集計すべきですが、単純に特定ページが何回閲覧されたかを調査する場合は、各種のパラメーターは不要な情報です。

本項では、用途に応じて、URLから必要な情報を取り出して、集計する方法を紹介します。

URL別に集計する

まずは、基本となるURL別に集計するクエリを紹介します。

コード6.1.2.1：URL別に集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    url
  , COUNT(DISTINCT short_session) AS access_count
  , COUNT(DISTINCT long_session) AS access_users
  , COUNT(*) AS page_view
FROM
  access_log
GROUP BY
  url
;
```

```
                url                 | access_count | access_users | page_view
------------------------------------+--------------+--------------+----------
 http://www.example.com/            |         8925 |         7052 |     10592
 http://www.example.com/detail?id=1 |         5624 |         4568 |      8956
 http://www.example.com/detail?id=2 |         4351 |         3568 |      6578
 http://www.example.com/detail?id=3 |         3648 |         3164 |      4598
 http://www.example.com/list/cd     |         6156 |         4156 |      8435
 http://www.example.com/list/dvd    |         5198 |         3157 |      7621
 http://www.example.com/list/newly  |         6289 |         3916 |      8165
…
```

パス別に集計する

上述のURL別に集計するSQLの出力結果には、URLにクエリパラメーターが含まれており、各ページ毎に集計が可能ですが、集計の粒度が細かすぎます。各ページの集計結果ではなく、「/detail?id=**」を「詳細ページ」としてまとめられるように、クエリパラメーターを省略してパス別に集計しましょう。下記コード例にクエリを示します。

コード6.1.2.2：パス別に集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_path AS (
  -- URLからパスを抽出する
  SELECT *
    -- ■ PostgreSQLの場合、正規表現でパス部分を抽出する
     , substring(url from '//[^/]+([^?#]+)') AS url_path
    -- ■ Redshiftの場合、regexp_substr関数とregexp_replace関数を組み合わせる
    -- , regexp_replace(regexp_substr(url, '//[^/]+[^?#]+'), '//[^/]+', '') AS url_path
    -- ■ BigQueryの場合、正規表現にregexp_extractを用いる
    -- , regexp_extract(url, '//[^/]+([^?#]+)') AS url_path
    -- ■ Hive、SparkSQLの場合はparse_url関数でURLのパス部分を抽出する
    -- , parse_url(url, 'PATH') AS url_path
  FROM access_log
)
SELECT
    url_path
  , COUNT(DISTINCT short_session) AS access_count
  , COUNT(DISTINCT long_session) AS access_users
  , COUNT(*) AS page_view
FROM
  access_log_with_path
GROUP BY
  url_path
;
```

```
  url_path   | access_count | access_users | page_view
-------------+--------------+--------------+----------
 /           |         8925 |         7052 |     10592
 /detail     |        11562 |         9513 |     20132
 /list/cd    |         6156 |         4156 |      8435
 /list/dvd   |         5198 |         3157 |      7621
 /list/newly |         6289 |         3916 |      8165
…
```

URLに大きな意味を持たせて集計する

パス別に集計するクエリの出力結果では、URLの「/detail?id=**」は「/detail」にまとめられ、詳細ページ全体の集計が可能です。この状態では「/list/cd」や「/list/dvd」などとリストページが各カテゴリ毎に分かれています。

続いて、これを「カテゴリ・リストページ」としてまとめ、「/list/newly/」はカテゴリ・リストページではなく、「新着リストページ」としてまとめて、URLに大きな意味を持たせて集計します。

コード6.1.2.3：URLに大きな意味を持たせて集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_path AS (
  -- コード6.1.2.2参照
)
, access_log_with_split_path AS (
  -- パスの1階層目と2階層目を取得する
  SELECT *
    -- ■ PostgreSQL, Redshiftの場合、split_partでn番目の要素を抽出できる
    , split_part(url_path, '/', 2) AS path1
    , split_part(url_path, '/', 3) AS path2
    -- ■ BigQueryの場合、split関数で配列に分解するが、配列のインデックス指定が特殊になる
    -- , split(url_path, '/')[SAFE_ORDINAL(2)] AS path1
    -- , split(url_path, '/')[SAFE_ORDINAL(3)] AS path2
    -- ■ Hive、SparkSQLの場合も、split関数で配列に分解する
    -- ただし、配列のインデックスが0から始まる
    -- , split(url_path, '/')[1] AS path1
    -- , split(url_path, '/')[2] AS path2
  FROM access_log_with_path
)
, access_log_with_page_name AS (
  -- パスをスラッシュで分割し、条件に従ってページ名を付与する
  SELECT *
    , CASE
        WHEN path1 = 'list' THEN
          CASE
            WHEN path2 = 'newly' THEN 'newly_list'
```

```
            ELSE 'category_list'
          END
        -- 上記以外はパスを採用する
        ELSE url_path
      END AS page_name
  FROM access_log_with_split_path
)
SELECT
    page_name
  , COUNT(DISTINCT short_session) AS access_count
  , COUNT(DISTINCT long_session) AS access_users
  , COUNT(*) AS page_view
FROM access_log_with_page_name
GROUP BY page_name
ORDER BY page_name
;
```

```
   page_name   | access_count | access_users | page_view
---------------+--------------+--------------+-----------
 /             |         8925 |         7052 |     10592
 /detail       |        11562 |         9513 |     20132
 category_list |        10562 |         6015 |     16056
 newly_list    |         6289 |         3916 |      8165
```

以上で、全体ページビューをトップページ、カテゴリ・リストページ、詳細ページと大きな粒度での集計が可能になります。category_listとnewly_listにはCASE式で定義した名称、それ以外ではパスの抽出結果なので、/を含む形式で出力されます。求められるレポートの用途に応じて、集計の粒度を定め、集計する方法を身につけましょう。

ワンポイント

ログ送信時に、そのページが何であるか意味付けした「ページ名」を、URLとは別途送信する仕組みがあれば、集計側の作業は格段に楽になります。ログ送信を設計する段階から、以降の集計作業を見据えて、送信すべき要素を検討しましょう。

関連項目	3-1-1 コード値をラベルに置き換える 3-1-2 URLから要素を取り出す 3-1-3 文字列を配列に分解する

6-1-3 流入元別に訪問回数やCVRを集計する

SQL 正規表現、URL関数　　分析 流入元、URL生成ツール、CVR

Webサイトにアクセスする際、ブラウザに直接URLを入力する方法以外では、他のサイトのリンクなどを経由します。そこで主な流入経路を下記にまとめます。

- 検索エンジン
- 検索連動型広告
- Twitter、Facebook等のソーシャルメディア
- アフィリエイト
- アドネットワーク
- 他のWebサイトから貼られるリンク（ブログ、紹介記事）

上記の流入経路を個別に集計することで、Webサイトの訪問者がどのような行動の結果、Webサイトを訪問しているかが分かります。また、検索連動型広告やアフィリエイト、アドネットワークを管理するマーケティング部門の効果を可視化する目的でも利用可能です。

流入元の判定

流入元の判定方法をまとめます。リファラーには直前のページのURLが保存されます。そのリファラーとURLを使用し、下記の方法で流入元を判定します。

- URLパラメーターを元に判断する
- リファラーのドメインやランディングページで判定する

URLパラメーターを元に判断する

広告担当者がどの広告がどのぐらいクリックされたのか集計するため、URLに広告計測用パラメーターを設定します。Google Analyticsを導入している場合、URL生成ツールに従ってパラメーターを付与することが多いです。

Google Analyticsには、後述のURL生成ツールに従ってパラメーターを付与することで、その流入数がレポートされる仕組みが用意されています。

URL生成ツール

Google Analyticsで利用するURL生成ツール「Campaign URL Builder」（https://ga-dev-tools.appspot.com/campaign-url-builder/）は、Webサイトの「URL」や「Campaign Source」など必要なパラメーターを入力すれば、URLが生成される仕組みになっています。

URL生成ツール「Campaign URL Builder」で、生成されるURLを見てみましょう。出力されるURLは下表に示すルールに従います。

```
www.～～～.com?utm_source=google&utm_medium=cpc&utm_campaign=spring_sale
```

表 6.1.3.1：URL生成ツールのパラメーター

URL生成ツール項目	URLパラメーター	用途
Campaign Source(必須)	utm_source	The referrer サイト名や媒体の呼称
Campaign Medium	utm_medium	Marketing medium 流入の種類や分類
Campaign Name	utm_campaign	Product, promo code, or slogan キャンペーンの名称
Campaign Term	utm_term	Identify the paid keywords 主に検索連動型広告で使用する流入キーワード
Campaign Content	utm_content	Use to differentiate ads 広告を分類するための識別子

例えば、utm_source > utm_medium > utm_campaignと、階層構造を持って包含するイメージで命名していきます。下記に例を示します。

- yahoo > cpc > 201608
- yahoo > cpc > 201609
- yahoo > banner > 2016spring_sale

URLパラメーターを利用すれば、チラシや雑誌からの流入も、スマートフォン向けのコンテンツであれば計測できます。URLにどこから流入してきたかを判別するパラメーターを付与し、QRコードを作成します。そのQRコードをチラシに掲載することで、オフライン広告の効果を計測することが可能です。

リファラーのドメインやランディングページで判定する

URL生成ツールで生成したパラメーターはもちろん、それ以外のパラメーターが付与されるのは、広告担当者が予め設定した場合に限ります。検索エンジンや個人ブログ、Twitterなど、広告担当者が操作できない範囲からの流入は、リファラーに格納されるドメインで判定します。

また、別の手法として、チラシや雑誌などを使い会社名や商品名とは異なる、特徴のある語句で検索させて特設ページへ誘導し、ランディングページによって流入の契機を探ることも考えられます。この手法を用いることで、オフライン広告の効果を計測することが可能です。

流入元別に訪問回数を集計する

本項で紹介するSQLはリファラーが空ではなく、そのドメインが自サイトのドメインではないログを外部流入とし、下表に示すロジックで集計するものとします。

表6.1.3.2：流入元の判定ロジック

流入経路	判定方法	集計方法
検索連動型広告	URL生成ツールによって付与されたパラメーター	各流入経路のURLの全てには utm_source、utm_medium が付与されている前提の元、上記2つを結合した文字列を集計する
アフィリエイト		
アドネットワーク		
検索エンジン	ドメイン	リファラーのドメインが下記のものを検索エンジンとする ・search.yahoo.co.jp ・www.google.co.jp
ソーシャルメディア		リファラーのドメインが下記のものをソーシャルメディアとする ・twitter.com ・www.facebook.com
その他のサイト		上記に当てはまらない場合、otherとしてまとめる

上記の定義の通り、集計するクエリを下記コード例に示します。

コード6.1.3.1：流入元別に訪問回数を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_parse_info AS (
  -- 流入元情報を抽出する
  SELECT *
    -- ■ PostgreSQLの場合、正規表現で流入元情報を抽出する
    , substring(url from 'https?://([^/]*)') AS url_domain
    , substring(url from 'utm_source=([^&]*)') AS url_utm_source
    , substring(url from 'utm_medium=([^&]*)') AS url_utm_medium
    , substring(referrer from 'https?://([^/]*)') AS referrer_domain
    -- ■ Redshiftの場合、regexp_substr関数とregexp_replace関数を組み合わせる
    -- , regexp_replace(regexp_substr(url, 'https?://[^/]*'), 'https?://', '')
    --   AS url_domain
    -- , regexp_replace(regexp_substr(url, 'utm_source=[^&]*'), 'utm_source=', '')
    --   AS url_utm_source
    -- , regexp_replace(regexp_substr(url, 'utm_medium=[^&]*'), 'utm_medium=', '')
    --   AS url_utm_medium
    -- , regexp_replace(regexp_substr(referrer, 'https?://[^/]*'), 'https?://', '')
    --   AS referrer_domain
    -- ■ BigQueryの場合、正規表現にregexp_extractを用いる
    -- , regexp_extract(url , 'https?://([^/]*)') AS url_domain
    -- , regexp_extract(url , 'utm_source=([^&]*)') AS url_utm_source
    -- , regexp_extract(url , 'utm_medium=([^&]*)') AS url_utm_medium
    -- , regexp_extract(referrer, 'https?://([^/]*)') AS referrer_domain
    -- ■ Hive、SparkSQLの場合、parse_url関数でURLの各値を抽出する
    -- , parse_url(url, 'HOST') AS url_domain
```

```
    -- , parse_url(url, 'QUERY', 'utm_source') AS url_utm_source
    -- , parse_url(url, 'QUERY', 'utm_medium') AS url_utm_medium
    -- , parse_url(referrer, 'HOST') AS referrer_domain
  FROM access_log
)
, access_log_with_via_info AS (
  SELECT *
    , ROW_NUMBER() OVER(ORDER BY stamp) AS log_id
    , CASE
        WHEN url_utm_source <> '' AND url_utm_medium <> ''
          -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、3引数以上のconcat関数が使える
          THEN concat(url_utm_source, '-', url_utm_medium)
          -- ■ PostgreSQL, Redshiftの場合、文字列の連結に||演算子が使える
          -- THEN url_utm_source || '-' || url_utm_medium
        WHEN referrer_domain IN ('search.yahoo.co.jp', 'www.google.co.jp') THEN 'search'
        WHEN referrer_domain IN ('twitter.com', 'www.facebook.com') THEN 'social'
        ELSE 'other'
        -- ELSE referrer_domain とすることで、ドメイン別に集計可能
      END AS via
  FROM access_log_with_parse_info
  -- リファラーが空ではなく、そのドメインが自サイトのドメインではないログを流入元とする
  WHERE COALESCE(referrer_domain, '') NOT IN ('', url_domain)
)
SELECT via, COUNT(1) AS access_count
FROM access_log_with_via_info
GROUP BY via
ORDER BY access_count DESC;
```

```
       via        | access_count
------------------+--------------
 social           |        13205
 other            |        12564
 search           |        10515
 google-search    |         9502
 yahoo-search     |         7515
 mynavi-affiliate |         7246
```

流入元別にCVRを集計する

前述の出力結果で各訪問に対して購入した割合（CVR）を集計するクエリを、コード例として示します。本コード例では購入した割合を集計していますが、WITH句のaccess_log_with_purchase_amountテーブルを修正すれば、さまざまなアクションの割合を集計することも可能なので、目的に応じて適宜修正してください。

コード6.1.3.2：各訪問に対して購入した割合（CVR）を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_parse_info AS (
 -- コード6.1.3.1参照
)
, access_log_with_via_info AS (
 -- コード6.1.3.1参照
)
, access_log_with_purchase_amount AS (
 SELECT
     a.log_id
   , a.via
   , SUM(
       CASE
         -- ■ PostgreSQLの場合、interval型のデータを用いて日付/時刻の四則演算ができる
         WHEN p.stamp::date BETWEEN a.stamp::date AND a.stamp::date + '1 day'::interval
         -- ■ Redshiftの場合、dateadd関数を用いる
         -- WHEN p.stamp::date BETWEEN a.stamp::date AND dateadd(day, 1, a.stamp::date)
         -- ■ BigQueryの場合、date_add関数を用いる
         -- WHEN date(timestamp(p.stamp))
         --   BETWEEN date(timestamp(a.stamp))
         --     AND date_add(date(timestamp(a.stamp)), interval 1 day)
         -- ■ Hive, SparkSQLの場合、date_add関数を用いる
         --   ※ BigQueryとは書式が異なる
         -- WHEN to_date(p.stamp)
         --   BETWEEN to_date(a.stamp) AND date_add(to_date(a.stamp), 1)
         THEN amount
       END
     ) AS amount
 FROM
     access_log_with_via_info AS a
   LEFT OUTER JOIN
     purchase_log AS p
     ON a.long_session = p.long_session
 GROUP BY a.log_id, a.via
)
SELECT
   via
 , COUNT(1) AS via_count
 , COUNT(amount) AS conversions
 , AVG(100.0 * SIGN(COALESCE(amount, 0))) AS cvr
 , SUM(COALESCE(amount, 0)) AS amount
 , AVG(1.0 * COALESCE(amount, 0)) AS avg_amount
FROM
 access_log_with_purchase_amount
GROUP BY via
ORDER BY cvr DESC
;
```

```
       via        | via_count | conversions |  cvr  | amount |  avg_amount
------------------+-----------+-------------+-------+--------+-------------
 social           |     13205 |         214 |  1.62 | 214000 |    16.2059
 mynavi-affiliate |      7246 |         105 |  1.44 | 105000 |    14.4907
 yahoo-search     |      7515 |          75 |  0.99 |  75000 |     9.9800
 search           |     10515 |          95 |  0.90 |  95000 |     9.0347
 google-search    |      9502 |          84 |  0.88 |  84000 |     8.8402
 other            |     12564 |          20 |  0.15 |  20000 |     1.5918
```

まとめ

上記の結果を元に、どの流入経路に対して注力すべきか判断することが可能です。また、極端に低い流入経路に関してはその原因を探る、有料集客の施策を行っている場合は、効率が悪い掲載を削除するなどの判断を下しましょう。

6-1-4 アクセスされる曜日、時間帯を把握する

SQL 日付関数、文字列関数　　分析 曜日、時間帯

ユーザーがアクセスする曜日・時間帯はサービスによって特性があります。土日に頻繁に利用されるサービス、休日前に利用されるサービス、曜日時間帯を問わず、隙間時間で利用されるサービスなど、さまざまなパターンがあります。これらの特性を把握することで、お知らせやメールマガジンの配信タイミング、キャンペーンの開始日時と終了日時の検討など、ユーザー像を把握するための材料として利用できます。

外部メディアに対する露出が瞬間的に高まった時、お知らせやメールマガジン配信時などにはその効果がしっかりと現れます。定常的にモニタリングするレポートではありませんが、必要に応じて集計してみましょう。

図 6.1.4.1：曜日別、アクセス時間帯別ページビュー

曜日	17:00	17:30	18:00	18:30	19:00	19:30	20:00	20:30	21:00	21:30	22:00	22:30	23:00	23:30
日	4,897	5,058	5,093	5,122	7,323	5,986	6,089	6,208	6,243	6,377	6,798	6,656	6,584	6,692
月	5,206	5,505	5,564	5,718	5,787	5,897	6,346	6,694	6,746	6,926	7,526	7,583	7,650	7,408
火	4,985	5,222	5,405	5,524	5,601	5,699	6,325	6,665	6,764	6,981	7,117	7,360	7,209	7,026
水	4,614	4,839	4,994	5,182	5,246	5,463	5,921	6,370	6,728	6,635	6,894	6,962	6,999	6,944
木	5,548	5,596	5,696	5,963	7,929	7,030	7,108	7,296	7,398	7,685	7,635	7,871	7,786	7,595
金	4,821	4,923	5,207	5,358	5,647	6,417	6,232	6,622	6,550	6,837	7,162	11,124	8,390	7,670
土	4,911	5,094	5,103	5,324	5,145	5,222	5,567	5,905	6,156	6,484	6,735	6,861	6,947	6,930

前図に示すレポートの作成手順は、以下の通りです。

1. 24時間の抽出したい単位を定める（10分刻み、15分刻み、30分刻み）
2. アクセスされた時間をその単位に丸め、曜日と共に訪問者数を集計する

下記コード例に、上記手順をSQLで表現し、30分刻みで時間帯を集計するクエリを示します。曜日の集計は、日付関数を用いて曜日番号を求め、CASE式を用いて列に展開する方法を採用しています。ただし、曜日番号の定義がミドルウェアによって異なるため、注意が必要です。

下表に、ミドルウェア別の曜日番号を取得するためのSQL構文と出力フォーマット一覧を示します。本項のコード例では、日曜日（0）から土曜日（6）のフォーマットに合わせて、すべての場合のクエリを調整しています。また、時間帯を任意の時間幅で刻むために、時刻を一旦00:00:00からの経過秒数に変換し、指定した時間幅で割って切り捨てを行い、再び時刻表記に戻しています。

表6.1.4.1: 曜日番号を取得するためのSQL構文と出力フォーマット一覧

対応ミドルウェア	SQL構文	曜日番号の出力フォーマット
PostgreSQL, Redshift	date_part('dow' , timestamp)	日曜日(0)～土曜日(6)
PostgreSQL	date_part('isodow' , timestamp)	月曜日(1)～日曜日(7)
BigQuery	EXTRACT(dayofweek from timestamp)	日曜日(1)～土曜日(7)
Hive, SparkSQL	from_unixtime(unitimestamp, 'u')	月曜日(1)～日曜日(7)

コード6.1.4.1：曜日・時間帯別の訪問者数を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_dow AS (
  SELECT
      stamp
    -- 日曜日(0)から土曜日(6)までの曜日番号を取得する
    -- ■ PostgreSQL, Redshiftの場合、date_part関数を用いる
    , date_part('dow', stamp::timestamp) AS dow
    -- ■ BigQueryの場合、EXTRACT(dayofweek from ~)関数を用いる
    -- , EXTRACT(dayofweek from timestamp(stamp)) - 1 AS dow
    -- ■ Hive, SparkSQLの場合、from_unixtime関数を用いる
    -- , from_unixtime(unix_timestamp(stamp), 'u') % 7 AS dow

    -- 00:00:00からの経過秒数を計算する
    -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、
    --   substring関数で時、分、秒の位置を抽出し、総秒数に直す
    -- ■ BigQueryの場合、substringをsubstr, intをint64に修正する
    ,   CAST(substring(stamp, 12, 2) AS int) * 60 * 60
      + CAST(substring(stamp, 15, 2) AS int) * 60
      + CAST(substring(stamp, 18, 2) AS int)
      AS whole_seconds

    -- タイムスタンプを切り捨てる秒数を定義する
    -- ここでは30分(1800秒)に指定
    , 30 * 60 AS interval_seconds
```

```
  FROM access_log
)
, access_log_with_floor_seconds AS (
  SELECT
      stamp
    , dow
    -- 00:00:00からの経過秒数をinterval_secondsで切り捨てる
    -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合は下記のとおり
    -- ■ BigQueryの場合、intをint64に修正する
     , CAST((floor(whole_seconds / interval_seconds) * interval_seconds) AS int)
       AS floor_seconds
  FROM access_log_with_dow
)
, access_log_with_index AS (
  SELECT
      stamp
    , dow
    -- 経秒数からタイムスタンプの時刻表記に変換する
    -- ■ PostgreSQL, Redshiftの場合は下記のとおり
    ,    lpad(floor(floor_seconds / (60 * 60))::text     , 2, '0') || ':'
      || lpad(floor(floor_seconds % (60 * 60) / 60)::text, 2, '0') || ':'
      || lpad(floor(floor_seconds % 60)::text            , 2, '0')
    -- ■ BigQueryの場合は下記のとおり
    -- , concat(
    --     lpad(CAST(floor(floor_seconds / (60 * 60))        AS string), 2, '0'), ':'
    --   , lpad(CAST(floor(mod(floor_seconds, 60 * 60)) / 60 AS string), 2, '0'), ':'
    --   , lpad(CAST(floor(mod(floor_seconds, 60))           AS string), 2, '0')
    -- ■ Hive, SparkSQLの場合は下記のとおり
    -- , concat(
    --     lpad(CAST(floor(floor_seconds / (60 * 60))      AS string), 2, '0'), ':'
    --   , lpad(CAST(floor(floor_seconds % (60 * 60) / 60) AS string), 2, '0'), ':'
    --   , lpad(CAST(floor(floor_seconds % 60)             AS string), 2, '0')
      AS index_time
  FROM access_log_with_floor_seconds
)
SELECT
    index_time
  , COUNT(CASE dow WHEN 0 THEN 1 END) AS sun
  , COUNT(CASE dow WHEN 1 THEN 1 END) AS mon
  , COUNT(CASE dow WHEN 2 THEN 1 END) AS tue
  , COUNT(CASE dow WHEN 3 THEN 1 END) AS wed
  , COUNT(CASE dow WHEN 4 THEN 1 END) AS thu
  , COUNT(CASE dow WHEN 5 THEN 1 END) AS fri
  , COUNT(CASE dow WHEN 6 THEN 1 END) AS sat
FROM
  access_log_with_index
GROUP BY
  index_time
ORDER BY
  index_time
;
```

```
 index_time | sun | mon | tue | wed | thu | fri | sat
------------+-----+-----+-----+-----+-----+-----+-----
19:00:00    | 7323| 5787| 5601| 5246| 7929| 5647| 5145
19:30:00    | 5986| 5897| 5699| 5463| 7030| 6417| 5222
20:00:00    | 6089| 6346| 6325| 5921| 7108| 6232| 5567
20:30:00    | 6208| 6694| 6665| 6370| 7296| 6622| 5905
21:00:00    | 6243| 6746| 6764| 6728| 7398| 6550| 6156
21:30:00    | 6377| 6926| 6981| 6635| 7685| 6837| 6484
```

ワンポイント

ユーザーの訪問が多い時間帯に実施するキャンペーンは安定した効果が見込めますが、逆に利用者の少ない時間帯に対して、ECサイトならタイムセール、ソーシャルゲームであればアイテム獲得率アップのイベントなど、底上げする施策を検討するのも一案です。

関連項目	3-1-4 日付やタイムスタンプを扱う 3-3-3 縦持ちのデータを横持ちに変換する

デバイスによるユーザー行動の違いを意識しよう

PCサイトとSPサイトではユーザーが利用する場所が異なります。利用場所が異なれば、目的も異なる可能性があります。

PCサイトは主に家庭や職場で利用するユーザーが多いはずです。ユーザー行動として、じっくりと比較検討したり、携帯では面倒な申し込みなどをパソコンで行うケースなどが考えられます。一方、SPサイトは、家庭や職場だけではなく、移動中や外出先で利用しているケースが想定できます。外出先での利用では、地図が分かりやすい、すぐに電話できる、クーポンの表示が可能など、ユーザーの抱える課題を解決することが何よりも重要になります。

著名なWebサイトでPCサイトとSPサイトの構成を比較すると、SPサイトには上記の課題解決に直接繋がるアクションモジュールが、PCサイトよりも上部に配置されていることが多々あります。

PCサイトとSPサイトでは、ユーザーの行動パターンや求める情報が異なることを意識して、サービスを運営することと同様に、分析時でも利用状況の集計では、PCサイトとSPサイトを分類して集計し、相違点を発見するなどして、改善を提案しましょう。

6-2
サイト内のユーザー行動を把握する

本節ではWebサイトでの特徴的な指標（訪問者数や訪問回数、直帰率、離脱率など）、レポート（フォールアウトレポートやユーザーフローなど）を作成するSQLを紹介します。

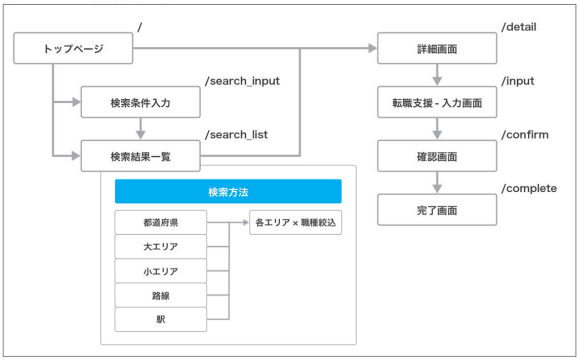
アクセス解析ツールを導入しているサービスでは、各種指標の数値は既に把握済みかもしれません。しかし、ビッグデータ基盤でデータが抽出可能になると、アクセス解析ツールとは違って、ユーザーデータや商品データなど各種業務データと紐付けることが可能となり、より詳細に分析できます。

また、直帰率と離脱率は、名前が類似していますが、集計内容は異なるものです。これらの指標に関しても定義から解説するので、しっかりと理解しましょう。

サンプルデータ

本節では一部の項を除いて、転職支援サービスを想定して説明します。もちろん、転職支援サービス以外でも、適用が可能な内容です。下図に、本項で扱うデータの主な遷移を示します。

図6.2.0.1：6-2で扱うサイトマップ



本項で取り扱う転職支援サービスは、検索結果の一覧表示で、検索条件としてエリアを指定する複数の手段を提供し、エリアの階層構造は都道府県、大エリア、小エリアで構成されるものとします。また、検索結果は任意の職種で絞り込み可能とします。検索結果（/search_list）の閲覧ログには、使用した検索方法を示す情報として検索タイプをログとして出力します（下表参照）。

表6.2.0.1：検索結果一覧における検索可能な条件と検索タイプに出力する文字列

都道府県	大エリア	小エリア	職種	検索タイプ
東京都	-	-	-	Pref
東京都	23区東部	-	-	Area-L
東京都	23区東部	上野・浅草・両国	-	Area-S
東京都	23区東部	葛飾・江戸川・江東	-	Area-S
東京都	-	-	人事	Pref-with-Job
東京都	23区西部	-	営業	Area-L-with-Job
東京都	23区西部	中野・杉並	技術	Area-S-with-Job

また別の検索方法として、路線と駅による検索を提供し、職種の絞り込みが可能にしています。その際にログに出力する検索タイプを下表に示します。

表6.2.0.2：検索結果一覧における検索可能な条件と検索タイプに出力する文字列

都道府県	路線	駅	職種	検索タイプ
東京都	山手線	-	-	Line
東京都	山手線	新宿駅	-	Station
東京都	山手線	代々木駅	-	Station
東京都	中央線	-	営業	Line-with-Job
東京都	中央線	四谷駅	人事	Station-with-Job

本節では、上述のサービスでユーザーがサイトを閲覧したログ（下記参照）を使用して、サイト内のユーザーの行動を把握する手法とSQLを紹介します。

データ6.2.0.1：検索タイプを含むアクションログ(activity_log)テーブル

```
      stamp      | session  | action | option |     path       | search_type
-----------------+----------+--------+--------+----------------+-----------------
 2016.. 21:47:15 | 0d486d94 | view   | page   | /              |
 2016.. 21:48:11 | 0d486d94 | view   | search | /search_list   | Area-S-with-Job
 2016.. 21:48:35 | 0d486d94 | view   | detail | /detail        |
 2016.. 21:49:26 | 0d486d94 | view   | page   | /              |
 2016.. 21:50:05 | 0d486d94 | view   | search | /search_list   | Area-S
 2016.. 21:47:15 | d9da20ba | view   | page   | /search_input  |
 2016.. 21:47:55 | d9da20ba | view   | page   | /              |
 2016.. 21:47:15 | 8af209cf | view   | search | /search_list   | Area-L-with-Job
 2016.. 21:47:33 | 8af209cf | view   | detail | /detail        |
 2016.. 21:48:13 | 8af209cf | view   | search | /search_list   | Area-S
..
```

6-2-1 入口ページと出口ページを把握する

SQL FIRST_VALUE関数、LAST_VALUE関数

分析 入口ページ、出口ページ

入口ページはサイトに訪問した際に、最初にアクセスしたページを指します。一部のアクセス解析ツールでは、ランディングページとも呼ばれます。出口ページは最後にアクセスし、離脱したページを指します。アクセス解析ツールによっては離脱ページとも呼ばれます。

入口ページと出口ページを集計する

まずは、下表に示す結果を得るためのクエリを考えてみましょう。

表6.2.1.1：入口ページと出口ページを集計する

type	パス	入口数・出口数
landing	/detail	3,539
landing	/search_list	2,330
exit	/detail	3,663
exit	/search_list	2,571

アクセスログの各セッションで、入口ページと出口ページのURLを取得するには、下記コード例に示す通り、FIRST_VALUE／LAST_VALUEウィンドウ関数が利用できます。OVER句の指定には、ORDER BY stamp ASCでタイムスタンプの昇順を指定しますが、ORDER BYを指定した場合のウィンドウ関数のパーティションは、デフォルトでは最初の行から現在の行までになるため、ROWS BETWEEN ～ を用いて、同一セッション内のすべての行を対象にします。

コード6.2.1.1：各セッションの入口ページと出口ページのパスを取得するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_landing_exit AS (
  SELECT
      session
    , path
    , stamp
    , FIRST_VALUE(path)
        OVER(
          PARTITION BY session
          ORDER BY stamp ASC
            ROWS BETWEEN UNBOUNDED PRECEDING
                     AND UNBOUNDED FOLLOWING
        ) AS landing
```

```
    , LAST_VALUE(path)
        OVER(
          PARTITION BY session
          ORDER BY stamp ASC
            ROWS BETWEEN UNBOUNDED PRECEDING
                     AND UNBOUNDED FOLLOWING
        ) AS exit
  FROM activity_log
)
SELECT *
FROM
  activity_log_with_landing_exit
;
```

```
 session  |     path       |        stamp        |    landing    |      exit
----------+----------------+---------------------+---------------+---------------
 0006fbb5 | /search_list   | 2016-12-29 21:47:15 | /search_list  | /detail
 0006fbb5 | /              | 2016-12-29 21:48:21 | /search_list  | /detail
 0006fbb5 | /search_input  | 2016-12-29 21:49:58 | /search_list  | /detail
 0006fbb5 | /search_list   | 2016-12-29 21:50:52 | /search_list  | /detail
 0006fbb5 | /detail        | 2016-12-29 21:51:49 | /search_list  | /detail
 0007a5f9 | /              | 2016-12-29 21:47:17 | /             | /search_list
 0007a5f9 | /detail        | 2016-12-29 21:47:17 | /             | /search_list
 0007a5f9 | /search_list   | 2016-12-29 21:48:19 | /             | /search_list
...
```

上記の出力結果を見ると、各セッションに対して、入口ページと出口ページのURLが付与されていることが分かります。

各セッションの入口ページと出口ページのURLが抽出できたら、下記コード例に示す通り、入口ページと出口ページのURLでユニークなセッション数を集計すれば、訪問回数が求められます。

コード6.2.1.2：各セッションの入口ページと出口ページの訪問回数を取得するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_landing_exit AS (
  -- コード6.2.1.1参照
)
, landing_count AS (
  -- 入口ページの訪問回数を集計する
  SELECT
      landing AS path
    , COUNT(DISTINCT session) AS count
  FROM
    activity_log_with_landing_exit
  GROUP BY landing
```

```
)
, exit_count AS (
  -- 出口ページの訪問回数を集計する
  SELECT
      exit AS path
    , COUNT(DISTINCT session) AS count
  FROM
    activity_log_with_landing_exit
  GROUP BY exit
)
  -- 入口ページと出口ページの訪問回数結果を同時に表示する
  SELECT 'landing' AS type, * FROM landing_count
UNION ALL
  SELECT 'exit'    AS type, * FROM exit_count
;
```

```
  type   |      path      | count
---------+----------------+-------
 landing | /              |  1561
 landing | /detail        |  3539
 landing | /search_input  |   763
 landing | /search_list   |  2330
 exit    | /              |   993
 exit    | /complete      |    44
 exit    | /confirm       |     5
 exit    | /detail        |  3663
 exit    | /input         |   211
 exit    | /search_input  |   706
 exit    | /search_list   |  2571
```

どこから閲覧を開始して、どこで離脱したのかを集計する

前述のコード例では、入口ページと出口ページの訪問回数は分かりますが、どのページから閲覧を開始したユーザーがどのページで離脱しているか、その組み合わせを把握できません。閲覧開始ページと離脱ページの組み合わせを集計するクエリをコード例に示します。

表 6.2.1.2：セッション別の入口ページと出口ページの組み合わせと訪問回数

入口ページ	出口ページ	訪問回数
/search_list	**/search_list**	1,128
/search_list	**/detail**	750
/search_list	**/**	232
/search_list	**/search_input**	146

コード6.2.1.3：セッション別の入口ページと出口ページの組み合わせを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_landing_exit AS (
  -- コード6.2.1.1参照
)
SELECT
    landing
  , exit
  , COUNT(DISTINCT session) AS count
FROM
  activity_log_with_landing_exit
GROUP BY
  landing, exit
;
```

```
    landing     |      exit      | count
----------------+----------------+-------
 /search_list   | /              |   232
 /search_list   | /complete      |    14
 /search_list   | /confirm       |     2
 /search_list   | /detail        |   750
 /search_list   | /input         |    58
 /search_list   | /search_input  |   146
 /search_list   | /search_list   |  1128
...
```

ワンポイント

Webサイト担当者はトップページからサイト設計を考えがちですが、トップページから閲覧を開始するユーザーは意外と少なく、詳細ページから閲覧を開始するユーザーが多いサイトも存在します。入口ページを把握することで、サイト設計の前提から考え直すきっかけになります。

関連項目	3-3-2 グループの中での順序を扱う 6-1-2 ページ毎の訪問者数・訪問回数・ページビューを集計する

6-2-2 離脱率と直帰率を計算する

SQL ROW_NUMBER関数、COUNTウィンドウ関数、AVG(CASE～END)

分析 離脱率、直帰率

前項では、出口ページの集計を紹介していますが、紹介したSQLでは出口ページとなった数のみを集計しているため、そのページを閲覧した母数に対する割合（離脱率）までは分かりません。本項では、前項の出口ページを用いて離脱率を求め、問題となるページを洗い出す方法を紹介します。

また、直帰率は該当ページのみを閲覧して離脱した訪問の割合を計算する指標です。直帰率が高いページは資料請求や会員登録、購入等の成果に繋がっていないページと判断できるため、何らかの対策が必要です。広告収益をメインとするサイトでは、単ページの閲覧だけではなく、その他のページも閲覧してもらうことで広告のクリックや収益が高まります。トップページの直帰率が高いのであれば、ユーザーがサイト内で何かを探す動機付けが弱いと判断できます。ナビゲーションの配置や各種モジュールの配置などを検討しましょう。

離脱率と直帰率はまったく異なる指標です。本項ではそれぞれその定義と計算方法を紹介します。

離脱率を集計する

離脱率を求めるには、下記の式を用いて計算します。

離脱率＝出口数 / ページビュー

単純に離脱率の高いページが悪い訳ではありません。ユーザーが満足して去っていくページ（購入完了画面や記事詳細画面）はもちろん離脱率は高くなり、問題視するものではありません。ユーザーが満足せず、その途中ページで離脱している状態は、改善の対象となります。

下表に示す離脱率のレポート（パスや出口数、ページビュー、離脱率）を作成するためのクエリと出力結果を次に示します。

表 6.2.2.1：離脱率

パス	出口数	ページビュー	離脱率
/detail	3,667	9,365	39.1%
/search_list	2,574	8,516	30.2%

コード6.2.2.1：パスごとの離脱率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_exit_flag AS (
 SELECT
     *
   -- 出口ページ判定
   , CASE
       WHEN ROW_NUMBER() OVER(PARTITION BY session ORDER BY stamp DESC) = 1  THEN 1
       ELSE 0
     END AS is_exit
 FROM
   activity_log
)
SELECT
   path
 , SUM(is_exit) AS exit_count
 , COUNT(1) AS page_view
 , AVG(100.0 * is_exit) AS exit_ratio
FROM
 activity_log_with_exit_flag
GROUP BY path
;
```

```
     path       | exit_count | page_view | exit_ratio
----------------+------------+-----------+------------
 /confirm       |          5 |        67 |       7.46
 /              |        994 |      4604 |      21.58
 /search_input  |        698 |      2640 |      26.43
 /detail        |       3667 |      9365 |      39.15
 /complete      |         44 |        52 |      84.61
 /input         |        211 |       327 |      64.52
 /search_list   |       2574 |      8516 |      30.22
```

直帰率を集計する

直帰率を求めるには、直帰数（1ページだけが閲覧された訪問回数）を求めた上で、下記の式を用いて計算します。

直帰率＝直帰数（1ページだけ閲覧された訪問回数）／入口数

または、別の考え方として次の式を用いることがあります。

直帰率＝直帰数（1ページだけ閲覧された訪問回数）／訪問回数

純粋にランディングページから他のページへ遷移しているかを評価する際は、前者の指標が直帰率として、より正しい計算式だと考えます。本書では前者を直帰率の式として採用しています。

該当ページのみを閲覧して離脱してしまう原因として、他の記事や商品へとユーザーを回遊させるモジュールが機能していないと考えられます。あるいは、そのページ自体のコンテンツにユーザーが満足するに足るものではない、遷移が複雑で次にどう操作すれば良いのか、ユーザーが直感的に理解できない構造など、さまざまな理由が考えられます。

下表に示す直帰率のレポート（パス、直帰数、入口数、直帰率）を作成するためのクエリと、その出力結果を次に示します。

表6.2.2.2：直帰率

パス	直帰数	入口数	直帰率
/detail	1,652	3,539	46.6%
/search_list	585	2,330	25.1%

コード6.2.2.2：パスごとの直帰率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_landing_bounce_flag AS (
 SELECT
     *
   -- 入口ページ判定
   , CASE
       WHEN ROW_NUMBER() OVER(PARTITION BY session ORDER BY stamp ASC) = 1  THEN 1
       ELSE 0
     END AS is_landing
   -- 直帰判定
   , CASE
       WHEN COUNT(1) OVER(PARTITION BY session) = 1  THEN 1
       ELSE 0
     END AS is_bounce
 FROM
   activity_log
)
SELECT
   path
 , SUM(is_bounce) AS bounce_count
 , SUM(is_landing) AS landing_count
 , AVG(100.0 * CASE WHEN is_landing = 1 THEN is_bounce END) AS bounce_ratio
FROM
 activity_log_with_landing_bounce_flag
GROUP BY path
;
```

```
     path       | bounce_count | landing_count | bounce_ratio
----------------+--------------+---------------+--------------
 /confirm       |            0 |             0 |
 /              |          236 |          1561 |      15.11
 /search_input  |          228 |           763 |      29.88
 /detail        |         1652 |          3539 |      46.67
 /complete      |            0 |             0 |
 /input         |            0 |             0 |
 /search_list   |          585 |          2330 |      25.10
```

ワンポイント

パソコン向けサイトとスマートフォン向けサイトが個別に存在する場合は、各種コンテンツの配置が異なるケース、ユーザーが求めるコンテンツに相違があるケースがあるので、パソコン向けサイトとスマートフォン向けサイトは、それぞれ分けて集計しましょう。

関連項目 | 3-3-2 グループの中での順序を扱う

6-2-3 成果に結び付くページを把握する

SQL SIGN関数、SUMウィンドウ関数　　分析 CVR

訪問回数をいくら上げても成果に結び付かないページに誘導するよりも、より成果に直結するページへ誘導した方が、Webサイト全体のCVRが向上します。例えば、複数の検索方法を提供しているケースでは、成果に至る割合が少ない検索機能は表示位置の優先順位を下げたり、機能の削除を検討することができます。

また、ページ毎の比較において、アクセス数では注目すべき数字ではないが、特定のページで閲覧ユーザーが成果に至る割合が高ければ、該当ページの訴求がユーザーに良い影響を与えていると判断でき、ページの構成を再検討する際の材料となります。

本項では、どのページに訪問したら成果が上がりやすいのか大雑把に把握するために、各ページまたはパスの訪問回数とその訪問したセッションで成果に結び付いているかを集計します。

機能での比較、ページでの比較、キャンペーンでの比較、コンテンツ種類での比較など、さまざまな比較パターンを、下図に示します。

図 6.2.3.1：様々な比較パターン

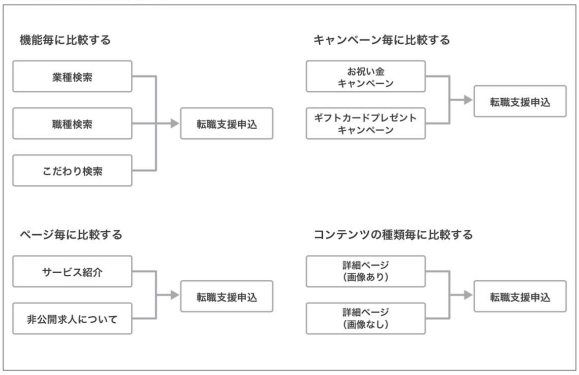


表 6.2.3.1：パス毎の訪問回数と成果

パス	訪問回数	成果数	CVR
/detail	5,894	52	0.88%
/search_list	5,045	31	0.61%

本項での分析では、完了画面（/complete）に到達したことを成果（コンバージョン）として定義し、完了画面に至るまでにユーザーが訪問したパスに成果を計上します。

下記のコード例に、コンバージョンしたページより前のアクセスにフラグを立てるクエリを示します。ウィンドウ関数を用いて、セッションごとにタイムスタンプの降順に並べ替え、/completeのページにアクセスするまでのアクセスログにフラグを立てます。

コード6.2.3.1: コンバージョンしたページより前のアクセスにフラグを立てるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
, activity_log_with_conversion_flag AS (
  SELECT
      session
    , stamp
    , path
      -- コンバージョンしたページより前のアクセスにフラグを立てる
    , SIGN(SUM(CASE WHEN path = '/complete' THEN 1 ELSE 0 END)
          OVER(PARTITION BY session ORDER BY stamp DESC
              ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW))
      AS has_conversion
  FROM
    activity_log
)
SELECT *
FROM
  activity_log_with_conversion_flag
ORDER BY
  session, stamp
;
```

```
 session  |        stamp        |     path       | has_conversion
----------+---------------------+----------------+----------------
 0006fbb5 | 2016-12-29 21:47:15 | /search_list   |              0
 0006fbb5 | 2016-12-29 21:48:21 | /              |              0
 0006fbb5 | 2016-12-29 21:49:58 | /search_input  |              0
 0006fbb5 | 2016-12-29 21:50:52 | /search_list   |              0
 0006fbb5 | 2016-12-29 21:51:49 | /detail        |              0
...
 05e9cc2b | 2016-12-29 21:49:50 | /search_list   |              1
 05e9cc2b | 2016-12-29 21:50:21 | /detail        |              1
 05e9cc2b | 2016-12-29 21:50:37 | /input         |              1
 05e9cc2b | 2016-12-29 21:51:03 | /confirm       |              1
 05e9cc2b | 2016-12-29 21:52:39 | /complete      |              1
 05e9cc2b | 2016-12-29 21:54:05 | /              |              0
 05e9cc2b | 2016-12-29 21:54:46 | /detail        |              0
...
```

上記の出力結果を確認すると、/completeへのアクセスを含むセッション（05e9cc2b）のログでは、/completeに至るまでのアクセスログのhas_session_conversionカラムにフラグ１が立っています。

/complete以降のアクセスではページ価値の計算に含めないため、フラグは０となります。

また、/completeへのアクセスを含まないセッションは、すべてのログのフラグが０となっています。

次のステップ 続いて、下記コード例に、パスごとの訪問回数と成果数を集計して、CVRを計算するクエリを示します。ここでの比率計算では、分母がセッションのユニーク数（DISTINCT）となるので、単純なコンバージョンフラグのAVGではなく、SUM / COUNTとして計算しています。

コード6.2.3.2：パスごとの訪問回数と成果数を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_conversion_flag AS (
  -- コード6.2.3.1参照
)
SELECT
    path
  -- 訪問回数
  , COUNT(DISTINCT session) AS sessions
  -- 成果数
  , SUM(has_conversion) AS conversions
  -- 成果数 / 訪問回数
  , 1.0 * SUM(has_conversion) / COUNT(DISTINCT session) AS cvr
FROM
  activity_log_with_conversion_flag
-- パスごとに集約
GROUP BY path
;
```

```
     path       | sessions | conversions |  cvr
----------------+----------+-------------+-------
 /confirm       |       65 |          52 | 0.800
 /              |     3445 |          18 | 0.005
 /search_input  |     2182 |           7 | 0.003
 /detail        |     5894 |          52 | 0.008
 /complete      |       52 |          52 | 1.000
 /input         |      314 |          52 | 0.165
 /search_list   |     5045 |          31 | 0.006
```

ワンポイント

商品購入、資料請求、会員登録等を成果とする場合、成果直前にあるフォームはCVRがかなり高くなります。同じ階層のコンテンツ同士、類似するコンテンツ同士で比較しましょう。

関連項目 | 3-3-2 グループの中での順序を扱う

6-2-4 ページの価値を算出する

SQL ROW_NUMBER関数　　分析 ページ価値

成果に結び付かないページへユーザーを回遊させるよりも、より成果に結び付くページへの回遊を増やすことは、Webサイトにおける売上向上の手段の1つです。特定ページを経由したユーザーがより成果に至る割合が高いのであれば、該当ページを経由させる構成にサイトマップを変更したり、該当ページのコンテンツやクリエイティブをその他のページにも適用していきます。

前項「6-2-3 成果に結び付くページを把握する」(P.269) で紹介した方法でも、成果に結び付きやすいページを確認できますが、さらに金額の概念を持たせて、成果をどのように各ページに割り振るかを考慮した「ページ価値」と呼ばれる指標を用いることで、より詳細な分析が可能です。

ページの価値を集計する準備

成果を数値化し、回遊したページにその数値を割り当てて、ページの価値を算出しますが、そのためには、「何を成果の数値とするか」と「何を評価したいのか」を決める必要があります。

成果の数値化

成果の数値化では、該当サイトでの成果が「購入」であれば、購入金額を成果の数値とします。

また、成果を資料請求や見積もり依頼の申込み、もしくは各請求や依頼フォームへの遷移ボタンとした場合、成果の数値を1と設定、または資料請求1件あたりの売上を設定します。

売上金額などの具体的な情報が算出できないケースでは、仮の数値として1CV当たりの価値を1,000と設定すれば、ページ価値を相対的に比較することが可能です。

ページに価値を割り当てる方法

ページの価値を用いて、どのような判断を下したいのか、それによってページに価値を割り当てるロジックが変わってきます。ページの価値を計算するロジックは、ツールによって内容や提供しているものが異なりますが、本項では下記の5つを紹介します。

- 最後のページに割り当てる
- 最初のページに割り当てる
- 均等に配分する
- 成果地点から近いページにより高く割り当てる
- 成果地点から遠いページにより高く割り当てる

最後のページに割り当てる

直接的な効果があったページに対して、成果を割り当てます。これにより、直接売上に貢献するページを見つけ出すことができます。

最初のページに割り当てる

成果に至るきっかけとなる最初のページに成果を割り当てます。これにより、間接的な効果として、価値の高いページを見つけ出せます。例えば、広告や検索エンジン等の外部流入に対して、価値の高いページを見つける際に利用します。

均等に配分する

成果に至るまでに経由した全ページに成果を均等に割り当てます。経由することで購入や申込みの意欲が高まるページを見つけ出せます。より少ないページ閲覧で成果に結び付いた場合は、必然的に各ページの価値が高くなります。また、何度も閲覧されるページの価値がより高くなります。

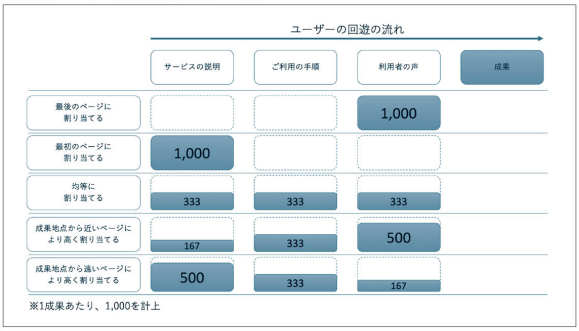
成果地点から近いページにより高く割り当てる

最後のページに割り当てるロジックに対して、段階を与えます。

成果地点から遠いページにより高く割り当てる

最初のページに割り当てるロジックに対して、段階を与えます。

図 6.2.4.1：ページに価値を割り当てる５つの方法



任意の階級数でヒストグラムを作成する

ページの価値を求めるための「何を成果の数値とするか」と「何を評価したいのか」が決まったら、実際に集計するためのSQLを作成します。

本項では例として、申込みフォームの完了画面を表示した訪問を成果として、その成果の数値を1,000として計算し、経由ページに均等に割り当てて集計します。また、ページの価値を入力〜確認〜完了ページに対して割り当てると正しく集計できないため、計算の対象からは除外しています。

下記のコード例に、ページ価値の割り振りを計算するクエリを示します。ページ価値の計算対象となる条件は、前項のコード例（コード6.2.3.1）に示したhas_conversionのフラグが1かつ、「入力」「確認」「完了」ページ以外のアクセスログです。これらの条件をWHERE句に指定します。

SELECT句中では、1コンバージョン当たり1,000の価値を各ロジックに割り振った場合の、1ログ当たりの価値をウィンドウ関数を用いて計算します。

コード6.2.4.1: ページ価値の割り振りを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_session_conversion_flag AS (
  -- コード6.2.3.1参照
)
, activity_log_with_conversion_assing AS (
  SELECT
      session
    , stamp
    , path
    -- コンバージョンに至るアクセスログに昇順で番号を振る
    , ROW_NUMBER() OVER(PARTITION BY session ORDER BY stamp ASC) AS asc_order
    -- コンバージョンに至るアクセスログに降順で番号を振る
    , ROW_NUMBER() OVER(PARTITION BY session ORDER BY stamp DESC) AS desc_order
    -- コンバージョンに至るアクセスログの数をカウントする
    , COUNT(1) OVER(PARTITION BY session) AS page_count

    -- 1. コンバージョンに至るアクセスログに均等に価値を割り振る
    , 1000.0 / COUNT(1) OVER(PARTITION BY session) AS fair_assign

    -- 2. コンバージョンに至るアクセスログの最初のページに価値を割り振る
    , CASE
        WHEN ROW_NUMBER() OVER(PARTITION BY session ORDER BY stamp ASC) = 1
          THEN 1000.0
        ELSE 0.0
      END AS first_assign

    -- 3. コンバージョンに至るアクセスログの最後のページに価値を割り振る
    , CASE
```

```
        WHEN ROW_NUMBER() OVER(PARTITION BY session ORDER BY stamp DESC) = 1
          THEN 1000.0
        ELSE 0.0
      END AS last_assign

    -- 4. コンバージョンに至るアクセスログの成果地点から近いページにより高く価値を割り振る
    , 1000.0
      * ROW_NUMBER() OVER(PARTITION BY session ORDER BY stamp ASC)
        -- 連番の合計値で割る ( N*(N+1)/2 )
      / (   COUNT(1) OVER(PARTITION BY session)
          * (COUNT(1) OVER(PARTITION BY session) + 1)
          / 2)
      AS decrease_assign

    -- 5. コンバージョンに至るアクセスログの成果地点から遠いページにより高く価値を割り振る
    , 1000.0
      * ROW_NUMBER() OVER(PARTITION BY session ORDER BY stamp DESC)
        -- 連番の合計値で割る ( N*(N+1)/2 )
      / (   COUNT(1) OVER(PARTITION BY session)
          * (COUNT(1) OVER(PARTITION BY session) + 1)
          / 2)
      AS increase_assign
  FROM activity_log_with_conversion_flag
  WHERE
    -- コンバージョンにつながるセッションのログのみを抽出
    has_conversion = 1
    -- 入力、確認、完了ページはページ価値を計算しない
    AND path NOT IN ('/input', '/confirm', '/complete')
)
SELECT
    session
  , asc_order
  , path
  , fair_assign AS fair_a
  , first_assign AS first_a
  , last_assign AS last_a
  , decrease_assign AS dec_a
  , increase_assign AS inc_a
FROM
  activity_log_with_conversion_assing
ORDER BY
  session, asc_order;
```

```
 session  | asc_order |     path      |  fair_a | first_a | last_a |  dec_a  |   inc_a
----------+-----------+---------------+---------+---------+--------+---------+---------
 1551f4aa |         1 | /search_list  |  500.00 |  1000.0 |    0.0 |  333.33 |  666.66
 1551f4aa |         2 | /detail       |  500.00 |     0.0 | 1000.0 |  666.66 |  333.33
 168c11f3 |         1 | /             |  166.66 |  1000.0 |    0.0 |   47.61 |  285.71
 168c11f3 |         2 | /search_list  |  166.66 |     0.0 |    0.0 |   95.23 |  238.09
 168c11f3 |         3 | /detail       |  166.66 |     0.0 |    0.0 |  142.85 |  190.47
 168c11f3 |         4 | /detail       |  166.66 |     0.0 |    0.0 |  190.47 |  142.85
 168c11f3 |         5 | /detail       |  166.66 |     0.0 |    0.0 |  238.09 |   95.23
 168c11f3 |         6 | /detail       |  166.66 |     0.0 | 1000.0 |  285.71 |   47.61
 361e0cac |         1 | /search_list  |  250.00 |  1000.0 |    0.0 |  100.00 |  400.00
 361e0cac |         2 | /detail       |  250.00 |     0.0 |    0.0 |  200.00 |  300.00
 361e0cac |         3 | /search_list  |  250.00 |     0.0 |    0.0 |  300.00 |  200.00
...
```

上記の出力結果を確認すると、1コンバージョン当たり1,000の価値が、fair_assignの場合はセッション中のアクセスに均等に割り振られ、first_assingの場合は最初のアクセスのみ、last_assignの場合は最後のアクセスのみに割り振られていることが分かります。

次のステップ 続いて、ページ価値の合計をパスごとに集計します。下記コード例のクエリの出力結果でページの価値を判断することも可能です。しかし、この状態ではページビューが多いページでは、ページの価値が高くなる傾向となります。

コード6.2.4.2: パスごとのページ価値合計を求めるクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
activity_log_with_session_conversion_flag AS (
  -- コード6.2.3.1参照
)
, activity_log_with_conversion_assing AS (
  -- コード6.2.4.1参照
)
, page_total_values AS (
  -- ページ価値の合計を計算
  SELECT
      path
    , SUM(fair_assign)  AS fair_assign
    , SUM(first_assign) AS first_assign
    , SUM(last_assign)  AS last_assign
  FROM
    activity_log_with_conversion_assing
  GROUP BY
    path
)
SELECT * FROM page_total_values;
```

```
      path      | sum_fair | sum_first | sum_last |  sum_dec |  sum_inc
----------------+----------+-----------+----------+----------+----------
 /detail        | 34347.61 |   27000.0 |  43000.0 | 36744.58 | 32194.37
 /search_list   |  6578.57 |    8000.0 |   2000.0 |  5419.69 |  7588.96
 /              |  3154.76 |    7000.0 |      0.0 |  2332.90 |  3881.38
 /search_input  |   919.04 |    3000.0 |      0.0 |   502.81 |  1335.28
```

次のステップ ページ価値の合計を求めたら、ページビューが少ないけれども、高いページ価値のページを見つけ出せるように、該当ページの訪問回数やページビューで割ります。

コード6.2.4.3: パスごとの平均ページ価値を求めるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_session_conversion_flag AS (
  -- コード6.2.3.1参照
)
, activity_log_with_conversion_assing AS (
  -- コード6.2.4.1参照
)
, page_total_values AS (
  -- コード6.2.4.2参照
)
, page_total_cnt AS (
  -- ページビューを計算
  SELECT
      path
    , COUNT(1) AS access_cnt  -- ページビュー
    -- 訪問回数で割りたい場合は下記を用いる
    -- , COUNT(DISTINCT session) AS access_cnt
  FROM
    activity_log
  GROUP BY
    path
)
SELECT
  -- 1訪問あたりのページ価値を計算
    s.path
  , s.access_cnt
  , v.sum_fair  / s.access_cnt AS avg_fair
  , v.sum_first / s.access_cnt AS avg_first
  , v.sum_last  / s.access_cnt AS avg_last
  , v.sum_dec   / s.access_cnt AS avg_dec
  , v.sum_inc   / s.access_cnt AS avg_inc
FROM
    page_total_cnt AS s
  JOIN
```

```
    page_total_values AS v
    ON s.path = v.path
ORDER BY
  s.access_cnt DESC;
```

```
      path      | access_cnt | avg_fair | avg_first | avg_last | avg_dec | avg_inc
----------------+------------+----------+-----------+----------+---------+---------
 /detail        |       5708 |     6.01 |      4.73 |     7.53 |    6.43 |    5.64
 /search_list   |       5066 |     1.29 |      1.57 |     0.39 |    1.06 |    1.49
 /              |       3554 |     0.88 |      1.96 |     0.00 |    0.65 |    1.09
 /search_input  |       2492 |     0.36 |      1.20 |     0.00 |    0.20 |    0.53
```

まとめ

アクセス解析ツールによっては、ページの価値を算出するロジックが限定されている場合があります。何を評価したいのかを明確にして、自由に計算できるようになれば、アクセス解析ツールの制限を受けることなく、新しい価値を提供することが可能になります。

関連項目	3-3-2 グループの中での順序を扱う

6-2-5 検索条件毎のユーザー行動を可視化する

SQL SIGN関数、SUM(CASE～END)、AVG(CASE～END)、LAG関数

分析 CTR、CVR

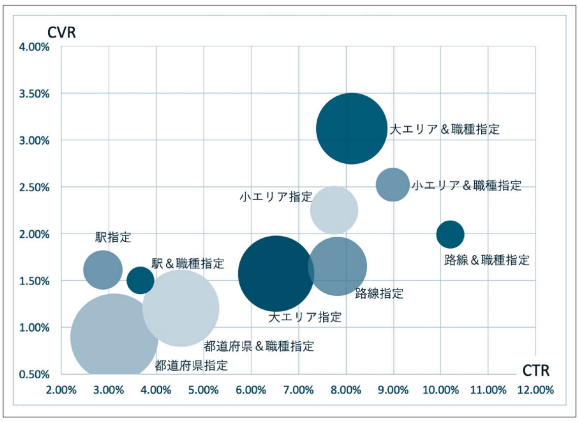
商品や情報を検索するサイトでは、複数の検索方法が提供されているケースが多々あります。例えば、ECサイトであれば、カテゴリやメーカー、価格帯などの絞込機能が提供されています。求人サイトの場合は、地域や業種、職種などの絞込機能が提供されています。

検索条件をより詳細に指定して検索するユーザーほど、モチベーションが明確で成果に結びつく割合が高くなる傾向があります。検索条件の指定が緩く曖昧、また幅広い条件の場合は、条件指定を促して、行動を促進させることで、成果の向上が見込めます。

検索条件ごとに詳細ページへ遷移した割合（CTR）と、詳細ページ閲覧後に成果に結び付いた割合（CVR）を2軸で表現したものを、次図6.2.5.1に示します。

各検索タイプにおける円の大きさは訪問回数を示します。CTRとCVRのいずれも高い、右上の検索条件寄りにユーザーを遷移させることで、成果の増加が見込めます。また、その検索条件をより利用してもらえるよう、表示するUIやその並び順などを工夫しましょう。

図 6.2.5.1：検索タイプ別のCTR、CVRと訪問回数



上図を作成するため、下表の結果を集計します。検索タイプは検索結果ページにおけるURLパラメーターを分解して分類します。CTRはその検索結果では、詳細画面へ1回遷移しても3回遷移しても、詳細画面への遷移数は1として扱います。成果も同様に1回の訪問で複数回行われても、1として扱います。

表 6.2.5.1：検索タイプ別のCTR、CVRと訪問回数

検索タイプ	訪問回数	CTR	詳細画面遷移セッション数	CVR	成果セッション数
都道府県	1,683	51.4%	866	1.2%	11
大エリア	1,199	50.9%	611	1.1%	7
大エリア＋職種指定	603	56.0%	338	0.2%	1

上表に掲載されている項目を集計するクエリを、次のコード例に示します。

詳細ページへの遷移を意味するクリックフラグと、最終的にコンバージョンまで達したかどうかを判断するコンバージョンフラグを、ウィンドウ関数を用いて計算します。

コード6.2.5.1：クリックフラグとコンバージョンフラグを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_session_click_conversion_flag AS (
  SELECT
      session
    , stamp
    , path
    , search_type
      -- 詳細ページより前のアクセスにフラグを立てる
    , SIGN(SUM(CASE WHEN path = '/detail' THEN 1 ELSE 0 END)
          OVER(PARTITION BY session ORDER BY stamp DESC
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW))
      AS has_session_click
      -- コンバージョンしたページより前のアクセスにフラグを立てる
    , SIGN(SUM(CASE WHEN path = '/complete' THEN 1 ELSE 0 END)
          OVER(PARTITION BY session ORDER BY stamp DESC
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW))
      AS has_session_conversion
  FROM
    activity_log
)
SELECT
    session
  , stamp
  , path
  , search_type
  , has_session_click AS click
  , has_session_conversion AS cnv
FROM
  activity_log_with_session_click_conversion_flag
ORDER BY
  session, stamp
;
```

```
 session  |        stamp        |     path       |   search_type    | click | cnv
----------+---------------------+----------------+------------------+-------+-----
 0006fbb5 | 2016-12-29 21:47:15 | /search_list   | Area-L           |     1 |   0
 0006fbb5 | 2016-12-29 21:48:21 | /              |                  |     1 |   0
 0006fbb5 | 2016-12-29 21:49:58 | /search_input  |                  |     1 |   0
 0006fbb5 | 2016-12-29 21:50:52 | /search_list   | Area-S-with-Job  |     1 |   0
 0006fbb5 | 2016-12-29 21:51:49 | /detail        |                  |     1 |   0
 0007a5f9 | 2016-12-29 21:47:17 | /              |                  |     1 |   0
 0007a5f9 | 2016-12-29 21:47:17 | /detail        |                  |     1 |   0
 0007a5f9 | 2016-12-29 21:48:19 | /search_list   | Station-with-Job |     0 |   0
...
 05e9cc2b | 2016-12-29 21:47:15 | /search_list   | Area-S-with-Job  |     1 |   1
 05e9cc2b | 2016-12-29 21:48:02 | /search_list   | Area-L           |     1 |   1
 05e9cc2b | 2016-12-29 21:49:04 | /search_input  |                  |     1 |   1
```

```
 05e9cc2b | 2016-12-29 21:49:50 | /search_list   | Pref            |    1 |   1
 05e9cc2b | 2016-12-29 21:50:21 | /detail        |                 |    1 |   1
 05e9cc2b | 2016-12-29 21:50:37 | /input         |                 |    1 |   1
 05e9cc2b | 2016-12-29 21:51:03 | /confirm       |                 |    1 |   1
 05e9cc2b | 2016-12-29 21:52:39 | /complete      |                 |    1 |   1
...
```

上述のクエリでクリックフラグとコンバージョンフラグが計算できたら、続いて、下記コード例に示すクエリで検索ログのみに絞り込み、検索条件で集約して、アクセス数とクリック数、コンバージョン数、CTR、CVRを計算します。

コード6.2.5.2：検索タイプ別のCTR, CVRを計算するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
activity_log_with_session_click_conversion_flag AS (
  -- コード6.2.5.1参照
)
SELECT
    search_type
  , COUNT(1) AS count
  , SUM(has_session_click) AS detail
  , AVG(has_session_click) AS ctr
  , SUM(CASE WHEN has_session_click = 1 THEN has_session_conversion END) AS conversion
  , AVG(CASE WHEN has_session_click = 1 THEN has_session_conversion END) AS cvr
FROM
  activity_log_with_session_click_conversion_flag
WHERE
  -- 検索ログのみに絞り込む
  path = '/search_list'
-- 検索条件で集約する
GROUP BY
  search_type
ORDER BY
  count DESC
;
```

```
 search_type      | count | detail |   ctr  | conversion |   cvr
------------------+-------+--------+--------+------------+--------
 Pref             |  1896 |    691 | 0.3644 |          9 | 0.0130
 Area-L           |  1239 |    474 | 0.3825 |         10 | 0.0210
 Area-S           |  1252 |    493 | 0.3937 |         15 | 0.0304
 Line-with-Job    |   661 |    256 | 0.3872 |          3 | 0.0117
 Pref-with-Job    |   656 |    252 | 0.3841 |          8 | 0.0317
 Area-L-with-Job  |   623 |    232 | 0.3723 |          9 | 0.0387
 Station-with-Job |   614 |    246 | 0.4006 |          1 | 0.0040
 Station          |   609 |    222 | 0.3645 |          3 | 0.0135
```

```
 Area-S-with-Job  |   579 |    222 | 0.3834 |          5 | 0.0225
 Line             |   563 |    201 | 0.3570 |          5 | 0.0248
```

上記のコード例では、1回の訪問で複数の検索タイプで検索した場合、それぞれカウントされます。全体観を把握するケースでは利用できますが、成果直前の検索結果のみを対象にする場合は、上記WITH句を下記コード例に示す通りに修正して実行してください。

LAG関数で詳細ページへのアクセス直近のアクセスにフラグを立てます。なお、このLAG関数のOVER句で「ORDER BY stamp DESC」とタイムスタンプの降順にするのは、has_session_conversionカラムのOVER句とソート条件を揃えることで、ソート処理を効率的に実行するためです。

コード6.2.5.3：クリックフラグを直近のページのみに限定したクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_session_click_conversion_flag AS (
  SELECT
      session
    , stamp
    , path
    , search_type
      -- 詳細ページの直近のアクセスにフラグを立てる
    , CASE
        WHEN LAG(path) OVER(PARTITION BY session ORDER BY stamp DESC) = '/detail'
          THEN 1
        ELSE 0
      END AS has_session_click
      -- コンバージョンしたページより前のアクセスにフラグを立てる
    , SIGN(
        SUM(CASE WHEN path = '/complete' THEN 1 ELSE 0 END)
          OVER(PARTITION BY session ORDER BY stamp DESC
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      ) AS has_session_conversion
  FROM
    activity_log
)
SELECT
    session
  , stamp
  , path
  , search_type
  , has_session_click AS click
  , has_session_conversion AS cnv
FROM
  activity_log_with_session_click_conversion_flag
ORDER BY
  session, stamp
;
```

```
 session  |        stamp        |     path       |   search_type    | click | cnv
----------+---------------------+----------------+------------------+-------+-----
 0006fbb5 | 2016-12-29 21:47:15 | /search_list   | Area-L           |     0 |   0
 0006fbb5 | 2016-12-29 21:48:21 | /              |                  |     0 |   0
 0006fbb5 | 2016-12-29 21:49:58 | /search_input  |                  |     0 |   0
 0006fbb5 | 2016-12-29 21:50:52 | /search_list   | Area-S-with-Job  |     1 |   0
 0006fbb5 | 2016-12-29 21:51:49 | /detail        |                  |     0 |   0
 0007a5f9 | 2016-12-29 21:47:17 | /              |                  |     0 |   0
 0007a5f9 | 2016-12-29 21:47:17 | /detail        |                  |     0 |   0
 0007a5f9 | 2016-12-29 21:48:19 | /search_list   | Station-with-Job |     0 |   0
...
 05e9cc2b | 2016-12-29 21:47:15 | /search_list   | Area-S-with-Job  |     0 |   1
 05e9cc2b | 2016-12-29 21:48:02 | /search_list   | Area-L           |     0 |   1
 05e9cc2b | 2016-12-29 21:49:04 | /search_input  |                  |     0 |   1
 05e9cc2b | 2016-12-29 21:49:50 | /search_list   | Pref             |     1 |   1
 05e9cc2b | 2016-12-29 21:50:21 | /detail        |                  |     0 |   1
 05e9cc2b | 2016-12-29 21:50:37 | /input         |                  |     0 |   1
 05e9cc2b | 2016-12-29 21:51:03 | /confirm       |                  |     0 |   1
 05e9cc2b | 2016-12-29 21:52:39 | /complete      |                  |     0 |   1
 05e9cc2b | 2016-12-29 21:54:05 | /              |                  |     1 |   0
 05e9cc2b | 2016-12-29 21:54:46 | /detail        |                  |     0 |   0
...
```

ワンポイント

条件を細かく指定できるものの、検索でのヒットする項目が少なかったり皆無だった場合、ユーザーは離脱してしまう可能性があります。各検索条件とヒット件数のバランスを考慮して、カテゴリの見直しも視野に入れて改善を進めましょう。

関連項目
3-4-3 条件のフラグを0と1で表現する
8-1-1 NoMatch率とそのワードを集計する

6-2-6 フォールアウトレポートを用いてユーザーの回遊を可視化する

SQL LAG関数、FIRST_VALUE関数　　分析 フォールアウト

トップページから検索条件の入力、そして検索結果一覧から遷移先の詳細画面、入力フォームとその確認画面や完了画面など、一連のユーザー回遊の流れのどこで離脱しているのか、どれぐらい遷移しているのかを調査し改善することで、全体のCVRを向上させることが可能です。

前項までのSQLでは、該当ページを閲覧したユーザーが最終的に成果に至ったかを確認する方法を紹介しましたが、本項ではその中間地点の到達率を含めて集計する方法を紹介します。

ある地点からある地点への遷移をフォールスルーと呼び、ある地点での離脱をフォールアウトと呼びます。例えば、下図に示す通り、これらの複数地点での遷移率をまとめたものをフォールアウトレポートと呼びます。

図6.2.6.1：フォールアウトレポート

ステップ	訪問回数	先頭からの遷移率	直前からの遷移率
/	30,841	100.0%	100.0%
/list	15,603	50.6%	50.6%
/detail	12,042	39.0%	77.2%
/input	565	1.8%	4.7%
/complete	188	0.6%	33.3%

上図のURLに記載した地点間の遷移はアクセスしたユーザーで、他のページを経由もしくは直後に遷移したかは問わず、次の地点に到達した訪問回数を数えます。

必ず直後に次に地点に遷移したことを意味するわけではありません。直前・直後の遷移を知りたい場合は、後述の「6-2-7 サイト内のユーザーフローを把握する」(P.291) を参照してください。

上述の順序を考慮するデータをSQLで取り扱うには少々工夫が必要です。まず、次のコード例に示す通り、ステップの順序を番号で明示したマスターテーブル（mst_fallout_step）を作成し、ログデータと結合します。さらに、セッションごとにステップ順（とURL）で集約し、フォールアウトレポートで必要なログに絞り込むためのカラムをいくつか計算します。

コード6.2.6.1：フォールアウトのステップ順をアクセスログと結合するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_fallout_step AS (
  -- フォールアウトのステップとパスのマスタ
           SELECT 1 AS step, '/'            AS path
  UNION ALL SELECT 2 AS step, '/search_list' AS path
  UNION ALL SELECT 3 AS step, '/detail'      AS path
  UNION ALL SELECT 4 AS step, '/input'       AS path
  UNION ALL SELECT 5 AS step, '/complete'    AS path
)
, activity_log_with_fallout_step AS (
  SELECT
      l.session
    , m.step
    , m.path
      -- そのステップにおけるパスへの初回/最終アクセス時刻
    , MAX(l.stamp) AS max_stamp
    , MIN(l.stamp) AS min_stamp
  FROM
      mst_fallout_step AS m
    JOIN
      activity_log AS l
      ON m.path = l.path
  GROUP BY
    -- セッションごとに、ステップ順とパスで集約
    l.session, m.step, m.path
)
, activity_log_with_mod_fallout_step AS (
  SELECT
      session
    , step
    , path
    , max_stamp
      -- 直近のステップにおけるパスへの初回アクセス時刻
    , LAG(min_stamp)
        OVER(PARTITION BY session ORDER BY step)
      -- ■ SparkSQLの場合、LAG関数のフレーム指定が必要
      -- OVER(PARTITION BY session ORDER BY step
      --   ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
      AS lag_min_stamp
      -- セッション中のステップ順の最小値
    , MIN(step) OVER(PARTITION BY session) AS min_step
      -- そのステップに至るまでの累計ステップ数
    , COUNT(1)
        OVER(PARTITION BY session ORDER BY step
          ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      AS cum_count
  FROM
    activity_log_with_fallout_step
)
```

```
SELECT
  *
FROM
  activity_log_with_mod_fallout_step
ORDER BY
  session, step
;
```

```
 session  | step |     path      |   max_stamp   | lag_min_stamp | min_step | cum_count
----------+------+---------------+---------------+---------------+----------+-----------
 0006fbb5 |    1 | /             | ... 21:48:21 |               |        1 |         1
 0006fbb5 |    2 | /search_list  | ... 21:50:52 | ... 21:48:21  |        1 |         2
 0006fbb5 |    3 | /detail       | ... 21:51:49 | ... 21:47:15  |        1 |         3
 0007a5f9 |    1 | /             | ... 21:47:17 |               |        1 |         1
 0007a5f9 |    2 | /search_list  | ... 21:48:19 | ... 21:47:17  |        1 |         2
 0007a5f9 |    3 | /detail       | ... 21:47:17 | ... 21:48:19  |        1 |         3
 00090424 |    3 | /detail       | ... 21:47:16 |               |        3 |         1
 000db23c |    3 | /detail       | ... 21:47:17 |               |        3 |         1
 00175f4b |    2 | /search_list  | ... 21:47:16 |               |        2 |         1
 003106eb |    3 | /detail       | ... 21:47:15 |               |        3 |         1
 003241dc |    1 | /             | ... 21:47:17 |               |        1 |         1
 003241dc |    3 | /detail       | ... 21:48:08 | ... 21:47:17  |        1 |         2
 003ce216 |    2 | /search_list  | ... 21:47:17 |               |        2 |         1
 003ce216 |    3 | /detail       | ... 21:47:40 | ... 21:47:17  |        2 |         2
 003ce216 |    4 | /input        | ... 21:48:46 | ... 21:47:40  |        2 |         3
 00406196 |    2 | /search_list  | ... 21:47:17 |               |        2 |         1
 004deefc |    3 | /detail       | ... 21:47:16 |               |        3 |         1
 00525285 |    2 | /search_list  | ... 21:47:17 |               |        2 |         1
 00525285 |    3 | /detail       | ... 21:48:55 | ... 21:47:17  |        2 |         2
 005b0933 |    1 | /             | ... 21:47:15 |               |        1 |         1
...
```

上記コードの出力結果から、フォールアウトレポートで必要なログに絞り込みを行います。そのための条件は少し複雑ですが、以下に示す通りです。

1 セッション中にステップ順=1のURLにアクセスしている
2 現在のステップ順が、そのステップに至るまでの累計ステップ数と同じ
3 直近のステップの初回アクセス時刻が、現在のステップの最終アクセス時刻より前である

1 は、そもそもステップ順が1のURL（今回の場合は「/」のトップページ）にアクセスしていないログは、フォールアウトレポートの対象外であることを意味します。
2 は、ステップ順=2のページ（/search_list）をスキップして、ステップ順=3のページ（/detail）に直接アクセスしているログなどを排除するための条件です。

3 は、ステップ順=3のページにアクセスした後にステップ順=2にアクセスしているログでは、ステップ順=3のログを排除することを意味します。その場合の条件として、「ステップ順=2の初回アクセス時刻がステップ順=3の最終アクセス時刻より前である」条件を付与します。ただし、ステップ順=1の場合、直近のステップが存在しないため、例外的に扱う必要があります。

上述の絞り込み条件をSQLで表現したクエリを、下記コード例に示します。

コード6.2.6.2：フォールアウトレポートに必要なログを絞り込むクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_fallout_step AS (
  -- コード6.2.6.1参照
)
, activity_log_with_fallout_step AS (
  -- コード6.2.6.1参照
)
, activity_log_with_mod_fallout_step AS (
  -- コード6.2.6.1参照
)
, fallout_log AS (
  -- フォールアウトレポートに使用するログのみを抽出する
  SELECT
      session
    , step
    , path
  FROM
    activity_log_with_mod_fallout_step
  WHERE
        -- セッション中にステップ順が1のURLにアクセスしている
        min_step = 1
        -- 現在のステップ順が、そのステップに至るまでの累計ステップ数と同じ
    AND step = cum_count
        -- 直近のステップの初回アクセス時刻について
        -- NULL または 現在のステップの最終アクセス時刻より前である
    AND (lag_min_stamp IS NULL
         OR max_stamp >= lag_min_stamp)
)
SELECT
  *
FROM
  fallout_log
ORDER BY
  session, step
;
```

```
 session  | step |     path
----------+------+---------------
 0006fbb5 |    1 | /
 0006fbb5 |    2 | /search_list
 0006fbb5 |    3 | /detail
 0007a5f9 |    1 | /
 0007a5f9 |    2 | /search_list
 003241dc |    1 | /
 005b0933 |    1 | /
 005b0933 |    2 | /search_list
 005fce13 |    1 | /
 00745cc1 |    1 | /
 00745cc1 |    2 | /search_list
 007da4b9 |    1 | /
 0087634d |    1 | /
 0087634d |    2 | /search_list
 009a1f8f |    1 | /
 009a1f8f |    2 | /search_list
 00d5b1cd |    1 | /
 00d5b1cd |    3 | /detail
 00dc5d86 |    1 | /
 00dc5d86 |    2 | /search_list
..
```

上記コード例の出力結果までログを整形すれば、あとはステップ順とURLで集約し、アクセス数やページ遷移率を計算するだけです。下記コード例に、ステップごとのアクセス数やページ遷移率を計算するクエリを示します。

コード6.2.6.3：フォールアウトレポートを出力するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_fallout_step AS (
  -- コード6.2.6.1参照
)
, activity_log_with_fallout_step AS (
  -- コード6.2.6.1参照
)
, fallout_log AS (
  -- コード6.2.6.2参照
)
SELECT
    step
  , path
  , COUNT(1) AS count
    -- ステップ順=1のURLからの遷移率
  , 100.0 * COUNT(1)
    / FIRST_VALUE(COUNT(1))
```

```
        OVER(ORDER BY step ASC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS first_trans_rate
    -- 直近のステップからの遷移率
  , 100.0 * COUNT(1)
    / LAG(COUNT(1)) OVER(ORDER BY step ASC)
    -- ■ SparkSQLの場合､LAG関数のフレーム指定が必要
    -- / LAG(COUNT(1)) OVER(ORDER BY step ASC ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
    AS step_trans_rate
FROM
  fallout_log
GROUP BY
  step, path
ORDER BY
  step
;
```

```
 step |     path      | count |    first_trans_rate    |   step_trans_rate
------+---------------+-------+------------------------+---------------------
    1 | /             |  3445 |  100.0000000000000000 |
    2 | /search_list  |  2128 |   61.7706821480406386 | 61.7706821480406386
    3 | /detail       |  1505 |   43.6865021770682148 | 70.7236842105263158
    4 | /input        |    92 |    2.6705370101596517 |  6.1129568106312292
    5 | /complete     |    14 |    0.4063860667634252 | 15.2173913043478261
```

まとめ

フォールアウトレポートはWebサイト内におけるユーザーの回遊を大局的に把握する際に、非常に有効なレポートです。成果に至ったユーザー、至らなかったユーザーの違いなど、ユーザーのセグメントを分けて集計することで、新たな発見が得られるかもしれません。

関連項目	3-3-2 グループの中での順序を扱う 3-4-5 擬似的なテーブルを作成する 6-3-2 入力～確認～完了までの遷移率を集計する

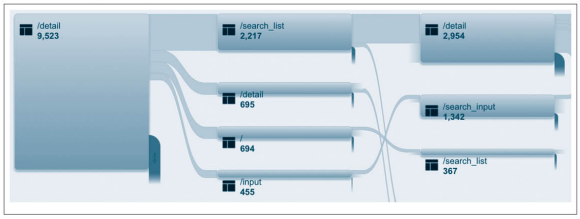
6-2-7 サイト内のユーザーフローを把握する

SQL LAG関数、LEAD関数、SUMウィンドウ関数　分析 ユーザーフロー

ユーザーがどのようにWebサイトを回遊しているのか把握することで、サイトマップの見直し、メニューやモジュール配置の検討に活かすことができます。ユーザーの利用する機能がサービス提供側が想定しているものと異なることが分かれば、サービス利用者の利用に即した形でサイト構成を変更したり、逆にサービス提供側の意図をユーザーの回遊に反映し、モデルとなるサイト利用形態に近付けるように検討できます。

前項の「6-2-6 フォールアウトレポートを用いてユーザーの回遊を可視化する」(P.285) では、特定ページを閲覧したユーザーのなかで、指定ページに遷移したユーザーがどの程度存在するか確認する方法を紹介しましたが、本項では、より詳細に把握するため、ユーザーフローを用いて分析する方法を紹介します。

図6.2.7.1：Google Analyticsにおけるユーザーフロー図



上図に示すユーザーフロー図の結果を得るために、最初に何を分析したいかを決定し、起点とするページを決めます。

- トップページからどのように回遊しているのか
  - まず検索を利用しているか
  - サービスを紹介するページを見ているか
- 詳細画面の前後では、どのような行動をしているのか
  - 検索結果ではなく、オススメモジュールから遷移しているのか
  - 検索結果から詳細画面に遷移しているのか
  - 詳細画面表示後、お気に入り追加、カート追加ページへ遷移するのか
  - 詳細画面表示後、レコメンドにより別の詳細画面に遷移するのか

次のページを集計する

起点とするページを設定したら、下記に示す基準点からその後どのように回遊したかをまとめた表を集計するSQLを示します。

表6.2.7.1：ユーザーフローを表形式で表現(次のページ)

基準点	訪問回数	次のページ1	訪問回数1	遷移率1	次のページ2	訪問回数2	遷移率2
/detail	16,930	/detail	5,683	33.6%	/	1,793	31.6%
					/list	2,426	42.7%
					NULL	1,464	25.8%
		/list	3,048	18.0%	/map	1,532	50.3%
					/	493	16.2%
					NULL	1,023	33.6%
		...	...	...	...	...	...
		NULL	5,301	31.3%	NULL	5,301	31.3%

上記は、ユーザーフローの/detailページ訪問時を基準点として、その後の遷移を2ステップまでを集計し、表にまとめたものです。遷移率はその前のページの訪問回数を母数として算出します。次のページがNULLレコードは、次のページを閲覧したログがないことを意味しており、該当レコードはユーザーの離脱を示します。

コード6.2.7.1：/detailページ以降のユーザーフローを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_lead_path AS (
  SELECT
     session
   , stamp
   , path AS path0
     -- 1つ後にアクセスしたパスを取得する
   , LEAD(path, 1) OVER(PARTITION BY session ORDER BY stamp ASC) AS path1
     -- 2つ後にアクセスしたパスを取得する
   , LEAD(path, 2) OVER(PARTITION BY session ORDER BY stamp ASC) AS path2
  FROM
    activity_log
)
, raw_user_flow AS(
  SELECT
      path0
      -- 起点となるパスへのアクセス数
    , SUM(COUNT(1)) OVER() AS count0
      -- 1つ後にアクセスしたパス(存在しない場合は文字列のNULL)
    , COALESCE(path1, 'NULL') AS path1
      -- 1つ後にアクセスしたパスへのアクセス数
    , SUM(COUNT(1)) OVER(PARTITION BY path0, path1) AS count1
```

```
      -- 2つ後にアクセスしたパス（存在しない場合は文字列のNULL）
    , COALESCE(path2, 'NULL') AS path2
      -- 2つ後にアクセスしたパスへのアクセス数
    , COUNT(1) AS count2
  FROM
    activity_log_with_lead_path
  WHERE
    -- 詳細ページを起点とする
    path0 = '/detail'
  GROUP BY
    path0, path1, path2
)
SELECT
    path0
  , count0
  , path1
  , count1
  , 100.0 * count1 / count0 AS rate1
  , path2
  , count2
  , 100.0 * count2 / count1 AS rate2
FROM
  raw_user_flow
ORDER BY
    count1 DESC
  , count2 DESC
;
```

```
  path0   | count0 |     path1      | count1 | rate1 |     path2      | count2 | rate2
----------+--------+----------------+--------+-------+----------------+--------+------
 /detail  |   9365 | NULL           |   3663 | 39.11 | NULL           |   3663 | 100.00
 /detail  |   9365 | /search_list   |   2164 | 23.10 | NULL           |    697 |  32.20
 /detail  |   9365 | /search_list   |   2164 | 23.10 | /detail        |    691 |  31.93
 /detail  |   9365 | /search_list   |   2164 | 23.10 | /search_list   |    288 |  13.30
 /detail  |   9365 | /search_list   |   2164 | 23.10 | /              |    283 |  13.07
 /detail  |   9365 | /search_list   |   2164 | 23.10 | /search_input  |    191 |   8.82
 /detail  |   9365 | /search_list   |   2164 | 23.10 | /input         |     14 |   0.64
 /detail  |   9365 | /detail        |   1717 | 18.33 | NULL           |    582 |  33.89
 /detail  |   9365 | /detail        |   1717 | 18.33 | /search_list   |    445 |  25.91
 /detail  |   9365 | /detail        |   1717 | 18.33 | /detail        |    328 |  19.10
 /detail  |   9365 | /detail        |   1717 | 18.33 | /              |    280 |  16.30
 /detail  |   9365 | /detail        |   1717 | 18.33 | /input         |     73 |   4.25
 /detail  |   9365 | /detail        |   1717 | 18.33 | /search_input  |      9 |   0.52
```

前述の出力結果は、重複したレコードが何度も登場するため、どこまでが1つのページに対する流入か分かりにくくなっています。レポートツールなどでユーザーフロー図を出力するケースでは必要ありませんが、重複部分をSQLで取り除いて、可読性が高い形式に整形することも可能です。

下記コード例に、1つ上のレコードと同じ値を表示しない形式に整形するクエリを示します。LAG関数を用いて、1つ上のレコードのパス名を取得して値が異なる場合のみ、そのレコードの値を出力しています。ただし、先頭のレコードはLAG関数の値がNULLとなる可能性があるため、COALESCE関数を用いて適切なデフォルト値を設定しています。

コード6.2.7.2：一つ上のレコードと同じ値を表示しない形に整形するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
activity_log_with_lead_path AS (
  -- コード6.2.7.1
)
, raw_user_flow AS(
  -- コード6.2.7.1
)
SELECT
    CASE
      WHEN
        COALESCE(
          -- 一つ上のレコードのpath0を取得する(存在しない場合はNOT FOUNDとする)
          LAG(path0)
            OVER(ORDER BY count1 DESC, count2 DESC)
        , 'NOT FOUND') <> path0
      THEN path0
    END AS path0
  , CASE
      WHEN
        COALESCE(
          LAG(path0)
            OVER(ORDER BY count1 DESC, count2 DESC)
        , 'NOT FOUND') <> path0
      THEN count0
    END AS count0
  , CASE
      WHEN
        COALESCE(
        -- 一つ上のレコードの複数の値を取得するために、文字列として連結して取得
        -- ■ PostgreSQL, Redshiftの場合、文字列連結に||演算子を用いる
        -- ■ Hive, BigQuery, SparkSQLの場合、文字列連結にconcat関数を用いる
        --   (以下同様)
          LAG(path0 || path1)
            OVER(ORDER BY count1 DESC, count2 DESC)
        , 'NOT FOUND') <> (path0 || path1)
      THEN path1
    END AS page1
  , CASE
```

```
      WHEN
        COALESCE(
          LAG(path0 || path1)
            OVER(ORDER BY count1 DESC, count2 DESC)
        , 'NOT FOUND') <> (path0 || path1)
      THEN count1
    END AS count1
  , CASE
      WHEN
        COALESCE(
          LAG(path0 || path1)
            OVER(ORDER BY count1 DESC, count2 DESC)
        , 'NOT FOUND') <> (path0 || path1)
      THEN 100.0 * count1 / count0
    END AS rate1
  , CASE
      WHEN
        COALESCE(
          LAG(path0 || path1 || path2)
            OVER(ORDER BY count1 DESC, count2 DESC)
        , 'NOT FOUND') <> (path0 || path1 || path2)
      THEN path2
    END AS page2
  , CASE
      WHEN
        COALESCE(
          LAG(path0 || path1 || path2)
            OVER(ORDER BY count1 DESC, count2 DESC)
        , 'NOT FOUND') <> (path0 || path1 || path2)
      THEN count2
    END AS count2
  , CASE
      WHEN
        COALESCE(
          LAG(path0 || path1 || path2)
            OVER(ORDER BY count1 DESC, count2 DESC)
        , 'NOT FOUND') <> (path0 || path1 || path2)
      THEN 100.0 * count2 / count1
    END AS rate2
FROM
  raw_user_flow
ORDER BY
    count1 DESC
  , count2 DESC
;
```

```
  path0   | count0 |     path1      | count1 | rate1 |     path2      | count2 | rate2
----------+--------+----------------+--------+-------+----------------+--------+------
 /detail  |   9365 | NULL           |   3663 | 39.11 | NULL           |   3663 | 100.00
          |        | /search_list   |   2164 | 23.10 | NULL           |    697 |  32.20
          |        |                |        |       | /detail        |    691 |  31.93
          |        |                |        |       | /search_list   |    288 |  13.30
          |        |                |        |       | /              |    283 |  13.07
          |        |                |        |       | /search_input  |    191 |   8.82
          |        |                |        |       | /input         |     14 |   0.64
          |        | /detail        |   1717 | 18.33 | NULL           |    582 |  33.89
          |        |                |        |       | /search_list   |    445 |  25.91
          |        |                |        |       | /detail        |    328 |  19.10
          |        |                |        |       | /              |    280 |  16.30
          |        |                |        |       | /input         |     73 |   4.25
          |        |                |        |       | /search_input  |      9 |   0.52
```

前のページを集計する

前述の次ページ集計と同様に、前のページを集計します。

下表は、ユーザーフローの/detailページ訪問時を基準点として、その前の遷移を2ステップまで集計したものです。前ページがNULLのレコードは、前ページを閲覧したログがないレコードであり、該当レコードは直接の訪問、もしくは他サイトから参照され着地したページであることを示します。

上記の集計結果を得るためのクエリを下記コード例に示します。基本的には、前述の「次のページを集計する」で紹介したコード例（6.2.7.2）のLEAD関数をLAG関数に変更するだけで対応できます。

表6.2.7.2：ユーザーフローを表形式で表現(前のページ)

前のページ2	訪問回数2	遷移率2	前のページ1	訪問回数1	遷移率1	基準点	訪問回数
/	3,781	61.1%	**/list**	6,191	36.6%	**/detail**	16,930
/detail	1,983	32.0%					
NULL	427	6.9%					
/list	302	15.2%	/	1,981	11.7%		
/detail	692	34.9%					
NULL	987	49.8%					
…	…	…	**…**	…	…		
NULL	6,049	35.7%	**NULL**	6,049	35.7%		

コード6.2.7.3：/detailページ以前のユーザーフローを計算するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
activity_log_with_lag_path AS (
  SELECT
    session
  , stamp
  , path AS path0
```

```
    -- 1つ前にアクセスしたパスを取得する(存在しない場合は文字列のNULL)
  , COALESCE(LAG(path, 1) OVER(PARTITION BY session ORDER BY stamp ASC), 'NULL') AS path1
    -- 2つ前にアクセスしたパスを取得する(存在しない場合は文字列のNULL)
  , COALESCE(LAG(path, 2) OVER(PARTITION BY session ORDER BY stamp ASC), 'NULL') AS path2
  FROM
    activity_log
)
, raw_user_flow AS(
  SELECT
      path0
      -- 起点となるパスへのアクセス数
    , SUM(COUNT(1)) OVER() AS count0
    , path1
      -- 1つ前にアクセスしたパスへのアクセス数
    , SUM(COUNT(1)) OVER(PARTITION BY path0, path1) AS count1
    , path2
      -- 2つ前にアクセスしたパスへのアクセス数
    , COUNT(1) AS count2
  FROM
    activity_log_with_lag_path
  WHERE
    -- 詳細ページを起点とする
    path0 = '/detail'
  GROUP BY
    path0, path1, path2
)
SELECT
    path2
  , count2
  , 100.0 * count2 / count1 AS rate2
  , path1
  , count1
  , 100.0 * count1 / count0 AS rate1
  , path0
  , count0
FROM
  raw_user_flow
ORDER BY
    count1 DESC
  , count2 DESC
;
```

```
     path2     | count2 |  rate2 |     path1     | count1 | rate1 |  path0  | count0
---------------+--------+--------+---------------+--------+-------+---------+--------
 NULL          |   3539 | 100.00 | NULL          |   3539 | 37.78 | /detail |   9365
 NULL          |    903 |  30.47 | /search_list  |   2963 | 31.63 | /detail |   9365
 /detail       |    691 |  23.32 | /search_list  |   2963 | 31.63 | /detail |   9365
 /             |    548 |  18.49 | /search_list  |   2963 | 31.63 | /detail |   9365
```

```
/search_list  |   462 |  15.59 | /search_list |   2963 | 31.63 | /detail |   9365
/search_input |   355 |  11.98 | /search_list |   2963 | 31.63 | /detail |   9365
/input        |     3 |   0.10 | /search_list |   2963 | 31.63 | /detail |   9365
/confirm      |     1 |   0.03 | /search_list |   2963 | 31.63 | /detail |   9365
NULL          |   614 |  35.76 | /detail      |   1717 | 18.33 | /detail |   9365
/search_list  |   573 |  33.37 | /detail      |   1717 | 18.33 | /detail |   9365
/detail       |   328 |  19.10 | /detail      |   1717 | 18.33 | /detail |   9365
/             |   189 |  11.00 | /detail      |   1717 | 18.33 | /detail |   9365
/search_input |     9 |   0.52 | /detail      |   1717 | 18.33 | /detail |   9365
/input        |     3 |   0.17 | /detail      |   1717 | 18.33 | /detail |   9365
/complete     |     1 |   0.05 | /detail      |   1717 | 18.33 | /detail |   9365
```

ワンポイント

本項で紹介したSQLを元に、パソコンユーザーとスマートフォンユーザー、広告経由での流入をはじめ、参照元別や入口ページ別など、さまざまなセグメント別にユーザーフローを抽出し、サイト回遊の特徴を見つけ出し、課題の発見と解決に役立てましょう。

関連項目 | 3-3-2 グループの中での順序を扱う

6-2-8 ページの読了率を集計する

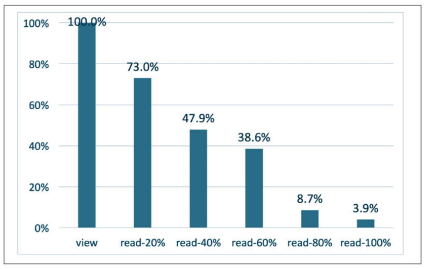
SQL SUM(CASE～END)　　分析 読了率

直帰率、離脱率、ページビューを確認しても、ユーザーがページを最後まで閲覧し、満足して離脱したのか、すぐに離脱したのかは分かりません。情報サイトを例にすると、サイトに用意している記事を最後まで閲覧してもらえていない状態では、ユーザーに価値を提供できているとは言えません。ユーザーに価値をしっかりと提供することで、将来のより高い価値へと繋がります。

次に示す図の通り、ページを最後まで読み進めている割合（＝読了率）を集計し、ユーザーにしっかり価値を提供できているか確認しましょう。

例えば、特定コンテンツのみ読了率が高かったり、低かったりするのであれば、ユーザーの求めるコンテンツがどのようなものか調べる材料となります。また、全体的に読了率が低いのであれば、ページが読みづらい状態なのではないか、などと仮説を立てることができます。

図 6.2.8.1 : 読了率の集計



読了率を集計するためには、ページ閲覧のログの他に、どこまで閲覧されたかを把握するために、下記に示すログデータが必要です。そのためには、Webサイト内にJavaScriptでログを送信する仕組みを用意する必要があります。

データ 6.2.8.1 : 読了率を集計するためのログ　例1

```
         stamp         | session  |  action   |          url
-----------------------+----------+-----------+----------------------
 2016-12-29 21:45:47 | afbd3d09 | view      | http://...?id=news341
 2016-12-29 21:45:47 | df6eb25d | view      | http://...?id=news731
 2016-12-29 21:45:56 | df6eb25d | read-20%  | http://...?id=news731
 2016-12-29 21:46:05 | df6eb25d | read-40%  | http://...?id=news731
 2016-12-29 21:46:13 | df6eb25d | read-60%  | http://...?id=news731
 2016-12-29 21:46:22 | df6eb25d | read-80%  | http://...?id=news731
 2016-12-29 21:46:25 | df6eb25d | read-100% | http://...?id=news731
...
```

上述のログは、JavaScriptで該当ページの長さを判断して、30%閲覧、60%閲覧などのログを送信する仕組みを実装すれば可能ですが、記事の長さでログが送信されるページの位置が異なります。例えば、ある記事ではread-60%がもう末尾まで閲覧された状態、ある記事ではまだ記事本文を閲覧途中の状態と、定義が異なってしまう可能性があります。

そこで、ページの高さを割合で示すのではなく、モジュールが表示された段階でログを送信することで、定義を揃えることが可能です。

データ6.2.8.2：読了率を集計するためのログ　例2

```
        stamp        | session  |      action      |          url
---------------------+----------+------------------+-----------------------
 2016-12-29 21:45:47 | afbd3d09 | view             | http://...?id=news341
 2016-12-29 21:45:47 | df6eb25d | view             | http://...?id=news731
 2016-12-29 21:45:56 | df6eb25d | read-news-start  | http://...?id=news731
 2016-12-29 21:46:05 | df6eb25d | read-recommend   | http://...?id=news731
 2016-12-29 21:46:13 | df6eb25d | read-ranking     | http://...?id=news731
 2016-12-29 21:46:22 | df6eb25d | read-newly-news  | http://...?id=news731
 2016-12-29 21:46:25 | df6eb25d | read-finish      | http://...?id=news731
...
```

下記コード例に、冒頭のデータを用いて読了率を集計するSQLを示します。

コード6.2.8.1：読了率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    url
  , action
  , COUNT(1) AS count
  , 100.0
    * COUNT(1)
    / SUM(CASE WHEN action = 'view' THEN COUNT(1) ELSE 0 END)
        OVER(PARTITION BY url)
    AS action_per_view
FROM read_log
GROUP BY
  url, action
ORDER BY
  url, count DESC
;
```

```
                url                |  action   | count | action_per_view
-----------------------------------+-----------+-------+-----------------
 http://.../article?id=news731     | view      |  1655 |          100.00
 http://.../article?id=news731     | read-20%  |  1012 |           61.14
 http://.../article?id=news731     | read-40%  |   903 |           54.56
 http://.../article?id=news731     | read-60%  |   774 |           46.76
 http://.../article?id=news731     | read-80%  |   620 |           37.46
 http://.../article?id=news731     | read-100% |   517 |           31.23
 http://.../article?id=trend925    | view      |  1709 |          100.00
 http://.../article?id=trend925    | read-20%  |   979 |           57.28
 http://.../article?id=trend925    | read-40%  |   877 |           51.31
 http://.../article?id=trend925    | read-60%  |   745 |           43.59
 http://.../article?id=trend925    | read-80%  |   583 |           34.11
 http://.../article?id=trend925    | read-100% |   486 |           28.43
```

ワンポイント

ECサイトでは商品詳細下部に「おすすめ商品」を表示するモジュール、ニュースサイトでは記事の最後に関連ニュースを表示するモジュールなどを設置しているケースがあります。しかし、これらのモジュールは最後まで閲覧される前提で設置されているため、そもそもの該当ページで読了率が低ければ、モジュールをいくら改修しても効果は限定的です。したがって、ページを最後まで閲覧しているユーザーの割合を必ず確認し、モジュール改修で十分な効果が得られる状態か判断しましょう。

6-2-9 ユーザー行動の全体像を可視化する

一般的なアクセス解析ツールでも、本節で紹介した各種レポート機能が提供されているため、実際に目にしたことがある方も多いはずです。ただ、これらのレポートを受動的にぼんやりと閲覧している状態に留まり、仮説を立てて施策を立案するに至っていないケースが多々見受けられます。

本節で紹介している各レポートをそれぞれ参考にして、現状を分析し施策を立案することもデータ分析の1つの手段ですが、各項はあくまでも断片的な情報に過ぎず、ここから得られる知見や仮説は局所的なものです。

つまり、各項の集計・レポートはもちろん、アクセス解析ツールのレポートは、「点」の情報です。点の情報であるがゆえに全体像を捉えきれず、十分に活かすことができません。そのため点の情報から施策を検討してリリースしたとしても、改善幅はあくまでも局所的でかつ効果も限定的なものに留まるかもしれません。点と点を結び「線」として捉え、線と線を結び「面」として捉えることができてはじめて、全体像を把握でき最適化のために何をすべきかが見えてくるはずです。

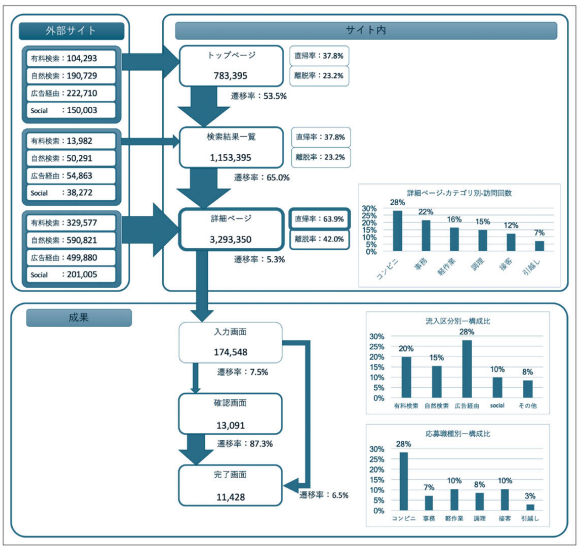
さらに補足すると、各ページの離脱率を調べることは「点」の状態です。続いて、フォームでの入力、確認、完了に至るまでの一連の流れで、各ページの離脱率をまとめたり、入力開始から完了までの流れで離脱率をまとめることで、「線」の状態になります。例えば、エントリーフォームを改善するチームと検索機能を改善するチームなど、各機能をそれぞれ別のチームが担当しているケースでは、この線の情報が重要になります。

全体を監督する方に、作成したレポートをもとに説明もしくは提案した結果、芳しくない反応が返ってきた経験があるかもしれません。サービス全体を監督する立場では、この線と線の情報を結び付けて「面」として捉え、どこで施策を実行し、どこに重点的に人員を配置すべきか考える必要があります。そのため、提出したレポートが「点」や「線」の状態に過ぎず、局所的な最適化には適したレポートであっても、全体の最適化を模索する観点では、必要とする情報が不足していたのかもしれません。

俯瞰図でサービスの全体像を把握する

それでは、どのようなレポートが、「面」として事象を捉えるのに最適なのでしょうか。その回答の1つとして、サービス全体を俯瞰的に捉える図（俯瞰図）を作成することをおすすめします。下図に俯瞰図の一例を示します。

図 6.2.9.1：ユーザーの回遊における俯瞰図



上図に示した俯瞰図では、本節で紹介したSQLやアクセス解析ツールから得られる情報を繋ぎ合わせて、各種の情報を1枚の図として表現しています。表計算ソフトなどで自動的に作成できるレポートではなく、線の太さや矢印の大きさなどで、ユーザーの回遊の大小を表現したり、枠を用いて内訳を記載するなど、手動で作成しています。

俯瞰図を作成することで、サイトや組織の全体像が把握でき、どこに問題点があるのか、何を改善すべきなのか、チームの枠組みを越えて議論することが可能になります。「木を見て森を見ず」とことわざにありますが、この森を最初に捉えずに、各担当者やチームで、それぞれの木を見ている状態が数多くの現場で見受けられます。

例えば、下記にあげる思い込みや単純に担当だからと進めていたことが、本来は違うところに注力すべきだったと後悔した経験はないでしょうか。

- ユーザーはトップページからアクセスするものだと思い込み、トップページからの検索画面と検索結果一覧画面の最適化に注力していたが、実際のユーザーは詳細画面から閲覧を開始しているため、最適化した機能があまり使われていなかった。
- エントリーフォームの最適化に注力しても効果は限定的で、エントリーフォームの改善よりも、エントリーフォームに至るまでの過程に力を注ぐべきだった。

新しくサービスを担当する際には、必ずこの俯瞰図を作成して、注力すべき箇所とさほど注力しなくても構わない箇所を把握した上で、個別箇所における詳細な分析を開始すべきです。

サービスの形態にあわせて俯瞰図を作成する

サービス全体を捉える俯瞰図はWebサイトに限定したものではありません。複数のサービスを抱えるプラットフォームであれば、サービス間の利用状況、重複利用率、各サービスの売上やユーザー数など、プラットフォームにおける各サービスへの送客状況等を俯瞰的に捉えることで、全体的な戦略の立案に役立ちます。その上でチームでの個別最適を実行すべきです。

しかし、俯瞰図はサービスによって何を表現すべきか、何を盛り込むべきかは、サービスごとに異なります。誰が見ても悩むことなくサービス全体の状況を把握できるフォーマットが俯瞰図と考えてください。どのような形の俯瞰図を用意するかは、分析担当者の腕の見せ所です。

まとめ

データの抽出だけが分析担当者の仕事ではなく、その結果をもとにサービスや組織を改善していくまでが仕事です。そのため、サービスや組織の状態を俯瞰的に捉えて、メンバー全員がサービスの状況を正しく理解し、本来取り組むべき仕事への道筋を示すことが重要です。データの抽出はその中のプロセスの1つに過ぎません。

しかし、そのデータの抽出に時間を要したり、抽出方法やコツが分からず、日々苦労しているケースも多々あるはずです。データの抽出に悩むのは本来の仕事ではありません。知りたい情報を自由に取り出せる助けとして、本書に記載するSQLを活用してください。

6-3 エントリーフォームを最適化する

資料請求フォームや購入フォームなどをエントリーフォームと呼びます。エントリーフォームに遷移し、個人情報を入力している際に項目が多すぎるとストレスを感じ、入力エラーが頻発すると最後まで入力することを諦め、ユーザーが離脱してしまう可能性があります。

フォーム入力完了手前で離脱するユーザーの発生を抑え、成果を向上させるために工夫することをエントリーフォームの最適化（EFO：Entry Form Optimization）と呼びます。

エントリーフォームの最適化では、主に下記に例として示す対策を実施します。

- 必須入力と任意入力を明確にして、入力漏れを減らす
  - 必須入力の項目を目立たせる
- エラーの発生頻度を少なくする
  - 入力例を明記する
  - フォーマットに従っていない入力は即座に通知する
- 入力の手間を減らす
  - 入力項目を減らす
  - 入力補助機能を提供する（郵便番号から住所補完）
- 離脱する要素は取り除く
  - ヘッダーは入力フォーム専用の簡素化したものを準備する
  - 不要なナビゲーションは取り除く
- システム側で吸収する
  - 全角、半角の変換はシステム側で対応する

上記の対策を実施する前に、現状ではどの程度のエラーが発生しているのか、入力の開始から完了までにどの程度のユーザーが離脱しているのかを明確に把握した上で、対策実施後はどの程度改善されたのか、もしくは更なる改善の余地があるか把握するため、必ず数値化しておくことが重要です。

まったくエラーが発生していない項目に対して、いくら対策を施しても効果はありません。また、対策実施後の改善幅を計測することで、どんな対策を実施したらどの程度の改善が見込めるかなどのノウハウを蓄積でき、将来的に類似するEFOを実施する際は、効率の良い手段を選択することが可能となります。

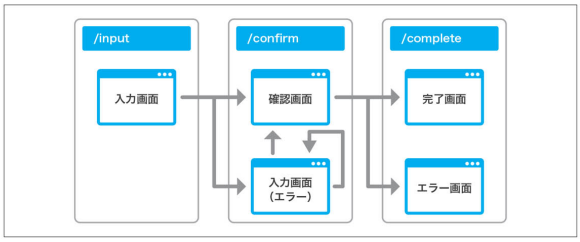
本節では、EFOの手段そのものを紹介するのではなく、EFOの際に用いる代表的な指標やレポートを紹介します。基本的には本章で紹介した手法を用いることで各種の集計は可能ですが、同じ呼称の指標でも、エントリーフォーム特有の落とし穴や、若干意味合いの異なるものも存在するため、整理して紹介します。

6-3-1 エラー率を集計する

SQL SUM(CASE～), AVG(CASE～)　　分析 エラー率

エントリーフォームの構造によっては、入力画面から確認画面へ遷移するボタンを押下した際に、URLそのものはconfirmに遷移しても、入力エラーが存在すると確認画面ではなく、入力画面を再表示してエラーを通知するケースがあります。このような場合は、URLで確認画面への遷移率を計測することはできません（下図参照）。

図6.3.1.1：同じURLでも表示している内容が異なる場合がある



確認画面へと遷移した際に、エラー表示を伴う入力画面の再表示なのか、確認画面の表示なのか、正しく集計できるようにログを出力しましょう。下記に示すログデータでは、エラー発生時はページ閲覧ログにステータスとしてerrorを出力します。本項ではこのログを用いてエラー率を集計します。

データ6.3.1.1：エントリーフォームにおける表示ログとエラーログ

```
         stamp         | session  | action |         url         | status
-----------------------+----------+--------+---------------------+--------
 2016-12-30 00:56:08 | 004a26c5 | view   | /regist/input       |
 2016-12-30 00:57:17 | 004a26c5 | view   | /regist/confirm     |
```

```
2016-12-30 00:58:57 | 004a26c5 | view   | /regist/complete  |
2016-12-30 00:56:08 | 5a53b5c2 | view   | /regist/input     |
2016-12-30 00:57:21 | 5a53b5c2 | view   | /regist/confirm   | error
2016-12-30 01:00:08 | 5a53b5c2 | view   | /regist/confirm   |
2016-12-30 01:01:33 | 5a53b5c2 | view   | /regist/complete  |
```

/confirmページで、エラー表示を伴う再入力画面か確認画面の表示を集計するクエリを、下記のコード例に示します。分母としてURLが「/regist/confirm」であるページを指定し、ステータスのエラーをCASE式でフラグに変換して、SUM関数やAVG関数を用いてエラー数やエラー率を計算しています。

コード6.3.1.1：確認画面におけるエラー率を集計するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
SSELECT
  COUNT(*) AS confirm_count
, SUM(CASE WHEN status = 'error' THEN 1 ELSE 0 END) AS error_count
, AVG(CASE WHEN status = 'error' THEN 1.0 ELSE 0.0 END) AS error_rate
, SUM(CASE WHEN status = 'error' THEN 1.0 ELSE 0.0 END) / COUNT(DISTINCT session)
  AS error_per_user
FROM form_log
WHERE
 -- 確認画面ページの判定
 path = '/regist/confirm'
;
```

```
 confirm_count | error_count | error_rate | error_per_user
---------------+-------------+------------+----------------
          1294 |         331 |      0.255 |          0.334
```

ワンポイント

エントリーフォームで1訪問当たりのエラー発生回数が多いケースでは、エラーの通知方法がユーザーにとって分かりづらく、繰り返しエラーになっている可能性も考えられます。エラーの通知方法を見直してみましょう。

関連項目 | 3-4-3 条件のフラグを0と1で表現する

6-3-2 入力〜確認〜完了までの遷移率を集計する

SQL LAG関数、MIN関数、COUNTウィンドウ関数、FIRST_VALUE関数

分析 確定率、離脱率

エントリーフォームを最適化する際、まずは「入力〜確認〜完了」までのフォールアウトレポートをざっと確認してみましょう。前項「6-3-1 エラー率を集計する」(P.305) で説明した通り、URLでは確認画面の表示なのか、エラー表示を伴う再入力画面の表示か判別できないケースがあるため、別途ステータスのフィールドを用意し、エラーステータスを判別できるログを取得します。フォールアウトレポートを集計するには、「6-2-6 フォールアウトレポートを用いてユーザーの回遊を可視化する」(P.285) を参照してください。

表6.3.2.1：エントリーフォームにおけるフォールアウトレポート

ステップ	訪問回数	先頭からの遷移率		直前からの遷移率	
input	7,292	100.0%		100.0%	
confirm	1,234	16.9%		※1	16.9%
complete	1,033	※2	14.2%	83.7%	

エントリーフォームの最適化では、入力開始から確認画面へ遷移した割合(上表※1)を「確定率」と呼ぶことがあります。同様に、完了画面まで遷移した割合(上表※2)を「CVR」、100%から完了画面まで遷移した割合 (CVR) を引いた値を「離脱率」と呼ぶことがあります。

前項「6-3-1 エラー率を集計する」で使用したログを用いて、フォールアウトレポートを作成するためのクエリを、下記のコード例に示します。

コード6.3.2.1：エントリーフォームにおけるフォールアウトレポート

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_fallout_step AS (
 -- /registフォームにおけるフォールアウトのステップとパスのマスタ
          SELECT 1 AS step, '/regist/input'    AS path
 UNION ALL SELECT 2 AS step, '/regist/confirm'  AS path
 UNION ALL SELECT 3 AS step, '/regist/complete' AS path
)
, form_log_with_fallout_step AS (
 SELECT
     l.session
   , m.step
   , m.path
```

```
    -- そのステップにおけるパスへの初回/最終アクセス時刻
    , MAX(l.stamp) AS max_stamp
    , MIN(l.stamp) AS min_stamp
  FROM
      mst_fallout_step AS m
    JOIN
      form_log AS l
      ON m.path = l.path
  -- 確認画面のステータスがエラーでないものに絞る
  WHERE status = ''
  -- セッションごとに、ステップ順とパスで集約
  GROUP BY l.session, m.step, m.path
)
, form_log_with_mod_fallout_step AS (
  SELECT
      session
    , step
    , path
    , max_stamp
      -- 直近のステップにおけるパスへの初回アクセス時刻
    , LAG(min_stamp)
        OVER(PARTITION BY session ORDER BY step)
      -- ■ SparkSQLの場合、LAG関数のフレーム指定が必要
      -- OVER(PARTIITION BY session ORDER BY step
      --   ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
      AS lag_min_stamp
      -- セッション中のステップ順の最小値
    , MIN(step) OVER(PARTITION BY session) AS min_step
      -- そのステップに至るまでの累計ステップ数
    , COUNT(1)
        OVER(PARTITION BY session ORDER BY step
          ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      AS cum_count
  FROM form_log_with_fallout_step
)
, fallout_log AS (
  -- フォールアウトレポートに使用するログのみを抽出する
  SELECT
      session
    , step
    , path
  FROM
    form_log_with_mod_fallout_step
  WHERE
        -- セッション中にステップ順が1のURLにアクセスしている
        min_step = 1
        -- 現在のステップ順が、そのステップに至るまでの累計ステップ数と同じ
    AND step = cum_count
        -- 直近のステップの初回アクセス時刻について
        -- NULL または 現在のステップの最終アクセス時刻より前である
```

```
   AND (lag_min_stamp IS NULL
        OR max_stamp >= lag_min_stamp)
)
SELECT
   step
 , path
 , COUNT(1) AS count
   -- ステップ順=1のURLからの遷移率
 , 100.0 * COUNT(1)
   / FIRST_VALUE(COUNT(1))
       OVER(ORDER BY step ASC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
   AS first_trans_rate
   -- 直近のステップからの遷移率
 , 100.0 * COUNT(1)
   / LAG(COUNT(1)) OVER(ORDER BY step ASC)
   -- ■ SparkSQLの場合､LAG関数のフレーム指定が必要
   -- / LAG(COUNT(1)) OVER(ORDER BY step ASC ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
   AS step_trans_rate
FROM
 fallout_log
GROUP BY
 step, path
ORDER BY
 step
;
```

```
 step |      path        | count | first_trans_rate | step_trans_rate
------+------------------+-------+------------------+-----------------
    1 | /regist/input    |  3068 |           100.00 |
    2 | /regist/confirm  |   991 |            32.30 |           32.30
    3 | /regist/complete |   782 |            25.48 |           78.91
```

ワンポイント

ページ閲覧のログを利用して、上記のレポートを作成すると、フォーム入力画面を表示した際に、入力の意思があったかどうかは判別できません。よって、入力の意思を確認したい場合は、最初の入力項目をクリックした際や最初の入力時に、JavaScriptを用いてログを送信することで、「入力画面表示～入力開始～確認画面表示～完了画面表示」のフォールアウトレポートを作成できます。

関連項目	3-3-2 グループの中での順序を扱う 3-4-5 擬似的なテーブルを作成する 6-2-6 フォールアウトレポートを用いてユーザーの回遊を可視化する

6-3-3 フォーム直帰率を集計する

SQL SUM(CASE～)、SIGN関数　　分析 フォーム直帰率

入力画面へ遷移した後、入力開始や確認画面、エラー画面を表示したログがない状態のレコードをカウントしたものを「フォーム直帰数」と呼びます。

フォーム直帰数が高い場合は、ユーザーが入力をすぐに諦めてしまうほど過剰な項目数であったり、表示レイアウトが煩雑であることなどが理由として考えられます。また、入力画面へ遷移するまでの過程でユーザーのモチベーション喚起が十分ではなく、唐突に入力フォームが表示される印象を与えているなどの問題も考えられます。

表6.3.3.1：フォーム直帰率

日付	フォーム訪問回数	フォーム直帰数	フォーム直帰率
2016/11/1	1,382	692	50.1%
2016/11/2	1,463	677	46.3%
2016/11/3	1,311	598	45.6%
2016/11/4	1,289	569	44.1%

フォーム直帰率は入力画面への訪問回数を母数として、フォーム直帰数の割合を集計することで計算できます。前述の「6-3-1 エラー率を集計する」(P.305) に記載したデータを用いて、フォーム直帰率を集計するSQLを、下記コード例に示します。

まず、セッションごとに入力画面（/regist/input）の訪問回数、確認画面（/regist/confirm）や完了画面（/regist/complete）の訪問回数を、SUM（CASE～）構文を用いてカウントして、SIGN関数を使用してフラグにしています。

さらに、入力画面を訪問しているセッションのみに絞り込み、同様にSUM（CASE～）構文を用いて直帰数を計算し、AVG（CASE～）構文を用いて直帰率を計算しています。

コード6.3.3.1：フォーム直帰率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
form_with_progress_flag AS (
  SELECT
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、substringで日付部分を抽出する
    substring(stamp, 1, 10) AS dt
  -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、substrが利用できる
  -- substr(stamp, 1, 10) AS dt

  , session
  -- 入力画面への訪問フラグを計算
  , SIGN(
```

```
      SUM(CASE WHEN path IN ('/regist/input') THEN 1 ELSE 0 END)
    ) AS has_input
  -- 入力確認画面または完了画面への訪問フラグを計算
  , SIGN(
      SUM(CASE WHEN path IN ('/regist/confirm', '/regist/complete') THEN 1 ELSE 0 END)
    ) AS has_progress
  FROM form_log
  GROUP BY
  -- ■ PostgreSQL, Redshift, BigQueryの場合、
  --   SELECT句で定義した別名をGROUP BYに指定できる
  dt, session
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、
  --   SELECT句で別名を指定する前の式をGROUP BYに指定できる
  -- substring(stamp, 1, 10), session
)
SELECT
    dt
  , COUNT(1) AS input_count
  , SUM(CASE WHEN has_progress = 0 THEN 1 ELSE 0 END) AS bounce_count
  , 100.0 * AVG(CASE WHEN has_progress = 0 THEN 1 ELSE 0 END) AS bounce_rate
FROM
  form_with_progress_flag
WHERE
  -- 入力画面に訪問しているセッションに絞り込む
  has_input = 1
GROUP BY
  dt
;
```

```
     dt     | input_count | bounce_count | bounce_rate
------------+-------------+--------------+-------------
 2016-12-30 |        3068 |         2077 |       67.69
```

ワンポイント

ページ閲覧のログを利用して、上記のレポートを作成すると、入力開始後の離脱なのか、入力は一切行わずに直帰したか判別できません。前述の「6-3-2 入力〜確認〜完了までの遷移率を集計する」（P.307）と同様に、最初の入力項目をクリックした際や最初の入力時に、JavaScriptを用いてログを送信することで、入力すら開始していない、より厳しい条件のフォーム直帰率を集計可能です。

関連項目 | 6-2-2 離脱率と直帰率を計算する

6-3-4 エラーが発生している項目と内容を集計する

SQL SUMウィンドウ関数

エントリーフォームでのエラー率の集計は、前述の「6.3.1 エラー率を集計する」(P.305) で紹介しましたが、発生したエラーがどのような原因で発生したのかまでは分かりません。原因を把握することで、場当たり的な対応ではなく、根拠を持って原因に対する適切な対応が可能になります。

エントリーフォームでエラーが発生した場合では、下記データに示す通り、適切なエラーログを出力すると良いでしょう。

データ6.3.4.1：エントリーフォームエラーログ(form_error_log)

```
        stamp        | session  |  form   | field |  error_type  |    value
---------------------+----------+---------+-------+--------------+-------------
 2016-12-30 00:56:08 | 004dc3ef | regist  | email | require      |
 2016-12-30 00:56:08 | 004dc3ef | regist  | kana  | require      |
 2016-12-30 00:57:21 | 004dc3ef | regist  | zip   | format_error | 101-
 2016-12-30 00:56:08 | 00700be4 | cart    | email | format_error | xxx---.co.jp
 2016-12-30 00:56:09 | 01061716 | regist  | email | format_error | xxx@---cojp
 2016-12-30 00:56:42 | 01061716 | regist  | kana  | not_kana     | 山田 太郎
 2016-12-30 00:56:09 | 02596e8a | regist  | kana  | require      |
 2016-12-30 00:56:09 | 035a1ebb | cart    | tel   | format_error | 03-99999999
```

ちなみに、上記データでは、エントリーフォーム入力時に、何を入力した結果としてどんなエラーが発生したかをレポートするため､入力された文字列をそのままログに出力しています｡エントリーフォームには個人情報が含まれている可能性があるため、個人情報と思われる入力項目では、入力された文字列を格納しないこと、ログ転送時にはセキュアな接続で転送すること、ログにアクセスできるユーザーを制限するなど、諸々のセキュリティ対策が必要です。所属する組織の規定に基いて対応しましょう。

ログの出力結果を集計し、どの項目に対してどれぐらいのエラーが発生しているのかをまとめたレポート例を、下表に示します。share列はフォーム単位でその割合を集計しています。

表6.3.4.1：各フォームのエラー発生箇所と原因

form	name	error_type	count	share
cart	kana	require	30	60%
cart	kana	not_kana	10	20%
cart	email	require	10	20%
contact	email	require	3	60%
contact	tel	length_less	2	40%

前述の「各フォームのエラー発生箇所と原因」をまとめたレポートを作成するためのクエリを、下記コード例に示します。フォーム種類や入力項目、エラー種類で集約を行ない、エラー数や全体に占める割合を計算します。また、全体に占める割合を求めるために、SUMウィンドウ関数を用いて全体のエラー数の母数を取得しています。

コード6.3.4.1：各フォームのエラー発生箇所と原因を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    form
  , field
  , error_type
  , COUNT(1) AS count
  , 100.0 * COUNT(1) / SUM(COUNT(1)) OVER(PARTITION BY form) AS share
FROM
  form_error_log
GROUP BY
  form, field, error_type
ORDER BY
  form, count DESC
;
```

```
  form   |     field     |    error_type     | count | share
---------+---------------+-------------------+-------+-------
 cart    | email         | format_error      |   106 | 14.68
 cart    | name          | require           |    75 | 10.38
 cart    | zip           | format_error      |    73 | 10.11
 cart    | tel           | format_error      |    72 |  9.97
…
 cart    | kana          | require           |    67 |  9.27
 cart    | email         | require           |    62 |  8.58
 cart    | email         | invalid_character |    55 |  7.61
 contact | email         | format_error      |   109 | 16.07
 contact | kana          | require           |    77 | 11.35
…
```

ワンポイント

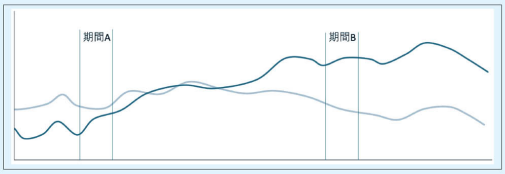
エラー発生率が高い項目から、本節「6-3 エントリーフォームを最適化する」(P.304)で説明した対策を実施していきましょう。

ここで落とし穴となるのが郵便番号です。郵便番号から住所を自動補完したり、登録済み郵便番号マスターと照合して存在しない場合にエラーを表示する仕組みを提供しているケースでは、郵便番号改定時にその都度対応する必要があります。また、市区町村をベースとした郵便番号以外に、配達物数が多い大口事業所を表す個別郵便番号が存在します（大口事業所個別番号）。必ず大口事業所個別番号も忘れずに管理しましょう。

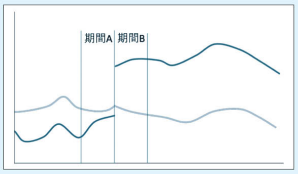
印刷したレポートを折ってみる！？

長期間の指標推移を折れ線グラフで表現したレポートを作成した際、ある期間とある期間がどの程度の成長･衰退なのか把握するには、その期間が離れていれば離れているほど、簡単には判別できなくなります。

例を挙げると、下図の期間Aと期間Bで、2つの指標がどれぐらい成長しているのか確認する際、青色の指標は成長していることが明白ですが、灰色の指標は成長・衰退の判断が難しいところです。



成長・衰退を把握したい場合、レポートを紙で受け取っていたら、期間Aの終了日と期間Bの開始日が重なるように折ってみましょう。紙を折ることで下図となり、その結果、青色の指標は明らかな成長が確認でき、灰色の指標は成長も衰退もしていないことが分かります。



紙だからこそ可能な簡単な比較方法です。レポート説明の場で参加者に折ってもらったところ、分かりやすいと好評でした。是非お試しください。

Chapter 7

データ活用の精度を高めるための分析術

自らデータを加工する手段を習得することで
データの不足やノイズなどの困難な状況を解消し、
目的の分析を正しく実行できる
スキルを身に付けましょう。

7-1 データを組み合わせて、新たな切り口を作る

データ分析では、「このデータがあればこんな分析も可能なのに」と思う場面に遭遇します。既存のデータを加工して意味を与えるのではなく、マスタデータが存在しないケースで、データがないから分析できないと諦めるのは、そこに改善の糸口が存在するのであれば残念なことです。

予め分析内容を網羅するデータ設計を行ったとしても、分析スキルが上がり対応できる幅が拡がると、データが不足する状況は発生します。また、十分に検討せずにデータ設計を行い、業務データの活用を優先して分析基盤を用意したのであれば、サービス運営を目的としたデータ設計ゆえに、より早い段階でこの壁にぶつかります。こうした状況でも、新たなデータの準備やデータ加工で、当初想定していなかったデータ分析も可能になります。

昨今では、国や大学、研究機関など各種団体が公開する、オープンデータと呼ばれる二次利用が可能なデータが存在します。自ら取得するには困難なデータを取り込み、担当するサービスのデータと組み合わせることで、新たな発見が得られるかもしれません。

本節では、外部データの取り込みやデータの加工で、分析の幅を広げ、データの精度を高める方法を紹介します。

7-1-1 IPアドレスから国・地域を補完する

SQL inet型　　分析 国、地域判定

ユーザーのログに付与されたIPアドレスを用いて、国や地域情報を補完できれば、地域別のセグメンテーションやタイムゾーンに応じた分析が可能となります。本項では、無料で利用できるジオロケーションデータベースを用いて、IPアドレスから国や地域の情報を補完する手法を紹介します。なお、コード例は、PostgreSQLを例に記載しています。

下記データ例に、本項で使用するサンプルデータを示します。

データ7.1.1.1：IPアドレスを含むアクションログ(action_log_with_ip)テーブル

```
 session | user_id | action |       ip       |        stamp
---------+---------+--------+----------------+---------------------
 0CVKaz  | U001    | view   | 216.58.220.238 | 2016-11-03 18:00:00
 1QceiB  | U002    | view   | 98.139.183.24  | 2016-11-03 19:00:00
 1hI43A  | U003    | view   | 210.154.149.63 | 2016-11-03 20:00:00
```

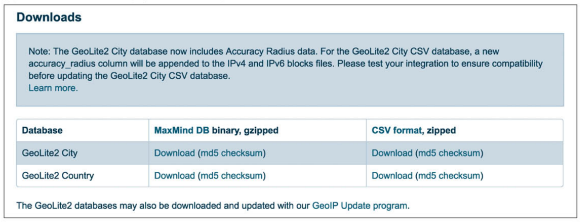
GeoLite2のデータロード

「GeoLite2」は、MaxMind社（https://www.maxmind.com）が提供する無料のジオロケーションデータベースです。GeoLite2のデータベースは、各種クライアントから利用できるバイナリファイルと、データベースにロードできるCSVファイル形式でダウンロードでき、月1回の頻度で更新されています。

本項では、提供されるCSVファイルをダウンロードし、PostgreSQLのテーブルにデータをロードして利用する方法を紹介します。

まずは、公式ダウンロードページ※1にアクセスして、CSVフォーマット形式のファイルをダウンロードします。国情報だけで十分な場合は「GeoLite2 Country」、より詳細な都市レベルの情報が必要な場合は「GeoLite2 City」を利用します。

図7.1.1.1:GeoLite2のダウンロードページ



Downloads

Note: The GeoLite2 City database now includes Accuracy Radius data. For the GeoLite2 City CSV database, a new accuracy_radius column will be appended to the IPv4 and IPv6 blocks files. Please test your integration to ensure compatibility before updating the GeoLite2 City CSV database.
Learn more.

Database	MaxMind DB binary, gzipped	CSV format, zipped
GeoLite2 City	Download (md5 checksum)	Download (md5 checksum)
GeoLite2 Country	Download (md5 checksum)	Download (md5 checksum)

The GeoLite2 databases may also be downloaded and updated with our GeoIP Update program.

ダウンロードしたZIPファイルを展開し、CSVファイルをPostgreSQLサーバ上の適当な場所に配置します。GeoLite2はIPv4とIPv6に対応していますが、今回はIPv4版の「GeoLite2-City-Blocks-IPv4.csv」を使用します。また、地域情報のマスタデータとして、英語版の「GeoLite2-City-Locations-en.csv」を使います。その他の言語が必要な場合は、該当言語に対応するファイルを使用してください。

続いて、CSVファイルのフォーマットに対応したテーブルを作成してデータをロードします。CSVファイルのフォーマットは、公式ドキュメント※2を参照してください。

下記のコード例に、上記2つのCSVファイルに対応するテーブル定義と、データをロードするクエリを示します。ネットワーク範囲の定義は、PostgreSQLでIPアドレスを扱うためのinet型で定義しています。

※1 http://dev.maxmind.com/geoip/geoip2/geolite2/
※2 http://dev.maxmind.com/geoip/geoip2/geoip2-city-country-csv-databases/

コード7.1.1.1：GeoLite2のCSVデータをロードするクエリ

PostgreSQL

```
DROP TABLE IF EXISTS mst_city_ip;
CREATE TABLE mst_city_ip(
    network                         inet PRIMARY KEY
  , geoname_id                      integer
  , registered_country_geoname_id   integer
  , represented_country_geoname_id  integer
  , is_anonymous_proxy              boolean
  , is_satellite_provider           boolean
  , postal_code                     varchar(255)
  , latitude                        numeric
  , longitude                       numeric
  , accuracy_radius                 integer
);

DROP TABLE IF EXISTS mst_locations;
CREATE TABLE mst_locations(
    geoname_id             integer PRIMARY KEY
  , locale_code            varchar(255)
  , continent_code         varchar(10)
  , continent_name         varchar(255)
  , country_iso_code       varchar(10)
  , country_name           varchar(255)
  , subdivision_1_iso_code varchar(10)
  , subdivision_1_name     varchar(255)
  , subdivision_2_iso_code varchar(10)
  , subdivision_2_name     varchar(255)
  , city_name              varchar(255)
  , metro_code             integer
  , time_zone              varchar(255)
);

COPY mst_city_ip   FROM '/path/to/GeoLite2-City-Blocks-IPv4.csv'  WITH CSV HEADER;
COPY mst_locations FROM '/path/to/GeoLite2-City-Locations-en.csv' WITH CSV HEADER;
```

上記コード例でロードしたCSVファイルのデータサンプルを、下記に示します（データ7.1.1.2、データ7.1.1.3）。

データ7.1.1.2：IPアドレスと地域情報のマスタテーブル(mst_city_ip)のサンプルデータ

```
-[ RECORD 1 ]------------------+----------
network                        | 1.0.0.0/24
geoname_id                     | 2077456
registered_country_geoname_id  | 2077456
represented_country_geoname_id |
is_anonymous_proxy             | f
is_satellite_provider          | f
postal_code                    |
```

```
latitude                        | -33.4940
longitude                       | 143.2104
accuracy_radius                 | 1000
-[ RECORD 2 ]-------------------+----------
network                         | 1.0.1.0/24
geoname_id                      | 1810821
registered_country_geoname_id   | 1814991
represented_country_geoname_id  |
is_anonymous_proxy              | f
is_satellite_provider           | f
postal_code                     |
latitude                        | 26.0614
longitude                       | 119.3061
accuracy_radius                 | 50
```

データ7.1.1.3：地域情報のマスタテーブル(mst_locations)のサンプルデータ

```
-[ RECORD 1 ]----------+-------------
geoname_id             | 16617
locale_code            | en
continent_code         | EU
continent_name         | Europe
country_iso_code       | CY
country_name           | Cyprus
subdivision_1_iso_code | 05
subdivision_1_name     | Pafos
subdivision_2_iso_code |
subdivision_2_name     |
city_name              | Choletria
metro_code             |
time_zone              | Asia/Nicosia
-[ RECORD 2 ]----------+-------------
geoname_id             | 49518
locale_code            | en
continent_code         | AF
continent_name         | Africa
country_iso_code       | RW
country_name           | Rwanda
subdivision_1_iso_code |
subdivision_1_name     |
subdivision_2_iso_code |
subdivision_2_name     |
city_name              |
metro_code             |
time_zone              | Africa/Kigali
```

IPアドレスから国や地域情報を補完する

GeoLite2のデータロードが完了したら、アクションログのIPアドレスと結合し、国や地域情報を補完してみましょう。下記コード例に、IPアドレスから地域や国、都市、タイムゾーンなどの地域情報を補完するクエリを示します。

コード7.1.1.2：アクションログのIPアドレスから地域情報を取得するクエリ

PostgreSQL

```
SELECT
    a.ip
  , l.continent_name
  , l.country_name
  , l.city_name
  , l.time_zone
FROM
    action_log AS a
  LEFT JOIN
    mst_city_ip AS i
    ON a.ip::inet << i.network
  LEFT JOIN
    mst_locations AS l
    ON i.geoname_id = l.geoname_id
;
```

```
       ip       | continent_name | country_name  |   city_name   |      time_zone
----------------+----------------+---------------+---------------+--------------------
 216.58.220.238 | North America  | United States | Mountain View | America/Los_Angeles
 98.139.183.24  | North America  | United States | Sunnyvale     | America/Los_Angeles
 210.154.149.63 | Asia           | Japan         |               | Asia/Tokyo
```

上記コード例の出力結果が示す通り、IPアドレスからアメリカや日本などの国情報、都市の名前などを補完できます。

ワンポイント

PostgreSQL以外のミドルウェアでIPアドレスを扱う手法は、「3-2-6 IPアドレスを扱う」(P.53) で紹介しています。その際は、ネットワーク範囲の最初と最後のIPアドレスを計算し、IPアドレスを比較可能な形式に変換するなどの前処理が必要です。CSVファイルのデータを前処理するツール「GeoIP2 CSV Format Converter」※3が提供されているので、必要に応じて活用してください。

関連項目	3-2-6 IPアドレスを扱う 7-2-4 特定のIPからのデータを除外する

※3 https://github.com/maxmind/geoip2-csv-converter/releases

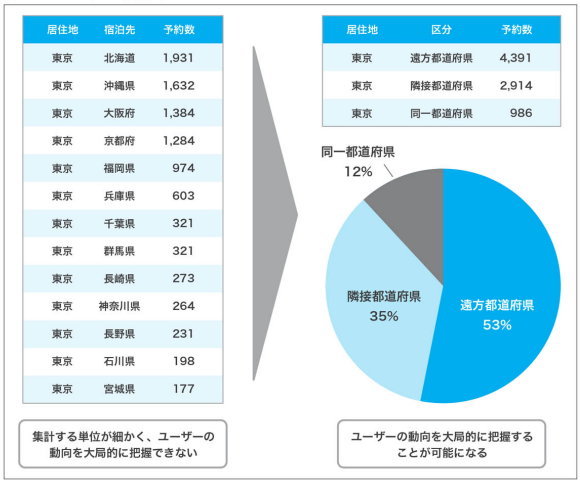
7-1-2 都道府県に隣接都道府県情報を付与する

SQL OUTER JOIN、CASE式

ユーザーマスタに都道府県が存在する場合、ユーザーがどの都道府県に対してアクション（閲覧やクチコミ、予約など）を起こしているのか集計することで、オススメする情報を変更して、CTRの改善を行ったり、次のアクションへ結び付けることを検討できます。

居住地（や出発地）と訪問先（や宿泊先）などを、「東京都→大阪府」などの都道府県レベルで集計すると、粒度が細かすぎて大局的にユーザー動向を捉えることが困難になります。そこで下図に示す通り、「同一都道府県」「隣接都道府県」「遠方都道府県」の3つの粒度で集計すると、ユーザーの特徴を捉えやすくなります。

図7.1.2.1：ユーザーの旅行先区分



隣接都道府県のマスターデータを準備する

前述の「同一都道府県」「隣接都道府県」「遠方都道府県」を判定するために、下記データ例に示す、マスタデータを準備します。

データ7.1.2.1：隣接都道府県(neighbor_pref)テーブル

```
 pref_id | pref_name | neighbor_pref_id | neighbor_pref_name
---------+-----------+------------------+--------------------
       1 | 北海道    |                2 | 青森県
       2 | 青森県    |                1 | 北海道
       2 | 青森県    |                3 | 岩手県
       2 | 青森県    |                5 | 秋田県
       3 | 岩手県    |                2 | 青森県
       3 | 岩手県    |                4 | 宮城県
       3 | 岩手県    |                5 | 秋田県
       4 | 宮城県    |                3 | 岩手県
       4 | 宮城県    |                5 | 秋田県
       4 | 宮城県    |                6 | 山形県
       4 | 宮城県    |                7 | 福島県
       5 | 秋田県    |                2 | 青森県
       5 | 秋田県    |                3 | 岩手県
       5 | 秋田県    |                4 | 宮城県
...
```

また、下記に示す、予約データと会員データを紐付けた状態のデータを使用します。

データ7.1.2.2：予約データと会員データを結合した予約(reservations)テーブル

```
 rsv_id | member_id | member_  | member_    | spot_id | spot_   | spot_     | stamp
        |           | pref_id  | pref_name  |         | pref_id | pref_name |
--------+-----------+----------+------------+---------+---------+-----------+--------
  27414 |     50063 |       21 |    岐阜県  |    4454 |      47 |    沖縄県 |2016-..
  27415 |     43065 |       19 |    山梨県  |    4899 |      27 |    大阪府 |2016-..
  27416 |     31038 |        6 |    山形県  |    7839 |      15 |    新潟県 |2016-..
  27417 |     53901 |       34 |    広島県  |    1972 |       4 |    宮城県 |2016-..
  27418 |     54998 |       12 |    千葉県  |    2227 |       3 |    岩手県 |2016-..
  27419 |     34078 |       47 |    沖縄県  |    5522 |      12 |    千葉県 |2016-..
  27420 |     53307 |       26 |    京都府  |    6559 |      12 |    千葉県 |2016-..
  27421 |     35423 |       24 |    三重県  |    5500 |      20 |    長野県 |2016-..
  27422 |     58570 |       12 |    千葉県  |    4485 |      24 |    三重県 |2016-..
  27423 |     27799 |       39 |    高知県  |    6052 |      11 |    埼玉県 |2016-..
```

マスターデータを用いてカテゴライズする

隣接都道府県マスタと予約データを組み合わせて、宿泊先の都道府県をカテゴライズするクエリを下記のコード例に示します。

隣接都道府県マスタは、都道府県ID（pref_id）と隣接都道府県ID（neighbor_pref_id）の組み合わせで一意になっているものとします。このとき、LEFT OUTER JOINで結合される都道府県マスタのレコードは0件または1件となるので、元の予約データのレコード数は変りません。続いて、SELECT句中で、宿泊先の都道府県IDが利用者の居住地と一致している場合は「同一都道府県」、隣接都道府県マスタと一致している場合は「隣接都道府県」、それ以外の場合は「遠方都道府県」と判定します。

コード7.1.2.1：隣接都道府県マスタを用いて宿泊先をカテゴライズするクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
rsv_with_category AS (
  SELECT
      r.rsv_id
    , r.member_id
    , r.member_pref_name
    , r.spot_pref_name
      -- 宿泊先の都道府県IDが、利用者の居住地か、隣接都道府県マスタに一致するかで場合分け
    , CASE r.spot_pref_id
        WHEN r.member_pref_id   THEN '同一都道府県'
        WHEN n.neighbor_pref_id THEN '隣接都道府県'
        ELSE '遠方都道府県'
      END AS category
  FROM
      reservations AS r
    LEFT OUTER JOIN
      neighbor_pref AS n
      -- 都道府県マスタは、pref_idとneighbor_pref_idで一意になっているものとする
      ON r.member_pref_id = n.pref_id
      AND r.spot_pref_id = n.neighbor_pref_id
)
SELECT *
FROM
  rsv_with_category
;
```

```
 rsv_id | member_id | member_pref_name | spot_pref_name |   category
--------+-----------+------------------+----------------+--------------
  29603 | 38612     | 東京都           | 宮城県         | 遠方都道府県
  31902 | 57811     | 佐賀県           | 福岡県         | 隣接都道府県
  27588 | 58078     | 千葉県           | 宮城県         | 遠方都道府県
  32960 | 48754     | 兵庫県           | 沖縄県         | 遠方都道府県
  27627 | 53905     | 滋賀県           | 京都府         | 隣接都道府県
```

```
  28935 | 34390     | 北海道             | 岡山県           | 遠方都道府県
  27640 | 45805     | 東京都             | 東京都           | 同一都道府県
...
```

次のステップ 続いて、利用者の居住地とカテゴライズされた宿泊先情報で集約し、居住地ごとの宿泊先カテゴリをカウントします。

コード7.1.2.2：居住地ごとの宿泊先カテゴリをカウントするクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
rsv_with_category AS (
  -- コード7.1.2.1参照
)
SELECT
    member_pref_name
  , category
  , COUNT(1) AS count
FROM
  rsv_with_category
GROUP BY
  member_pref_name, category
ORDER BY
  member_pref_name, count DESC
;
```

```
 member_pref_name |   category   | count
------------------+--------------+-------
 三重県           | 遠方都道府県 |   146
 三重県           | 隣接都道府県 |    23
 三重県           | 同一都道府県 |     2
 京都府           | 遠方都道府県 |   304
 京都府           | 隣接都道府県 |    49
 京都府           | 同一都道府県 |    14
 佐賀県           | 遠方都道府県 |   148
 佐賀県           | 隣接都道府県 |     5
 佐賀県           | 同一都道府県 |     2
 兵庫県           | 遠方都道府県 |   293
 兵庫県           | 隣接都道府県 |    37
 兵庫県           | 同一都道府県 |    16
...
```

ワンポイント

ユーザーの行動範囲は時期で異なる場合があります。夏休みや冬休み、大型連休などの特定期間とそれ以外でユーザーの行動を比較すると、異なる行動パターンが得られるかもしれません。

7-1-3 土日・祝日を判断する

分析 土日・祝日

土日や祝日に訪問回数やCVが増えるサービス、それとは逆に平日に増加傾向にあるサービスなど、サービスの提供内容によって特徴に違いがあります。月次目標を設定するシーンなどで、該当年や該当月に、土日や祝日が何日あるかを考慮することで、より正確な目標を設定できます。

データの増減が平日や土日祝日による変動なのか、カレンダーを確認しながら判断せず、データで判断可能な環境を整えましょう。

サンプルデータ

本項では、下記データに示すアクセスログを用いて、土日祝日のログかどうかを判断する方法を紹介します。

データ7.1.3.1：アクセスログ(access_log)テーブル

```
 session | user_id | action |        stamp
---------+---------+--------+---------------------
 98900e  | U001    | view   | 2016-11-01 18:00:00
 98900e  | U001    | view   | 2016-11-02 20:00:00
 98900e  | U001    | view   | 2016-11-03 22:00:00
 1cf768  | U002    | view   | 2016-11-04 23:00:00
 1cf768  | U002    | view   | 2016-11-05 00:30:00
 1cf768  | U002    | view   | 2016-11-06 02:30:00
 87b575  | U001    | view   | 2016-11-07 03:30:00
 87b575  | U001    | view   | 2016-11-08 04:00:00
 87b575  | U001    | view   | 2016-11-09 12:00:00
 eee2b2  | U002    | view   | 2016-11-10 13:00:00
 eee2b2  | U001    | view   | 2016-11-11 15:00:00
...
```

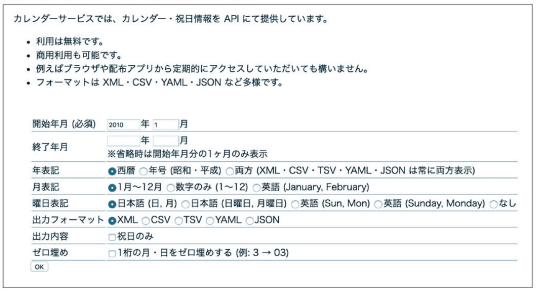
祝日情報を取り込む

日付データのみを使用するケースでは土日の判定は可能ですが、祝日の判定はできません。祝日に関するデータを外部から取得する必要があります。

数年先までのカレンダーをチェックして、データをまとめて登録しても構いませんが、ここでは、祝日情報をダウンロードできる「カレンダーサービス」※4から、カレンダー・祝日データをAPI経由で取得します。このWebサイトでは、指定期間の日本国内の祝日を、さまざまなフォーマットでダウンロード可能です。

※4 http://calendar-service.net/api.php

図 7.1.3.1：カレンダーサービスのカレンダー API



フォーマットを指定してダウンロードしてテーブルに保存します。今回はCSV形式を指定して、calendar.csvで保存したデータをmst_calendarテーブルにロードします。ここで取り扱うデータは日本語を含んでいるため、文字コードがデータベース側の設定と一致しているか事前に確認してください。

コード7.1.3.1：PostgreSQLでCSV形式のカレンダーデータをテーブルにロードするSQL

PostgreSQL

```
DROP TABLE IF EXISTS mst_calendar;
CREATE TABLE mst_calendar(
    year         integer
  , month        integer
  , day          integer
  , reign        varchar(10)
  , jp_year      integer
  , dow          varchar(10)
  , dow_num      integer
  , holiday_name varchar(255)
);

COPY mst_calendar   FROM '/path/to/calendar.csv'  WITH CSV HEADER;
```

データ7.1.3.2：インポートしたmst_calendarテーブル

```
 year | month | day | reign | jp_year | dow | dow_num | holiday_name
------+-------+-----+-------+---------+-----+---------+--------------
 2015 |     1 |   1 | 平成  |      27 | Thu |       4 | 元日
 2015 |     1 |   2 | 平成  |      27 | Fri |       5 |
 2015 |     1 |   3 | 平成  |      27 | Sat |       6 |
...
```

土日・祝日を判定する

下記コード例に、ロードしたカレンダーマスタをアクションログと組み合わせて、土日祝日を判定するクエリを示します。コード例では前処理を省略して、JOINの条件中にタイムスタンプから年月日を取り出して結合していますが、実際の運用ではクエリの効率を考慮して、ログ中に日付のみのカラムを用意し、マスタデータにも日付カラムを用意すべきでしょう。

コード7.1.3.2：土日や祝日の判定を行うクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    a.action
  , a.stamp
  , c.dow
  , c.holiday_name
    -- 休日判定
  , c.dow_num IN (0, 6)  -- 日曜、土曜日判定
    OR c.holiday_name IS NOT NULL -- 祝日判定
    AS is_day_off
FROM
    access_log AS a
  JOIN
    mst_calendar AS c
    -- アクションログのタイムスタンプから、年、月、日を取り出して結合している
    -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合は下記
    -- ■ BigQueryの場合、substringをsubstr, intをint64に修正
    ON  CAST(substring(a.stamp, 1, 4) AS int) = c.year
    AND CAST(substring(a.stamp, 6, 2) AS int) = c.month
    AND CAST(substring(a.stamp, 9, 2) AS int) = c.day
;
```

```
 action |     action_time     | dow | holiday_name | is_day_off
--------+---------------------+-----+--------------+------------
 view   | 2016-11-01 18:00:00 | Tue |              | f
 view   | 2016-11-02 20:00:00 | Wed |              | f
 view   | 2016-11-03 22:00:00 | Thu | 文化の日     | t
 view   | 2016-11-04 23:00:00 | Fri |              | f
 view   | 2016-11-05 00:30:00 | Sat |              | t
 view   | 2016-11-06 02:30:00 | Sun |              | t
 view   | 2016-11-07 03:30:00 | Mon |              | f
 view   | 2016-11-08 04:00:00 | Tue |              | f
 view   | 2016-11-09 12:00:00 | Wed |              | f
 view   | 2016-11-10 13:00:00 | Thu |              | f
 view   | 2016-11-11 15:00:00 | Fri |              | f
```

7-1-4 1日の集計範囲を変更する

SQL 日付/時刻関数　　分析 日跨ぎ考慮

日付別で集計する場合、日付と時刻に対して何も操作せずに集計すると、日付が切り替わる前後の一連の行動も、別の日のデータとして扱われます。サービスによっては午前零時前後がもっとも活性化する時間帯にも関わらず、日付が切り替わった瞬間に別の日のデータとして扱われると、ユーザー行動が捉えにくくなるケースがあります。

例えば、継続率を集計するケースでは、登録時刻が23:59で、日付が切り替わっても継続して利用した場合、たとえ一度だけの利用でも2日間の利用になり、継続ユーザーとして扱われてしまいます。

本項では、ユーザーの活動がもっとも少なくなる時刻が午前4時台であるサービスを想定して、午前4時から翌日の午前3時59分59秒までを一日の行動と捉え、データを加工する方法を紹介します。

サンプルデータ

本項では、下記に示すアクションログを用いて、午前0時～午前4時までのデータを、前日の日付で評価されるようにデータを加工します。

データ7.1.4.1：アクションログ(action_log)テーブル

```
 session | user_id | action |        stamp
---------+---------+--------+---------------------
 98900e  | U001    | view   | 2016-11-03 18:00:00
 98900e  | U001    | view   | 2016-11-03 20:00:00
 98900e  | U001    | view   | 2016-11-03 22:00:00
 1cf768  | U002    | view   | 2016-11-03 23:00:00
 1cf768  | U002    | view   | 2016-11-04 00:30:00
 1cf768  | U002    | view   | 2016-11-04 02:30:00
 87b575  | U001    | view   | 2016-11-04 03:30:00
 87b575  | U001    | view   | 2016-11-04 04:00:00
 87b575  | U001    | view   | 2016-11-04 12:00:00
 eee2b2  | U002    | view   | 2016-11-04 13:00:00
 eee2b2  | U001    | view   | 2016-11-04 15:00:00
```

1日の集計範囲を変更する

1日の集計範囲を午前4時スタートにずらす場合は、タイムスタンプから日付を抽出する際に、タイムスタンプの時刻から4時間を引きます。そのままの日付と4時間ずらした場合の日付を計算するクエリをコード例に示します。出力結果のmod_dateを確認すると、11月4日午前0時～3時59分のログは、11月3日として判定されています。

コード7.1.4.1：一日の集計範囲を午前4時から切り替えるクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_log_with_mod_stamp AS (
  SELECT *
    -- 4時間前の時刻を計算
    -- ■ PostgreSQLの場合、interval型のデータを用いて日付/時刻の四則演算ができる
    , CAST(stamp::timestamp - '4 hours'::interval AS text) AS mod_stamp
    -- ■ Redshiftの場合、dateadd関数を用いる
    -- , CAST(dateadd(hour, -4, stamp::timestamp) AS text) AS mod_stamp
    -- ■ BigQueryの場合、timestamp_sub関数を用いる
    -- , CAST(timestamp_sub(timestamp(stamp), interval 4 hour) AS string) AS mod_stamp
    -- ■ Hive, SparkSQLの場合、一度unixtimeに変換して秒単位の計算を行い、
    --   再度タイムスタンプに戻す
    -- , from_unixtime(unix_timestamp(stamp) - 4 * 60 * 60) AS mod_stamp
  FROM action_log
)
SELECT
    session
  , user_id
  , action
  , stamp
  -- 元のタイムスタンプの日付(raw_date)と、
  -- 1日の終わりを4時間後にずらしたときの日付(mod_date)を抽出
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合は下記
  , substring(stamp, 1, 10) AS raw_date
  , substring(mod_stamp, 1, 10) AS mod_date
  -- ■ BigQueryの場合、substringをsubstrに修正
  -- , substr(stamp, 1, 10) AS raw_date
  -- , substr(mod_stamp, 1, 10) AS mod_date
FROM action_log_with_mod_stamp;
```

```
 session | user_id | action |        stamp        |  raw_date  |  mod_date
---------+---------+--------+---------------------+------------+------------
 1cf768  | U002    | view   | 2016-11-03 23:00:00 | 2016-11-03 | 2016-11-03
 1cf768  | U002    | view   | 2016-11-04 00:30:00 | 2016-11-04 | 2016-11-03
 87b575  | U001    | view   | 2016-11-04 03:30:00 | 2016-11-04 | 2016-11-03
 87b575  | U001    | view   | 2016-11-04 04:00:00 | 2016-11-04 | 2016-11-04
 87b575  | U001    | view   | 2016-11-04 12:00:00 | 2016-11-04 | 2016-11-04
```

ワンポイント

1日の集計範囲を全レポートに対して適用する場合、既にビッグデータ分析基盤を運用しているのであれば、ログ収集とデータ集計に関連するバッチの処理時間を一日の集計範囲に合わせて変更する必要があります。また、日付通りに集計するシーンも考慮して、元データは残しつつ加工しましょう。

7-2 異常値を検出する

データ分析を進める上で元データが正しいことが大前提です。しかし、実際のデータには、値の欠損やノイズとなるデータが混入しているケースがあります。特にWebサイトのアクセスログやセンサーデータなどは、分析担当者が意図しない状態でデータが送られてくることが多々あります。

本節では、主にWebサイトのアクセスログを対象として、ノイズとなる異常値を含むデータを検出し、データ分析の前提となるデータをクレンジングする手法を紹介します。

サンプルデータ

本節で使用するアクションログのサンプルデータを下記に示します。

本データのカラムには、セッション、アクション種別、商品ID、URL、IPアドレス、ユーザーエージェント、アクション時刻が用意されています。

データ7.2.0.1：ノイズを含んだアクションログ(action_log_with_noise)のサンプルデータ

```
     stamp     | session |  action  | products |   url   |    ip    |  user_agent
---------------+---------+----------+----------+---------+----------+-------------
 2016-11-03... | 1b700   | view     |          | http... | 98.1...  | Mozilla/5...
 2016-11-03... | 1b700   | add_cart | D001     | http... | 98.1...  | Mozilla/5...
 2016-11-03... | 1b700   | purchase | D001     | http... | 98.1...  | Mozilla/5...
 2016-11-03... | 0fb22   | view     |          | http... | 210....  | Mozilla/5...
 2016-11-03... | 0fb22   | view     |          | http... | 210....  | Mozilla/5...
 2016-11-04... | fdb83   | view     |          | http... | 127....  | Mozilla/5...
 2016-11-04... | fe8df   | view     |          | http... | 192....  | Mozilla/5...
 2016-11-04... | fe8df   | view     |          | http... | 192....  | Mozilla/5...
 2016-11-04... | fe8df   | view     |          | http... | 192....  | Mozilla/5...
 2016-11-04... | 14bec   | view     |          | http... | 10.0...  | Mozilla/5...
 2016-11-04... | 14bec   | view     |          | http... | 10.0...  | Mozilla/5...
 2016-11-04... | 739ba   | view     |          | http... | 210....  | Mozilla/5...
 2016-11-04... | 739ba   | view     |          | http... | 210....  | Mozilla/5...
 2016-11-04... | 739ba   | add_cart |          | http... | 210....  | Mozilla/5...
 2016-11-04... | a2cfa   | view     |          | http... | 126....  | Mozilla/5...
 2016-11-04... | a2cfa   | view     |          | http... | 126....  | Mozilla/5...
...
```

7-2-1 データの分布を計算する

SQL PERCENT_RANK関数　　分析 上位n%、下位n%

ログデータから異常値を検出するもっとも基本的な方法は、データの分布を計算して、その分布から外れた値を異常値として判定する方法です。

例えば、Webサイトのアクセスログで、1セッションのページ閲覧数が極端に多いユーザーは、業者やクローラーの可能性があります。また、極端にアクセスが少ないページの場合は、実際には存在しないURLが誤って送信された可能性や、既に削除されたページのキャッシュからアクセスされた可能性も考えられます。

セッションあたりのページ閲覧数ランキングを割合で表示する

1セッションのページ閲覧数が極端に多いユーザーを確認するため、セッションあたりのページ閲覧数を計算して、閲覧数が多いセッションの上位n%を確認しましょう。

下記に示すコード例は、セッションごとにページ閲覧数を集計し、PERCENT_RANK関数を用いてページ閲覧数のランキングを割合と共に表示するクエリです。PERCENT_RANKの値は、(rank - 1) / (全行数 - 1) で計算される割合を示します。

コード7.2.1.1：セッションあたりのページ閲覧数ランキングを割合で表示するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
session_count AS (
  SELECT
      session
    , COUNT(1) AS count
  FROM
    action_log_with_noise
  GROUP BY
    session
)
SELECT
    session
  , count
  , RANK() OVER(ORDER BY count DESC) AS rank
  , PERCENT_RANK() OVER(ORDER BY count DESC) AS percent_rank
FROM
  session_count
;
```

```
 session | count | rank | percent_rank
---------+-------+------+--------------
 c33fb   |    17 |    1 |         0.00
 7af12   |     3 |    2 |         0.14
 1b700   |     3 |    2 |         0.14
 fe8df   |     3 |    2 |         0.14
 14bec   |     2 |    5 |         0.57
 0fb22   |     2 |    5 |         0.57
 fdb83   |     1 |    7 |         0.85
 694dd   |     1 |    7 |         0.85
```

上記コード例の出力結果から、例えば、percent_rankが0.05以下のものをフィルタすれば、セッションあたりのページ閲覧数が多い上位5%のセッションをフィルタできます。

URLのページビューのワーストランキングを割合で表示する

同様にURLのアクセス数ワーストランキングを割合で表示するクエリを、下記のコード例に示します。今回はアクセス数が少ないURLを上位に表示したいため、ウィンドウ関数内のORDER BY句でASCを指定しています。

コード7.2.1.2：URLのアクセス数ワーストランキングを割合で表示するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
url_count AS (
  SELECT
      url
    , COUNT(*) AS count
  FROM
    action_log_with_noise
  GROUP BY
     url
)
SELECT
    url
  , count
  , RANK() OVER(ORDER BY count ASC) AS rank
  , PERCENT_RANK() OVER(ORDER BY count ASC)
FROM
  url_count
;
```

```
                url                 | count | rank | percent_rank
------------------------------------+-------+------+--------------
 http://www.example.com/detail?id=4 |     1 |    1 |       0.0000
```

```
http://www.example.com/detail?id=1 |     9 |    2 |       0.3333
http://www.example.com/detail?id=2 |    10 |    3 |       0.6666
http://www.example.com/detail?id=3 |    12 |    4 |       1.0000
```

上記の結果から、アクセス数がある件数以下のものをフィルタしたり、percent_rankが指定値以下のものをフィルタすることで、アクセスが極端に少ないレコードを除外できます。

7-2-2 クローラーを除外する

SQL LIKE演算子　　分析 クローラーの除外

アクセスログには、ユーザー行動の他に検索エンジンや各種ツール（以降クローラー）がアクセスしたログが混入します。Webサイトによって頻度や件数は異なりますが、ユーザーのアクセス数と同等、もしくはそれ以上のクローラーからのアクセスがあるケースがあります。

分析を行う上で、クローラーのログはノイズとなるため、除去してから分析することをおすすめします。クローラーかユーザーか判別するには、ユーザーエージェントとその他の状況から判断します。

クローラーからのアクセスに対しては、ログを出力しない対応も可能です。しかし、新しいクローラーが開発されると、その対応は必ず事後となってしまい、ログデータにクローラーのアクセスログが混入せざるを得ないため、ログ側での除去が必要とあります。

また、クローラーのアクセスはクローラーの回遊状況分析で活用できる情報なので、データ容量に問題がない限り、ログ出力時に除外せずに保存し、必要に応じて集計時に除外する方が良いでしょう。

クローラーのアクセスを除外する方法

クローラーのアクセスを除外するにはいくつかの方法があります。本項では、下記の除去方法を解説します。

1. ルールベースで除外する
2. マスタデータを用いて除外する

ルールベースで除外する

クローラーのユーザーエージェントには特徴があり、下表に示す、「特定文字が含まれる場合」と「クローラー名が含まれる場合」があります。

表7.2.2.1：クローラーの特徴と判定するルール

特定文字列が含まれる	bot / crawler / spider etc...
クローラー名が含まれる	Googlebot / Baiduspider / Yeti / Yahoo / Tumblr etc..

表記載のルールをベースにクローラーの特徴を利用して、クローラーを除外する方法を下記のコード例に示します。ルールベースで除外することで、新たなクローラーが誕生した際も、記載ルールに従ったユーザーエージェントを使用している場合は、コードのメンテナンスなしで除外することが可能です。

コード7.2.2.1：ルールベースで除外するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
  *
FROM
  action_log_with_noise
WHERE
  NOT
  -- クローラーの判定条件を列挙
  (   user_agent LIKE '%bot%'
   OR user_agent LIKE '%crawler%'
   OR user_agent LIKE '%spider%'
   OR user_agent LIKE '%archiver%'
   --（以下略）
  )
;
```

```
     stamp     | session |  action  | products |   url   |   ip    |  user_agent
---------------+---------+----------+----------+---------+---------+-------------
 2016-11-03... | 0fb22   | view     |          | http... | 210.... | Mozilla/5...
 2016-11-03... | 0fb22   | view     |          | http... | 210.... | Mozilla/5...
 2016-11-04... | fdb83   | view     |          | http... | 127.... | Mozilla/5...
 2016-11-04... | fe8df   | view     |          | http... | 192.... | Mozilla/5...
 2016-11-04... | fe8df   | view     |          | http... | 192.... | Mozilla/5...
 2016-11-04... | fe8df   | view     |          | http... | 192.... | Mozilla/5...
 2016-11-04... | 14bec   | view     |          | http... | 10.0... | Mozilla/5...
 2016-11-04... | 14bec   | add_cart |          | http... | 10.0... | Mozilla/5...
 2016-11-04... | 694dd   | view     |          | http... | 172.... | ...
 2016-11-04... | 7af12   | view     |          | http... | 192.... | Mozilla/5...
 2016-11-04... | 7af12   | add_cart | D002     | http... | 192.... | Mozilla/5...
 2016-11-04... | 7af12   | purchase | D002     | http... | 192.... | Mozilla/5...
 2016-11-04... | c33fb   | view     |          | http... | 216.... | ...
```

マスタデータを用いて除外する

ユーザーエージェントのルールベースで除外する場合、SQLに直接ルールを記述すると、ルール追加時にSQLが長くなったり、同様の処理を行っている複数のSQLを更新する必要があるなど、その管理が煩雑になります。

別途、クローラーのマスタデータを準備することで、その煩わしさから開放されます。

コード7.2.2.2：マスタデータを用いて除外するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_bot_user_agent AS (
            SELECT '%bot%'      AS rule
  UNION ALL SELECT '%crawler%'  AS rule
  UNION ALL SELECT '%spider%'   AS rule
  UNION ALL SELECT '%archiver%' AS rule
)
, filtered_action_log AS (
  SELECT
      l.stamp, l.session, l.action, l.products
    , l.url , l.ip, l.user_agent
  -- UserAgentのルールのどれにも当てはまらないログのみを残す
  -- ■ PostgreSQL, Redshift, BigQueryの場合、
  --   WHERE句に相関サブクエリを用いることができる
   FROM
     action_log_with_noise AS l
   WHERE
     NOT EXISTS (
       SELECT 1
       FROM mst_bot_user_agent AS m
       WHERE
         l.user_agent LIKE m.rule
     )
  -- ■ 相関サブクエリが使用できない場合、
  --   CROSS JOINでマスタデータを結合し、HAVING句でルールに一致したレコードが0のログを残す
  --   ※ PostgreSQL, Hive, Redshift, BigQuery, SparkSQLで動作する
  -- FROM
  --     action_log_with_noise AS l
  --   CROSS JOIN
  --     mst_bot_user_agent AS m
  -- GROUP BY
  --     l.stamp, l.session, l.action, l.products
  --   , l.url , l.ip, l.user_agent
  -- HAVING SUM(CASE WHEN l.user_agent LIKE m.rule THEN 1 ELSE 0 END) = 0
)
SELECT
  *
FROM
  filtered_action_log
;
```

```
    stamp      | session |  action  | products |   url   |   ip    | user_agent
---------------+---------+----------+----------+---------+---------+------------
 2016-11-03... | 0fb22   | view     |          | http... | 210.... | Mozilla/5...
 2016-11-03... | 0fb22   | view     |          | http... | 210.... | Mozilla/5...
 2016-11-04... | fdb83   | view     |          | http... | 127.... | Mozilla/5...
 2016-11-04... | fe8df   | view     |          | http... | 192.... | Mozilla/5...
```

```
2016-11-04... | fe8df   | view     |          | http... | 192.... | Mozilla/5...
2016-11-04... | fe8df   | view     |          | http... | 192.... | Mozilla/5...
2016-11-04... | 14bec   | view     |          | http... | 10.0... | Mozilla/5...
2016-11-04... | 14bec   | add_cart |          | http... | 10.0... | Mozilla/5...
2016-11-04... | 694dd   | view     |          | http... | 172.... | ...
2016-11-04... | 7af12   | view     |          | http... | 192.... | Mozilla/5...
2016-11-04... | 7af12   | add_cart | D002     | http... | 192.... | Mozilla/5...
2016-11-04... | 7af12   | purchase | D002     | http... | 192.... | Mozilla/5...
2016-11-04... | c33fb   | view     |          | http... | 216.... | ...
```

クローラーを監視する

クローラーを除去する方法を決めたら、そのルールやマスタに追加すべきクローラーが新たに発生しているか確認するために、下記のコード例に示すクエリを定期的に実行しましょう。

マスターデータを用いて除外したログから、アクセスが多いユーザーエージェントをランキングで取得し、クローラーのマスターデータから漏れてしまったユーザーエージェントがないか確認します。

コード7.2.2.3：アクセスの多いユーザーエージェントを確認するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_bot_user_agent AS (
  -- コード7.2.2.2参照
)
, filtered_action_log AS (
  -- コード7.2.2.2参照
)
SELECT
    user_agent
  , COUNT(1) AS count
  , 100.0
    * SUM(COUNT(1)) OVER(ORDER BY COUNT(1) DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    / SUM(COUNT(1)) OVER() AS cumulative_ratio
FROM
  filtered_action_log
GROUP BY
  user_agent
ORDER BY
  count DESC
;
```

```
            user_agent            | count |   cumulative_ratio
----------------------------------+-------+---------------------
 Mozilla/5.0 (Windows NT 6.1; ... |     3 |            46.1538
 Mozilla/5.0 (Macintosh; Intel... |     3 |            46.1538
                                  |     2 |            92.3076
 Mozilla/5.0 (Windows NT 10.0;... |     2 |            92.3076
 Mozilla/5.0 (Macintosh; Intel... |     2 |            92.3076
 Mozilla/5.0 (Windows NT 6.1; ... |     1 |           100.0000
```

上記コード例のcumulative_ratioは構成比累計を集計しています。なお、構成比累計に関しては、「4-2-2 ABC分析で売れ筋を判別する」(P.115)で紹介しています。

まとめ

今まではアクセスがなかったクローラーがある日を境にアクセスが多発したり、新たなクローラーが追加されるなど、アクセス状況は常に変わり続けます。週次、少なくとも月次でクローラーのアクセスログを確認して、ログの精度を高めましょう。

関連項目	4-2-2 ABC分析で売れ筋を判別する

7-2-3 データの妥当性を確認する

SQL AVG(CASE〜)　　分析 データの検証

ユーザーのログデータを用いて分析する際も、そもそもログデータに欠損や誤りがあれば、正しい分析はできません。本項では、アクションログのデータ妥当性を手軽に確認するテクニックを紹介します。

本項では、下記に示す欠損値を含むアクションログをサンプルデータとして使います。

データ7.2.3.1：欠損値を含んだアクションログ(invalid_action_log)テーブル

```
 session | user_id |  action  | category | products  | amount |       stamp
---------+---------+----------+----------+-----------+--------+---------------------
 0CVKaz  | U001    | view     |          |           |        | 2016-11-03 18:00:00
 0CVKaz  | U001    | favorite | drama    | D001      |        | 2016-11-03 18:00:00
 0CVKaz  | U001    | add_cart | drama    | D002      |        | 2016-11-03 18:01:00
 0CVKaz  | U001    | add_cart | drama    |           |        | 2016-11-03 18:02:00
 0CVKaz  | U001    | purchase | drama    | D001,D002 |   2000 | 2016-11-03 18:10:00
 1QceiB  | U002    | purchase | drama    | D002      |   1000 | 2016-11-04 13:00:00
 1QceiB  | U001    | purchase | action   | A005,A006 |   1000 |
...
```

アクションログのテーブルは、アクションの種類に応じて、必須となるカラムが異なります。例えば、viewアクションには商品カテゴリや商品IDは不要だったり、商品の金額はpurchaseアクションのみに必要となるなどです。ログデータがこれらの要件を正しく満たしているかを一度に確認するクエリを、下記のコード例に示します。

コード例では、アクションごとにGROUP BYで集約して、セッションやユーザーIDなどの各カラムが満たすべき要件をCASE式で判定しています。そのカラムの要件を満たしていれば、CASE式の値は1となり、満たしていない場合は0となります。このCASE式の値をAVG関数で集約することで、ログデータ全体で、要件を満たしているデータの割合を計算できます。

コード7.2.3.1：ログデータの要件を満たしているか確認するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    action

    -- sessionは必ずNULLでない
  , AVG(CASE WHEN session IS NOT NULL THEN 1.0 ELSE 0.0 END) AS session

    -- user_idは必ずNULLでない
  , AVG(CASE WHEN user_id IS NOT NULL THEN 1.0 ELSE 0.0 END) AS user_id

    -- categoryは、action=viewの場合はNULL、それ以外はNULLでない
  , AVG(
      CASE action
        WHEN 'view' THEN
          CASE WHEN category IS NULL THEN 1.0 ELSE 0.0 END
        ELSE
          CASE WHEN category IS NOT NULL THEN 1.0 ELSE 0.0 END
      END
    ) AS category

    -- productsは、action=viewの場合はNULL、それ以外はNULLでない
  , AVG(
      CASE action
        WHEN 'view' THEN
          CASE WHEN products IS NULL THEN 1.0 ELSE 0.0 END
        ELSE
          CASE WHEN products IS NOT NULL THEN 1.0 ELSE 0.0 END
      END
    ) AS products

    -- amountは、action=purchaseの場合はNULLでなく、それ以外はNULL
  , AVG(
      CASE action
        WHEN 'purchase' THEN
          CASE WHEN amount IS NOT NULL THEN 1.0 ELSE 0.0 END
        ELSE
```

```
          CASE WHEN amount IS NULL THEN 1.0 ELSE 0.0 END
      END
    ) AS amount

    -- stampは必ずNULLでない
  , AVG(CASE WHEN stamp IS NOT NULL THEN 1.0 ELSE 0.0 END) AS stamp
FROM
  invalid_action_log
GROUP BY
  action
;
```

▼

```
  action  | session | user_id | category | products | amount |  stamp
----------+---------+---------+----------+----------+--------+---------
 favorite |    1.00 |    1.00 |     1.00 |     1.00 |   1.00 |   1.00
 add_cart |    1.00 |    1.00 |     1.00 |     0.92 |   1.00 |   1.00
 purchase |    1.00 |    1.00 |     1.00 |     1.00 |   1.00 |   0.75
 view     |    1.00 |    1.00 |     1.00 |     1.00 |   1.00 |   1.00
 review   |    1.00 |    1.00 |     1.00 |     1.00 |   1.00 |   1.00
```

上記コード例の出力結果が示す通り、すべてのデータが要件を満たしている場合には、カラムの値は1.00となります。一方、データの中に要件を満たしていないものがある場合は、カラムの値が1.00未満となります。この通り、AVG関数とCASE式を応用すれば、ログデータの妥当性を一度に網羅的に確認可能です。

7-2-4 特定IPアドレスからのアクセスを除外する

SQL inet型、文字列関数　　分析 IPアドレスの除外

Webサービスのアクセスログには、テストユーザーからのアクセスや社内からのアクセスなども混入しているケースがあります。また、特定のIPアドレスから本来のサービス利用者ではない不正アクセスをはじめ、クローラーなどのアクセスログが含まれていることもあります。

正規のサービス利用者以外のアクセスは、分析の際にはアクセスログから除外した方が良いでしょう。本項では、IPアドレスに基づいてテストユーザーや社内からのアクセスを判別するクエリを紹介します。

サンプルデータ

本項で利用するIPアドレスを含むアクションログテーブルのサンプルデータを、次に示します。

データ7.2.4.1：IPアドレスを含むアクションログ(action_log_with_ip)テーブル

```
 session  | user_id | action |       ip       |        stamp
----------+---------+--------+----------------+---------------------
 989004ea | U001    | view   | 216.58.220.238 | 2016-11-03 18:00:00
 47db0370 | U002    | view   | 98.139.183.24  | 2016-11-03 19:00:00
 1cf7678e | U003    | view   | 210.154.149.63 | 2016-11-03 20:00:00
 5eb2e107 | U004    | view   | 127.0.0.1      | 2016-11-03 21:00:00
 fe05e1d8 | U001    | view   | 216.58.220.238 | 2016-11-04 18:00:00
 87b5725f | U005    | view   | 10.0.0.3       | 2016-11-04 19:00:00
 5d5b0997 | U006    | view   | 172.16.0.5     | 2016-11-04 20:00:00
 111f2996 | U007    | view   | 192.168.0.23   | 2016-11-04 21:00:00
 3efe001c | U008    | view   | 192.0.0.10     | 2016-11-04 22:00:00
```

PostgreSQLで特定IPアドレスを除外する

アクセスログから特定IPアドレスのログを除外するために、まずは除外したいIPアドレスのネットワークを保持するマスタテーブルを作成します。下記のコード例に示す通り、予約IPアドレスを定義したマスタテーブル（mst_reserved_ip）を定義します。

コード7.2.4.1：予約IPアドレスを定義したマスタテーブル

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
mst_reserved_ip AS (
            SELECT '127.0.0.0/8'    AS network, 'localhost'       AS description
  UNION ALL SELECT '10.0.0.0/8'     AS network, 'Private network' AS description
  UNION ALL SELECT '172.16.0.0/12'  AS network, 'Private network' AS description
  UNION ALL SELECT '192.0.0.0/24'   AS network, 'Private network' AS description
  UNION ALL SELECT '192.168.0.0/16' AS network, 'Private network' AS description
)
SELECT *
FROM mst_reserved_ip
;
```

```
    network     |   description
----------------+-----------------
 127.0.0.0/8    | localhost
 10.0.0.0/8     | Private network
 172.16.0.0/12  | Private network
 192.0.0.0/24   | Private network
 192.168.0.0/16 | Private network
```

コード例に示すマスタテーブルは、除外したいIPアドレスのネットワークと、その意味を保持するテーブルです。例えば、「127.0.0.0/8」のネットワークは、127.0.0.0から127.255.255.255までに含まれるIPアドレスを表し、ローカルホストのIPとして用いられます。

続いて、下記コード例に示す通り、マスタテーブルとアクションログと結合してIPアドレスを判定します。マスタテーブルに登録されていないIPアドレスのログも残すため、アクションログに対してマスタテーブルをLEFT JOINで結合します。PostgreSQLの場合は、IPアドレスやネットワーク範囲にinet型を使用できます。なお、結合条件として、IPアドレスがマスタテーブルのネットワークに含まれているかを判定するため、<<演算子を用いています。

コード7.2.4.2：inet型を用いてIPアドレスの判定を行うクエリ

PostgreSQL

```
WITH
mst_reserved_ip AS (
  -- コード7.2.4.1参照
)
, action_log_with_reserved_ip AS (
  SELECT
      l.user_id
    , l.ip
    , l.stamp
    , m.network
    , m.description
  FROM
      action_log_with_ip AS l
    LEFT JOIN
      mst_reserved_ip AS m
    ON l.ip::inet << m.network::inet
)
SELECT *
FROM action_log_with_reserved_ip;
```

```
 user_id |       ip       |     action_time     |    network     |   description
---------+----------------+---------------------+----------------+-----------------
 U001    | 216.58.220.238 | 2016-11-03 18:00:00 |                |
 U002    | 98.139.183.24  | 2016-11-03 19:00:00 |                |
 U003    | 210.154.149.63 | 2016-11-03 20:00:00 |                |
 U004    | 127.0.0.1      | 2016-11-03 21:00:00 | 127.0.0.0/8    | localhost
 U001    | 216.58.220.238 | 2016-11-04 18:00:00 |                |
 U005    | 10.0.0.3       | 2016-11-04 19:00:00 | 10.0.0.0/8     | Private network
 U006    | 172.16.0.5     | 2016-11-04 20:00:00 | 172.16.0.0/12  | Private network
 U007    | 192.168.0.23   | 2016-11-04 21:00:00 | 192.168.0.0/16 | Private network
 U008    | 192.0.0.10     | 2016-11-04 22:00:00 | 192.0.0.0/24   | Private network
```

次のステップ 上記コード例の出力結果に示す通り、IPアドレスがマスタテーブルに定義されている場合は、networkとdescriptionが表示されています。

マスターテーブルに定義されているIPアドレスのログを除外するには、最後にWHERE句でnetwork IS NULLを指定します。下記にコード例とその出力結果を示します。

コード7.2.4.3：予約IPアドレスのログを除外するクエリ

PostgreSQL

```
WITH
mst_reserved_ip AS (
  -- コード7.2.4.1参照
)
, action_log_with_reserved_ip AS (
  -- コード7.2.4.2参照
)
SELECT *
FROM action_log_with_reserved_ip
WHERE network IS NULL;
```

▼

```
 user_id |       ip       |        stamp        | network | description
---------+----------------+---------------------+---------+-------------
 U001    | 216.58.220.238 | 2016-11-03 18:00:00 |         |
 U002    | 98.139.183.24  | 2016-11-03 19:00:00 |         |
 U003    | 210.154.149.63 | 2016-11-03 20:00:00 |         |
 U001    | 216.58.220.238 | 2016-11-04 18:00:00 |         |
```

PostgreSQL以外で特定IPアドレスを除外する

IPアドレスを扱うための特別なデータ型が用意されていない、PostgreSQL以外のミドルウェアで同様の処理を行うクエリも紹介します。詳細は「3-2-6 IPアドレスを扱う」(P.53) で前述していますが、手順として、下記の2ステップが必要です。

1 ネットワーク範囲を、最初のIPアドレスと最後のIPアドレスで表現する
2 IPアドレスを大小比較可能な形式に変換する

後者のIPアドレスを大小比較可能な形式に変換する方法として、次の2つの方法を紹介しています。

(A) IPアドレスを整数型で表現する
(B) IPアドレスを0埋めの文字列で表現する

以降のクエリでは、(B) のIPアドレスを0埋めの文字列で表現する手法を採用しています。まず、予約IPアドレスのマスタデータに、ネットワーク範囲を表す最初と最後のIPアドレスを付与したマスタテーブルを作成します。

コード7.2.4.4：ネットワーク範囲を表す最初と最後のIPアドレスを付与するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_reserved_ip_with_range AS (
  -- マスタテーブルに、ネットワーク範囲の最初のIPアドレスと最後のIPアドレスを追加する
    SELECT '127.0.0.0/8'     AS network
         , '127.0.0.0'       AS network_start_ip
         , '127.255.255.255' AS network_last_ip
         , 'localhost'       AS description
  UNION ALL
    SELECT '10.0.0.0/8'      AS network
         , '10.0.0.0'        AS network_start_ip
         , '10.255.255.255'  AS network_last_ip
         , 'Private network' AS description
  UNION ALL
    SELECT '172.16.0.0/12'   AS network
         , '172.16.0.0'      AS network_start_ip
         , '172.31.255.255'  AS network_last_ip
         , 'Private network' AS description
  UNION ALL
    SELECT '192.0.0.0/24'    AS network
         , '192.0.0.0'       AS network_start_ip
         , '192.0.0.255'     AS network_last_ip
         , 'Private network' AS description
  UNION ALL
    SELECT '192.168.0.0/16'  AS network
         , '192.168.0.0'     AS network_start_ip
         , '192.168.255.255' AS network_last_ip
         , 'Private network' AS description
)
SELECT *
FROM mst_reserved_ip_with_range;
```

```
    network     | network_start_ip | network_last_ip |   description
----------------+------------------+-----------------+-----------------
 127.0.0.0/8    | 127.0.0.0        | 127.255.255.255 | localhost
 10.0.0.0/8     | 10.0.0.0         | 10.255.255.255  | Private network
 172.16.0.0/12  | 172.16.0.0       | 172.31.255.255  | Private network
 192.0.0.0/24   | 192.0.0.0        | 192.0.0.255     | Private network
 192.168.0.0/16 | 192.168.0.0      | 192.168.255.255 | Private network
```

次のステップ 続いて、マスタテーブルのIPアドレスとログデータのIPアドレスを0埋めして、大小比較可能な形式に変換し、マスタテーブルに含まれるネットワークに該当するIPアドレスを持つログを排除します。

コード7.2.4.5：IPアドレスを0埋めした文字列に変換し、特定IPのログを排除するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_reserved_ip_with_range AS (
  -- コード7.2.4.4参照
)
, action_log_with_ip_varchar AS (
  -- アクションログのIPアドレスを0埋めの文字列で表現する
  SELECT
      *
    -- ■ PostgreSQL, Redshiftの場合、lpad関数で0埋めする
    ,    lpad(split_part(ip, '.', 1), 3, '0')
      || lpad(split_part(ip, '.', 2), 3, '0')
      || lpad(split_part(ip, '.', 3), 3, '0')
      || lpad(split_part(ip, '.', 4), 3, '0')
      AS ip_varchar

    -- ■ BigQueryの場合、split関数で配列に分解し、n番目の要素を取り出す
    --, CONCAT(
    --    lpad(split(ip, '.')[SAFE_ORDINAL(1)], 3, '0')
    --  , lpad(split(ip, '.')[SAFE_ORDINAL(2)], 3, '0')
    --  , lpad(split(ip, '.')[SAFE_ORDINAL(3)], 3, '0')
    --  , lpad(split(ip, '.')[SAFE_ORDINAL(4)], 3, '0')
    --  ) AS ip_varchar

    -- ■ Hive, SparkSQLの場合、split関数で配列に分解し、n番目の要素を取り出す
    --   ただし、コロンが特殊文字のため、バックスラッシュでエスケープを行う
    --, CONCAT(
    --    lpad(split(ip, '\\.')[0], 3, '0')
    --  , lpad(split(ip, '\\.')[1], 3, '0')
    --  , lpad(split(ip, '\\.')[2], 3, '0')
    --  , lpad(split(ip, '\\.')[3], 3, '0')
    --  ) AS ip_varchar
  FROM
    action_log_with_ip
)
, mst_reserved_ip_with_varchar_range AS (
  -- マスタテーブルのIPアドレスを0埋めの文字列で表現する
  SELECT
      *
    -- ■ PostgreSQL, Redshiftの場合、lpad関数で0埋めする
    ,    lpad(split_part(network_start_ip, '.', 1), 3, '0')
      || lpad(split_part(network_start_ip, '.', 2), 3, '0')
      || lpad(split_part(network_start_ip, '.', 3), 3, '0')
      || lpad(split_part(network_start_ip, '.', 4), 3, '0')
```

```
      AS network_start_ip_varchar
    ,    lpad(split_part(network_last_ip, '.', 1), 3, '0')
      || lpad(split_part(network_last_ip, '.', 2), 3, '0')
      || lpad(split_part(network_last_ip, '.', 3), 3, '0')
      || lpad(split_part(network_last_ip, '.', 4), 3, '0')
      AS network_last_ip_varchar

    -- ■ BigQueryの場合、split関数で配列に分解し、n番目の要素を取り出す
    --, CONCAT(
    --    lpad(split(network_start_ip, '.')[SAFE_ORDINAL(1)], 3, '0')
    --  , lpad(split(network_start_ip, '.')[SAFE_ORDINAL(2)], 3, '0')
    --  , lpad(split(network_start_ip, '.')[SAFE_ORDINAL(3)], 3, '0')
    --  , lpad(split(network_start_ip, '.')[SAFE_ORDINAL(4)], 3, '0')
    --  ) AS network_start_ip_varchar
    --, CONCAT(
    --    lpad(split(network_last_ip, '.')[SAFE_ORDINAL(1)], 3, '0')
    --  , lpad(split(network_last_ip, '.')[SAFE_ORDINAL(2)], 3, '0')
    --  , lpad(split(network_last_ip, '.')[SAFE_ORDINAL(3)], 3, '0')
    --  , lpad(split(network_last_ip, '.')[SAFE_ORDINAL(4)], 3, '0')
    --  ) AS network_last_ip_varchar

    -- ■ Hive, SparkSQLの場合、split関数で配列に分解し、n番目の要素を取り出す
    --   ただし、コロンが特殊文字のため、バックスラッシュでエスケープを行う
    --, CONCAT(
    --    lpad(split(network_start_ip, '\\.')[0], 3, '0')
    --  , lpad(split(network_start_ip, '\\.')[1], 3, '0')
    --  , lpad(split(network_start_ip, '\\.')[2], 3, '0')
    --  , lpad(split(network_start_ip, '\\.')[3], 3, '0')
    --  ) AS network_start_ip_varchar
    --, CONCAT(
    --    lpad(split(network_last_ip, '\\.')[0], 3, '0')
    --  , lpad(split(network_last_ip, '\\.')[1], 3, '0')
    --  , lpad(split(network_last_ip, '\\.')[2], 3, '0')
    --  , lpad(split(network_last_ip, '\\.')[3], 3, '0')
    --  ) AS network_last_ip_varchar
  FROM
    mst_reserved_ip_with_range
)
-- 0埋めの文字列で表現したIPアドレスで、ネットワーク範囲に含まれているかどうかを判定する
SELECT
    l.user_id
  , l.ip
  , l.ip_varchar
  , l.stamp
FROM
    action_log_with_ip_varchar AS l
  CROSS JOIN
    mst_reserved_ip_with_varchar_range AS m
GROUP BY
  l.user_id, l.ip, l.ip_varchar, l.stamp
```

```
-- 予約IPアドレスマスターに含まれないIPのログだけをHAVING句で絞り込む
HAVING
  SUM(CASE WHEN l.ip_varchar
    BETWEEN m.network_start_ip_varchar AND m.network_last_ip_varchar
    THEN 1 ELSE 0 END) = 0
;
```

```
 user_id |       ip       |  ip_varchar  |        stamp
---------+----------------+--------------+---------------------
 U001    | 216.58.220.238 | 216058220238 | 2016-11-03 18:00:00
 U002    | 98.139.183.24  | 098139183024 | 2016-11-03 19:00:00
 U003    | 210.154.149.63 | 210154149063 | 2016-11-03 20:00:00
 U001    | 216.58.220.238 | 216058220238 | 2016-11-04 18:00:00
```

ワンポイント

IPアドレスを大小比較可能な形式に変換する処理は、クエリで毎回実行するとコストが掛かるため、あらかじめ元のIPアドレスと変換後のIPアドレス双方をテーブルに保存し、クエリの種類に応じて使い分けるのがよいでしょう。

関連項目	3-2-6 IPアドレスを扱う 7-1-1 IPアドレスから国・地域を補完する

7-3

データの重複を検出する

RDBでは、適切にユニークキーを設定している場合、キーの値が重複すると自動的にエラーとなり、データの整合性が保証されます。しかし、HiveやBigQueryなど非RDBでは、データの重複を事前にチェックして弾く機能はありません。

データが重複している状態で集計すると、実態とは異なる結果となり、その誤った結果を元に誤った判断を下す可能性があります。データの重複に気付いたときは、どのデータがどれぐらい重複しているかを確認して、重複データを除外する必要があります。

もちろん、処理後ではなく事前に重複データがないことを確認すべきです。目視で確認できる件数であれば、人力での作業も無理ではありませんが、人為的な確認漏れや作業ミスが発生する可能性があります。ましてや、大量のデータから重複を目視して、手作業で確認することは非常に困難です。

本節では、マスタデータやログデータの重複を確認し、重複データを除外することで、データ活用の精度を向上させる手法を紹介します。

7-3-1 マスタデータの重複を検出する

SQL COUNT関数、COUNT(DISTINCT～)、HAVING句、COUNTウィンドウ関数

マスタデータに重複データが存在する場合、マスタデータをログデータと結合すると、ログが複数レコードとしてカウントされたり、誤った分析結果を導いてしまう危険性があります。本項では、マスタデータの重複を事前にチェックする手法をいくつか紹介します。

キーが重複しているデータの存在を確認する

マスタデータのレコードに重複が発生するケースとして、いくつか理由が考えられます。下記に一般的な原因を列挙します。

1. データをロードする際に誤って複数回ロードしてしまい、同じデータのレコードが作成された
2. あるタイミングでマスタデータの値が更新され、古いデータと新しいデータが別々のレコードとしてロードされた
3. 運用のミスなどで、同じIDを別のデータとして使い回した

いずれのケースでも、まずは分析に使用するマスタデータに重複データが存在しないことを確認するところから始めましょう。本項では下記のmst_categoriesテーブルをサンプルデータとして使用します。id=6のレコードが2月1日に書き換えられて、新旧のデータが重複して存在している状態です。

データ7.3.1.1：mst_categoriesテーブルの表示結果

```
 id |     name      |        stamp
----+---------------+---------------------
  1 | ladys_fashion | 2016-01-01 10:00:00
  2 | mens_fashion  | 2016-01-01 10:00:00
  3 | book          | 2016-01-01 10:00:00
  4 | game          | 2016-01-01 10:00:00
  5 | dvd           | 2016-01-01 10:00:00
  6 | food          | 2016-01-01 10:00:00
  7 | supplement    | 2016-01-01 10:00:00
  6 | cooking       | 2016-02-01 10:00:00
```

テーブル内にキーの重複が存在することは、テーブル全体のレコード数とキーのユニークな数を比較することで、簡単に確認できます。

コード7.3.1.1：キーの重複を確認するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    COUNT(1) AS total_num
  , COUNT(DISTINCT id) AS key_num
FROM
  mst_categories
;
```

```
 total_num | key_num
-----------+---------
         8 |       7
```

上記コード例の出力結果から、テーブル全体のレコード数とキーのユニークな数が一致していないため、mst_categoriesテーブルにはキーの重複が存在することが分かります。

キーが重複しているレコードを確認する

テーブル内にキーの重複が存在することは分かりましたが、具体的にはどのキーのレコードがどのように重複しているのでしょうか。重複しているIDを確認するためには、GROUP BYでIDごとに集約し、HAVING句を用いてレコード数が1より大きいレコードに絞り込みます。

そのIDのレコード値を確認するには、PostgreSQLとBIgQueryではstring_agg関数、Redshiftではlistagg関数、HiveとSparkSQLではcollect_list関数とconcat_ws関数を用いて、重複した値を配列として集約できます。下記コード例の出力結果で、id=6レコードで重複している名前とタイムスタンプを確認できます。

コード7.3.1.2：キーが重複しているレコードの値を確認するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    id
  , COUNT(*) AS record_num

  -- データを配列に集約し、カンマ区切りの文字列に変換
  -- ■ PostgreSQL, BigQueryの場合はstring_aggを用いる
  , string_agg(name , ',') AS name_list
  , string_agg(stamp, ',') AS stamp_list
  -- ■ Redshiftの場合はlistaggを用いる
  -- , listagg(name , ',') AS name_list
  -- , listagg(stamp, ',') AS stamp_list
  -- ■ Hive, SparkSQLの場合はcollect_listとconcat_wsを用いる
  -- , concat_ws(',', collect_list(name )) AS name_list
  -- , concat_ws(',', collect_list(stamp)) AS stamp_list
FROM
  mst_categories
GROUP BY id
HAVING COUNT(*) > 1 -- 重複しているIDに絞り込む
;
```

```
 id | record_num |  name_list   |              stamp_list
----+------------+--------------+----------------------------------------
  6 |          2 | food,cooking | 2016-01-01 10:00:00,2016-02-01 10:00:00
```

別解 string_agg関数など、値を配列にする集約関数を使用できない場合や、元のレコード形式のままで表示したい場合は、下記のコード例に示す通り、ウィンドウ関数を用いてIDの重複を数え、その値で絞り込む方法も考えられます。

コード7.3.1.3:ウィンドウ関数を用いて重複したレコードを絞り込むクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
mst_categories_with_key_num AS (
  SELECT
      *
      -- IDの重複を数える
    , COUNT(1) OVER(PARTITION BY id) AS key_num
  FROM
```

```
    mst_categories
)
SELECT
  *
FROM
  mst_categories_with_key_num
WHERE
  key_num > 1 -- IDが重複しているレコードに絞り込む
;
```

```
 id |  name   |        stamp        | key_num
----+---------+---------------------+---------
  6 | food    | 2016-01-01 10:00:00 |       2
  6 | cooking | 2016-02-01 10:00:00 |       2
```

まとめ

マスタデータのIDが重複しているケースは、基本的に何らかのミスやエラーが原因と考えられるので、まずはマスタデータのデータクレンジングを実行しましょう。

複数回のデータロードが原因であれば、データロードのフローを見直したり、毎回データを上書きすることで、データロードを何度実行しても同じ結果になることを保証するなどの改善策が考えられます。

マスタデータが途中で更新され新旧データが重複するケースであれば、新しいデータのみを残すか、タイムスタンプも含めて、タイムスタンプも含めてユニークキーにするなどの対応が考えられます。

7-3-2 ログの重複を検出する

SQL GROUP BY、MIN関数、ROW_NUMBER関数、LAG関数

マスタデータとは異なり、ログデータは正常なデータ取得でもデータが重複してしまう場合があります。例えば、ユーザーがボタンを連続で2回クリックしたり、ページのリロードでログが2重に送信されるケースです。本項では、重複を含むログデータから重複を検出し、適切に処理する方法を紹介します。

サンプルデータ

本項では下記をサンプルデータとして使用します。データ例のdup_action_logテーブルのデータには、特定商品に対するユーザーの「クリック」アクションが、セッションIDとタイムスタンプ付きでログデータに格納されています。

データ 7.3.2.1：重複のあるアクションログ(dup_action_log)テーブル

```
 session | user_id | action | products |     action_time
---------+---------+--------+----------+---------------------
 A       | U001    | click  | D001     | 2016-11-03 18:00:00
 B       | U002    | click  | D002     | 2016-11-03 19:00:00
 C       | U003    | click  | A001     | 2016-11-03 20:00:00
 D       | U004    | click  | A001     | 2016-11-03 21:00:00
 D       | U004    | click  | D001     | 2016-11-03 21:00:00
 E       | U001    | click  | D001     | 2016-11-04 18:00:00
 F       | U005    | click  | A001     | 2016-11-04 19:00:00
 G       | U006    | click  | D001     | 2016-11-04 20:00:00
 H       | U007    | click  | D002     | 2016-11-04 21:00:00
 I       | U008    | click  | A001     | 2016-11-04 22:00:00
 I       | U008    | click  | A001     | 2016-11-04 22:00:10
```

重複データを確認する

最初に重複データを確認します。前項「7-3-1 マスタデータの重複を検出する」(P.347) で紹介した手法と同様に、ユーザーと商品の組み合わせに対して、重複レコードのデータを確認します。

コード 7.3.2.1：ユーザーと商品の組み合わせに対して重複を確認するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id
  , products
  -- データを配列に集約し、カンマ区切りの文字列に変換
  -- ■ PostgreSQL, BigQueryの場合はstring_aggを用いる
  , string_agg(session, ',') AS session_list
  , string_agg(stamp  , ',') AS stamp_list
  -- ■ Redshiftの場合はlistaggを用いる
  -- , listagg(session, ',') AS session_list
  -- , listagg(stamp  , ',') AS stamp_list
  -- ■ Hive, SparkSQLの場合はcollect_listとconcat_wsを用いる
  -- , concat_ws(',', collect_list(session)) AS session_list
  -- , concat_ws(',', collect_list(stamp  )) AS stamp_list

FROM
  dup_action_log
GROUP BY
  user_id, products
HAVING
  COUNT(*) > 1
;
```

```
 user_id | products |   session_list    |              stamp_list
---------+----------+-------------------+---------------------------------------
 U001    | D001     | 989004ea,87b5725f | 2016-11-03 18:00:00,2016-11-04 18:00:00
 U008    | A001     | 3efe001c,3efe001c | 2016-11-04 22:00:00,2016-11-04 22:00:10
```

上記コード例の出力結果を確認すると、2つのケースでデータの重複が分かります。1つめのユーザーID＝U001と商品ID＝D001のログは、セッションIDが異なる上にタイムスタンプも日付を跨いでいます。ユーザーと商品の組み合わせが重複しているとしても、別個のアクションとして扱うべきでしょう。一方、ユーザーID＝U008と商品ID＝A001のログは、同一セッションでタイムスタンプも10秒しか変わりません。重複を排除して1クリックとして扱うのが良さそうです。

重複データを排除する

それでは、元々のアクションログから重複を排除したアクションログを作成してみましょう。重複排除の方針として、同一セッションIDで同一商品に対する同一アクションの場合は、タイムスタンプが最も古いデータのみを残す処理を考えてみましょう。

重複を排除するために取捨選択しなければならないカラムがタイムスタンプだけであれば、下記のコード例に示す通り、GROUP BYで集約したあと、MIN(stamp)で最も古いタイムスタンプだけを抽出する方法が簡単です。

コード7.3.2.2：GROUP BYとMINを用いて重複を排除するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    session
  , user_id
  , action
  , products
  , MIN(stamp) AS stamp
FROM
  dup_action_log
GROUP BY
  session, user_id, action, products
;
```

▼

```
 session  | user_id | action | products |        stamp
----------+---------+--------+----------+---------------------
 989004ea | U001    | click  | D001     | 2016-11-03 18:00:00
 47db0370 | U002    | click  | D002     | 2016-11-03 19:00:00
 1cf7678e | U003    | click  | A001     | 2016-11-03 20:00:00
 fe05e1d8 | U004    | click  | D001     | 2016-11-03 21:00:00
```

```
5eb2e107 | U004    | click  | A001     | 2016-11-03 21:00:00
87b5725f | U001    | click  | D001     | 2016-11-04 18:00:00
eee2bb21 | U005    | click  | A001     | 2016-11-04 19:00:00
5d5b0997 | U006    | click  | D001     | 2016-11-04 20:00:00
111f2996 | U007    | click  | D002     | 2016-11-04 21:00:00
3efe001c | U008    | click  | A001     | 2016-11-04 22:00:00
```

上記コード例の出力結果では、重複していたユーザーID＝U008と商品ID＝A001のログは、タイムスタンプが古いログのみが残されています。

別解 タイムスタンプ以外のカラムも残したり、取捨選択するデータがMIN関数やMAX関数などの単純な集約関数では取得できないケースでは、上記の方法は使用できません。

より一般的な方法としては、下記のコード例に示す通り、元データにROW_NUMBERウィンドウ関数を用いて重複するデータに順序を振り、付与した順序の最初のレコードだけを残す方法があります。

コード7.3.2.3：ROW_NUMBERを用いて重複を排除するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
dup_action_log_with_order_num AS (
  SELECT
      *
      -- 重複しているデータに順序を付与する
    , ROW_NUMBER()
        OVER(
          PARTITION BY session, user_id, action, products
          ORDER BY stamp
        ) AS order_num
  FROM
    dup_action_log
)
SELECT
    session
  , user_id
  , action
  , products
  , stamp
FROM
  dup_action_log_with_order_num
WHERE
  order_num = 1 -- 重複しているデータの1番目のみを残す
;
```

上記コード例の出力結果は、前述のコード7.3.2.2と同様です。

別解 本項のサンプルログはセッションIDを保持しているため、同一のユーザーと商品の組み合わせでも、異なるセッションであれば別のログとして扱いました。しかし、セッションIDなどが利用できないケースもあります。その場合はタイムスタンプの間隔をチェックして、一定時間以内のログは重複するものと判断する手法もあります。

コード7.3.2.4：前回のアクションからの経過時間を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
dup_action_log_with_lag_seconds AS (
  SELECT
      user_id
    , action
    , products
    , stamp
    -- 同じユーザーと商品の組み合わせに対して、前回のアクションからの経過時間を計算する
    -- ■ PostgreSQLの場合、timestmap型に変換して差分を取り、EXTRACT(epoc ~)で秒数に変換
     , EXTRACT(epoch from stamp::timestamp - LAG(stamp::timestamp)
         OVER(
           PARTITION BY user_id, action, products
           ORDER BY stamp
         )) AS lag_seconds
    -- ■ Redshiftの場合、datediff関数を秒単位で指定
    -- , datediff(second, LAG(stamp::timestamp)
    --     OVER(
    --       PARTITION BY user_id, action, products
    --       ORDER BY stamp
    --     ), stamp::timestamp) AS lag_seconds
    -- ■ BigQueryの場合、unix_seconds関数で秒単位のUNIX時間に変換して差分を取る
    -- , unix_seconds(timestamp(stamp)) - LAG(unix_seconds(timestamp(stamp)))
    --     OVER(
    --       PARTITION BY user_id, action, products
    --       ORDER BY stamp
    --     ) AS lag_seconds
    -- ■ Hive, SparkSQLの場合、unix_timestamp関数で秒単位のUNIX時間に変換して差分を取る
    -- , unix_timestamp(stamp) - LAG(unix_timestamp(stamp))
    --     OVER(
    --       PARTITION BY user_id, action, products
    --       ORDER BY stamp
    -- ■ SparkSQLの場合、下記のフレーム指定を追加する
    --       ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING
    --     ) AS lag_seconds
  FROM
    dup_action_log
)
SELECT
    *
FROM
  dup_action_log_with_lag_seconds
ORDER BY
  stamp;
```

```
 user_id | action | products |        stamp        | lag_seconds
---------+--------+----------+---------------------+-------------
 U001    | click  | D001     | 2016-11-03 18:00:00 |
 U002    | click  | D002     | 2016-11-03 19:00:00 |
 U003    | click  | A001     | 2016-11-03 20:00:00 |
 U004    | click  | D001     | 2016-11-03 21:00:00 |
 U004    | click  | A001     | 2016-11-03 21:00:00 |
 U001    | click  | D001     | 2016-11-04 18:00:00 |       86400
 U005    | click  | A001     | 2016-11-04 19:00:00 |
 U006    | click  | D001     | 2016-11-04 20:00:00 |
 U007    | click  | D002     | 2016-11-04 21:00:00 |
 U008    | click  | A001     | 2016-11-04 22:00:00 |
 U008    | click  | A001     | 2016-11-04 22:00:10 |          10
```

上記コード例の出力結果でlag_secondsカラムを確認すると、重複していないレコードではNULLとなり、ユーザーIDと商品IDが重複しているレコードに関しては、重複している前回ログのタイムスタンプからの経過時間が計算されています。

続いて、算出した経過時間が一定時間以内であるログを排除します。下記のコード例に示すクエリでは、30分以内の同一アクションは重複ログとして排除しています。

コード7.3.2.5：30分以内の同一アクションを重複として排除するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
dup_action_log_with_lag_seconds AS (
  -- コード8.3.2.3参照
)
SELECT
    user_id
  , action
  , products
  , stamp
FROM
  dup_action_log_with_lag_seconds
WHERE
  (lag_seconds IS NULL OR lag_seconds >= 30 * 60)
ORDER BY
  stamp
;
```

```
 user_id | action | products |        stamp
---------+--------+----------+---------------------
 U001    | click  | D001     | 2016-11-03 18:00:00
 U002    | click  | D002     | 2016-11-03 19:00:00
 U003    | click  | A001     | 2016-11-03 20:00:00
 U004    | click  | D001     | 2016-11-03 21:00:00
 U004    | click  | A001     | 2016-11-03 21:00:00
 U001    | click  | D001     | 2016-11-04 18:00:00
 U005    | click  | A001     | 2016-11-04 19:00:00
 U006    | click  | D001     | 2016-11-04 20:00:00
 U007    | click  | D002     | 2016-11-04 21:00:00
 U008    | click  | A001     | 2016-11-04 22:00:00
```

上記コード例の出力結果として、セッションIDを利用できない状態でも、一定時間以内でのログの重複を排除することができます。

まとめ

ログデータに重複を含んだままのレポート作成は、実態とは異なる値から間違った判断を下す原因になりかねません。まずは分析するデータをよく確認し、どのような方針でクレンジングするか検討しましょう。

関連項目	3-2-5 日付・時刻を計算する 7-3-1 マスタデータの重複を検出する

7-4

複数のデータセットを比較する

集計作業では、複数のデータセットを比較して、何かしらの判断するシーンがあります。本節では複数のデータセットを比較して、データが変化した部分を確認したり、どの程度の相違があるのか確認する際に、有効な手順とSQLを紹介します。

まずは、複数のデータセットを比較する必要があるシーンとして考えられるものを2つ紹介します。

データの差分を抽出する場面

同質のデータ、もしくは同じSQLでも異なる期間の集計結果を比べて、追加・変更がないか、削除・欠損がないかを調べるシーンでは、数十件のマスタデータであれば、目視でもその変更内容を確認できるかもしれません。しかし、大量のマスタデータで何が変更されたかを調べるには、目視での確認は不可能です。

例えば、市区町村の合併に伴う郵便番号の変更を取り込む際に、追加・変更・削除されるものが分かれば、ユーザーや店舗などの住所データで該当する郵便番号を登録している、更新が必要な対象データを確認できます。目視での確認は無理があり、SQLを利用すれば効率的に確認することが可能です。

データのランキングを比較する場面

例えば、Webサイト内で人気記事ランキングや旬の検索条件を表示するモジュールを想定しましょう。人気記事ランキングなどの表示は、変化のない状態だと頻繁に訪問するユーザーには、興味のない情報と化してしまいます。実際にサービス利用者からの指摘もあり、ランキングの集計期間や条件を変更し、より活性化させる目的で作り直すケースを考えてみます。

集計期間や表示条件を変更したところで、果たして旧ロジックに比べてどの程度の変化があったのか、リリースするまで不明といった状況は避けたいところです。過去のデータで旧ロジックと新ロジックを目視で確認しても、明確な数値化はできず、「何となく変わっているかも」「それっぽい」などの感覚的、かつ定性的な捉え方になりかねません。

こうしたシーンでは、ロジックの結果として出力されるランキングの類似度を数値化することで、新旧ロジックの変化を定量的に説明可能になります。

本節では、上記にあげた2つの場面で有効な手順やSQLを紹介します。

7-4-1 データの差分を抽出する

SQL LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN、IS DISTINCT FROM演算子

マスタデータを用いて何らかのデータ分析を行う場合、分析のタイミングでマスタデータの内容が違い、分析結果が異なることが起こり得ます。そうしたケースでは、分析結果が異なることの根拠を説明するため、マスタデータの変更内容を確認したくなります。本項では、異なるタイミングで取得された2つのマスタデータを用いて、データの差分を計算する方法を紹介します。

本項で用いる2つのマスタデータを下記に示します。それぞれ2016年12月1日と2017年1月1日時点での商品マスタテーブルであり、商品IDと商品名、価格、更新タイムスタンプを保持しています。商品IDは、このマスタテーブルの主キーとなっています。

データ7.4.1.1:2016年12月1日時点での商品マスタ(mst_products_20161201)テーブル

```
 product_id | name | price |     updated_at
------------+------+-------+---------------------
 A001       | AAA  |   300 | 2016-11-03 18:00:00
 A002       | AAB  |   400 | 2016-11-03 19:00:00
 B001       | BBB  |   500 | 2016-11-03 20:00:00
 B002       | BBD  |   300 | 2016-11-03 21:00:00
 C001       | CCA  |   400 | 2016-11-04 18:00:00
 D001       | DAA  |   500 | 2016-11-04 19:00:00
```

データ7.4.1.2：2017年1月1日時点での商品マスタ(mst_products_20170101)テーブル

```
 product_id | name | price |     updated_at
------------+------+-------+---------------------
 A001       | AAA  |   300 | 2016-11-03 18:00:00
 A002       | AAB  |   400 | 2016-11-03 19:00:00
 B002       | BBD  |   300 | 2016-11-03 21:00:00
 C001       | CCA  |   500 | 2016-12-04 18:00:00
 D001       | DAA  |   500 | 2016-11-04 19:00:00
 D002       | DAD  |   500 | 2016-12-04 19:00:00
```

追加されたマスタデータを抽出する

新しいマスタテーブルに追加されたレコードを抽出します。2つのテーブルで、片方にのみ存在するレコードを抽出するには、外部結合（OUTER JOIN）を用います。ここでは新しいマスタテーブルにのみ存在するレコードを抽出するので、新しいテーブルに対して古いテーブルをLEFT OUTER JOINして、古いテーブルのカラムがNULLとなるレコードを抽出します。

コード7.4.1.1：追加されたマスタデータを抽出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
  new_mst.*
FROM
    mst_products_20170101 AS new_mst
  LEFT OUTER JOIN
    mst_products_20161201 AS old_mst
  ON
    new_mst.product_id = old_mst.product_id
WHERE
  old_mst.product_id IS NULL
;
```

```
 product_id | name | price |     updated_at
------------+------+-------+---------------------
 D002       | DAD  |   500 | 2016-12-04 19:00:00
```

上記コード例の出力結果を確認すると、12月4日に追加された商品ID＝D002のレコードが抽出されていることが分かります。

削除されたマスタデータを抽出する

続いて、削除されたマスタデータを抽出するクエリを考えてみましょう。前述とは逆に古いテーブルのみに存在するレコードを抽出することになります。

前術のコード例でLEFT OUTER JOINのテーブル順序を逆にしても抽出できますが、RIGHT OUTER JOINを使用できるのであれば、下記コード例のクエリでも同様の結果を得られます。

コード7.4.1.2：削除されたマスタデータを抽出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
  old_mst.*
FROM
    mst_products_20170101 AS new_mst
  RIGHT OUTER JOIN
    mst_products_20161201 AS old_mst
  ON
    new_mst.product_id = old_mst.product_id
WHERE
  new_mst.product_id IS NULL
;
```

```
 product_id | name | price |     updated_at
------------+------+-------+---------------------
 B001       | BBB  |   500 | 2016-11-03 20:00:00
```

上記コード例の出力結果を確認すると、新しいマスタテーブルには存在しない、商品ID＝B001のレコードが抽出されています。

更新されたマスタデータを抽出する

追加・削除に続いて、更新されたマスタデータを抽出するクエリを考えてみましょう。古いテーブルと新しいテーブル双方に存在し、特定カラムの値が異なるレコードを抽出します。

下記に示すクエリでは、内部結合で2つのマスタデータを結合し、タイムスタンプの値が異なるレコードを抽出しています。コード例では、値が変更されると必ずタイムスタンプが更新されることを前提にしていますが、必要であれば、全カラムの内容を確認する条件をWHERE句に追加すると良いでしょう。

コード7.4.1.3：更新されたマスタデータを抽出するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
SELECT
    new_mst.product_id
  , old_mst.name AS old_name
  , old_mst.price AS old_price
  , new_mst.name AS new_name
  , new_mst.price AS new_price
  , new_mst.updated_at
FROM
    mst_products_20170101 AS new_mst
  JOIN
    mst_products_20161201 AS old_mst
  ON
    new_mst.product_id = old_mst.product_id
WHERE
  -- 更新時刻が異なるレコードのみを抽出する
  new_mst.updated_at <> old_mst.updated_at
;
```

```
 product_id | old_name | old_price | new_name | new_price |     updated_at
------------+----------+-----------+----------+-----------+---------------------
 C001       | CCA      |       400 | CCA      |       500 | 2016-12-04 18:00:00
```

上記コード例の結果を確認すると、価格が400円から500円に更新されている、商品ID＝C001のレコードが抽出されていることが分かります。

変更されたマスタデータをすべて抽出する

本項の最後に、「追加」「削除」「更新」されたレコードをすべて抽出するクエリを考えましょう。

ここまでで利用した「LEFT OUTER JOIN」「RIGHT OUTER JOIN」「(INNER) JOIN」を含む結合が必要なので、下記のコード例に示す通り、FULL OUTER JOINを用います。

変更されたレコードの条件は、両者のタイムスタンプが一致しないことなので、一見すると「new.stamp <> old.stamp」で構わないと勘違いします。しかし、OUTER JOINで一方のデータがない場合、欠如したカラムはNULLになるため、<>演算子だけではうまく比較できません。

NULLを含むレコードで2つのカラムが一致しないことを指定するには、「IS DISTINCT FROM」演算子を利用します。ただし、IS DISTINCT FROM演算子が実装されていないミドルウェアでは、COALESCE関数でNULLを排除した上で<>演算子を用いて代用します。

変更されたレコードが抽出できたら、SELECT句中で「追加」「削除」「更新」の判定を行ないます。古いテーブルの値がNULLであれば「追加」、新しいテーブルの値がNULLであれば「削除」、それ以外(両者ともNULLでない場合)で両者のタイムスタンプが異なる場合は「更新」と判定します。

コード7.4.1.4：変更されたマスタデータをすべて抽出するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    COALESCE(new_mst.product_id, old_mst.product_id) AS product_id
  , COALESCE(new_mst.name      , old_mst.name      ) AS name
  , COALESCE(new_mst.price     , old_mst.price     ) AS price
  , COALESCE(new_mst.updated_at, old_mst.updated_at) AS updated_at
  , CASE
      WHEN old_mst.updated_at IS NULL THEN 'added'
      WHEN new_mst.updated_at IS NULL THEN 'deleted'
      WHEN new_mst.updated_at <> old_mst.updated_at THEN 'updated'
    END AS status
FROM
    mst_products_20170101 AS new_mst
  FULL OUTER JOIN
    mst_products_20161201 AS old_mst
  ON
    new_mst.product_id = old_mst.product_id
WHERE
  -- ■ PostgreSQLの場合、IS DISTINCT FROM演算子を用いて、NULLを含めた比較ができる
  new_mst.updated_at IS DISTINCT FROM old_mst.updated_at
  -- ■ Redshift, BigQueryの場合、
  --   IS DISTINCT FROM句の代わりにCOALESCE関数でNULLをデフォルト値に置き換えて比較する
  --    COALESCE(new_mst.updated_at, 'infinity')
  -- <> COALESCE(old_mst.updated_at, 'infinity')
;
```

```
 product_id | name | price |     updated_at      | status
------------+------+-------+---------------------+---------
 C001       | CCA  |   500 | 2016-12-04 18:00:00 | updated
 D002       | DAD  |   500 | 2016-12-04 19:00:00 | added
 B001       | BBB  |   500 | 2016-11-03 20:00:00 | deleted
```

上記コード例の出力結果から、「追加」「削除」「更新」されたレコードがすべて抽出されていることが分かります。

7-4-2 2つのランキングの類似度を計算する

SQL RANK関数、SUM関数、COUNT関数　　分析 スピアマンの順位相関係数

データ分析では、さまざまな結果をランキング形式でまとめることが多々ありますが、どのようにスコアリングを行うかで、ランキングの結果も異なってきます。

例えば、「6-1-2 ページ毎の訪問回数・訪問者数・ページビューを集計する」(P.247) で紹介した3つの指標、「訪問回数」「訪問者数」「ページビュー」で各ページのランキングを作成できます。しかし、いずれが最適なスコアリングであるかを決めることは容易ではありません。

本項では、2つのランキングの類似度を計算し、ランキングを定量的に評価するための手法を紹介します。なお、本項で使うサンプルデータとして、下記に示すWebページのアクセスログから、「訪問回数」「訪問者数」「ページビュー」の3つの指標で作成したページランキングを扱います。

データ7.4.2.1：Webページのアクセスログ(access_log) テーブル

```
        stamp        | short_session | long_session |  path
---------------------+---------------+--------------+---------
 2016-10-01 12:00:00 | 0CVKaz        | 1CwlSX       | /detail
 2016-10-01 13:00:00 | 0CVKaz        | 1CwlSX       | /detail
 2016-10-01 13:00:00 | 1QceiB        | 3JMO2k       | /search
 2016-10-01 14:00:00 | 1QceiB        | 3JMO2k       | /detail
 2016-10-01 15:00:00 | 1hI43A        | 6SN6DD       | /search
 2016-10-01 16:00:00 | 1hI43A        | 6SN6DD       | /detail
 2016-10-01 17:00:00 | 2bGs3i        | 1CwlSX       | /top
 2016-10-01 18:00:00 | 2is8PX        | 7Dn99b       | /search
```

指標ごとのランキングを作成する

下記のコード例に、「訪問回数」「訪問者数」「ページビュー」に基づくランキングを作成するクエリを示します。

コード7.4.2.1：3つの指標に基づくランキングを作成するクエリ

```
WITH
path_stat AS (
 -- パス別の訪問回数、訪問者数、ページビューを計算する
 SELECT
     path
   , COUNT(DISTINCT long_session) AS access_users
   , COUNT(DISTINCT short_session) AS access_count
   , COUNT(*) AS page_view
 FROM
   access_log
 GROUP BY
   path
)
, path_ranking AS (
 -- 訪問回数、訪問者数、ページビューごとにランキングを作成する
 SELECT 'access_user'  AS type, path, RANK() OVER(ORDER BY access_users DESC) AS rank
 FROM path_stat
UNION ALL
 SELECT 'access_count' AS type, path, RANK() OVER(ORDER BY access_count DESC) AS rank
 FROM path_stat
UNION ALL
 SELECT 'page_view'    AS type, path, RANK() OVER(ORDER BY page_view    DESC) AS rank
 FROM path_stat
)
SELECT *
FROM
 path_ranking
ORDER BY
 type, rank
;
```

```
     type     |  path   | rank
--------------+---------+------
 access_count | /top    |    1
 access_count | /detail |    2
 access_count | /search |    3
 access_user  | /detail |    1
 access_user  | /search |    2
 access_user  | /top    |    3
 page_view    | /top    |    1
 page_view    | /search |    2
 page_view    | /detail |    3
```

上記コード例の結果には、typeカラムに指標の種類、訪問回数（access_count）と訪問者数（access_user）、ページビュー（page_view）が格納され、各指標について「/top」「/detail」「/search」の3ページのランキングが示されています。

パスごとにランキング順位の差分を計算する

前記の出力結果であるランキングに対して、あるパスの2つの指標におけるランキング順位の差分を計算してみましょう。例えば、パス「/top」は訪問回数のランキングでは1位、訪問者数のランキングでは3位です。このとき、ランキング順位の差分（diff）を両者の順位差の2乗と定義します。

下記に示すコード例では、パスをキーとしてランキングを自己結合し、2つの指標におけるランキング順位の差分を計算しています。

コード7.4.2.2：パスごとにランキング順位の差分を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
path_stat AS (
 -- コード7.4.2.1参照
)
, path_ranking AS (
 -- コード7.4.2.1参照
)
, pair_ranking AS (
 SELECT
     r1.path
   , r1.type AS type1
   , r1.rank AS rank1
   , r2.type AS type2
   , r2.rank AS rank2
     -- ランキング順位の差を計算する
   , POWER(r1.rank - r2.rank, 2) AS diff
 FROM
     path_ranking AS r1
   JOIN
     path_ranking AS r2
     ON r1.path = r2.path
)
SELECT
 *
FROM
 pair_ranking
ORDER BY
 type1, type2, rank1
;
```

```
  path   |    type1     | rank1 |    type2     | rank2 | diff
---------+--------------+-------+--------------+-------+------
 /top    | access_count |     1 | access_count |     1 |    0
 /detail | access_count |     2 | access_count |     2 |    0
 /search | access_count |     3 | access_count |     3 |    0
 /top    | access_count |     1 | access_user  |     3 |    4
```

```
 /detail | access_count |     2 | access_user  |     1 |    1
 /search | access_count |     3 | access_user  |     2 |    1
 /top    | access_count |     1 | page_view    |     1 |    0
 /detail | access_count |     2 | page_view    |     3 |    1
 /search | access_count |     3 | page_view    |     2 |    1
...
```

上記コード例の出力結果を確認すると、「訪問回数」のランキングに対して、同じく「訪問回数」のランキングを結合している部分は、順位は同じであるため当然diffの値もすべて0です。「訪問回数」と「訪問者数」の比較では、パス「/top」が1位と3位であるため、diffの値は(1-3)^2＝4となっています。

スピアマンの順位相関係数を計算する

各パスにおけるランキング順位の差分が計算できたら、続いてランキング全体の類似度を計算しましょう。本項では「スピアマンの順位相関係数」と呼ばれる指標を用いて、ランキングの類似度を計算します。

スピアマンの順位相関係数は、2つのランキング順位が完全に一致している場合は「1.0」、まったく一致していない場合は「-1.0」となる性質を持っています。スピアマンの順位相関係数の定義式は下記の通りです。

$$Spearman = 1 - \frac{6\sum diff}{N^3 - N} \quad (N：順位ペアの数)$$

上記の式に基づき、スピアマンの順位相関係数を計算するクエリを下記のコード例に示します。

コード7.4.2.3：スピアマンの順位相関係数でランキングの近さを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
path_stat AS (
  -- コード7.4.2.1参照
)
, path_ranking AS (
  -- コード7.4.2.1参照
)
, pair_ranking AS (
  -- コード7.4.2.2参照
)
SELECT
    type1
  , type2
  , 1 - (6.0 * SUM(diff) / (POWER(COUNT(1), 3) - COUNT(1))) AS spearman
FROM
```

```
  pair_ranking
GROUP BY
  type1, type2
ORDER BY
  type1, spearman DESC
;
```

```
    type1     |    type2     | spearman
--------------+--------------+----------
 access_count | access_count |      1.0
 access_count | page_view    |      0.5
 access_count | access_user  |     -0.5
 access_user  | access_user  |      1.0
 access_user  | access_count |     -0.5
 access_user  | page_view    |     -1.0
 page_view    | page_view    |      1.0
 page_view    | access_count |      0.5
 page_view    | access_user  |     -1.0
```

上記コード例の出力結果では、同じ指標のランキング比較ではランキングが完全に一致するため、スピアマンの順位相関係数もすべて1.0になっています。「訪問者数」と「ページビュー」のランキングの比較では、ランキングの順位がまったく異なるため、スピアマンの順位相関係数は-1.0となっています。

まとめ

2つのランキング類似度を1つの値に集約して表現することで、異なるスコアリングで作成したランキングがどれだけ類似しているのかを確認できます。例えば、理想的なランキングを人手で作成した上で、その理想にどれだけ近いランキングを自動生成できるかを定量的に評価できます。

Chapter 8

データを武器にするための分析術

課題発見と問題解決の手法に続いて、
検索エンジンの最適化をはじめ、
レコメンドの作成や並び順の最適化など、
より発展的なデータ活用と
新たな価値を創造する手法を紹介します。

8-1
検索機能を評価する

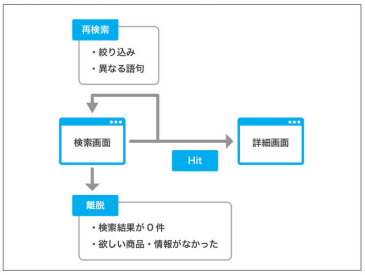
運用しているサービスで、サイト内検索機能を提供している場合、ユーザーがどのような検索クエリを入力し、どのような結果を得ているかを分析する作業は重要なことです。本節では、検索に関連する行動ログや予め人力で用意した評価用データを用いて、検索機能を定量的に評価し、改善する手法を紹介します。

検索におけるユーザー行動

検索機能におけるユーザー行動のパターンをまとめてみましょう。ユーザーが何らかの検索クエリを入力すると、そのクエリ結果を表示する検索画面に遷移します。ユーザーは検索結果に求めている情報が存在すれば、該当情報の詳細画面へ遷移します。求めている情報が存在しなければ、再度検索を実行するかそのまま離脱します。検索におけるユーザー行動をまとめたものが下図です。

ユーザーがどのようなクエリで検索し、その検索結果に対してどのような行動を取っているのか分かれば、検索機能を担当しているエンジニアに機能改善を求めることが可能です。検索機能を改善するには、どのような対応が必要になるのか把握しましょう。

図 8.1.0.1：検索におけるユーザーの行動



検索機能の改善方法

検索機能を改善する方法には、下記に挙げる項目があります。それぞれ、どのように対応するのか概略を説明します。

1. 検索ワードの揺らぎを吸収するため、「シノニム辞書を追加」する
2. 検索ワードを検索エンジンが理解できるように「ユーザー辞書を追加」する
3. 検索結果がよりユーザーの求める表示順になるように「ソート順の見直し」を行う

シノニム辞書の追加

ユーザーが常に正式な商品名で検索するとは限りません。商品の略称や通称で検索するケースも多々あります。また、英語の商品名をカタカナで検索する場合や、時には間違った検索ワードで検索する場合もあります。こうした検索ワードの揺らぎを吸収するため、正式な商品名に対して、別の検索ワードでもヒットするように、シノニム（同義語）辞書を設定します。

ユーザー辞書の追加

検索エンジンによっては、単語を独自アルゴリズムで分解してインデックスを作成し、入力された検索ワードに対して、そのインデックスからデータを検索して、結果を返却するものがあります。その場合、必ずしもサービス提供側が意図するように単語が分解されるとは限りません。

例えば、検索ワード「ウイスキー」が「ウイ」と「スキー」に分解された形でインデックスされると、スキーで検索してもウイスキーがヒットしてしまいます。こうしたケースでは、「ウイスキー」で1単語であることを予め登録しておけば、適切にインデックスが作成されます。

ソート順の見直し

検索エンジンは、条件に合致するアイテムをただ表示するだけでなく、検索クエリとの関連度を数値化し、スコアが高い順に並べ替えるランキング機能も特徴です。関連度のスコアを計算するための元データとして、検索ワードとアイテムに含まれる文章の一致度、検索ワードが出現した場所や回数、アイテムの更新日やそれまでのアクセス数など、多岐に渡るデータを組み合わせます。これらデータの組み合わせをチューニングして、最適な順序で検索結果を表示するには、適切な評価指標や評価方法が重要となります。

検索機能を評価するためのデータ

本節前半（8-1-1～8-1-5）では、データ8.1.0.1に示す、検索結果と詳細ページのアクセスログデータを用いて、検索に関するユーザー行動指標を分析します。

検索結果の表示ログは、検索結果ページが複数ページに跨がる場合でも、1レコードのログになるように整形しています。これは、検索結果の2ページ目以降を閲覧する場合と、新たな検索を実行した場

合を明確に区別するためです。検索結果の２ページ目以降を閲覧する場合に、新しいレコードが追加されるログを扱うケースでは、検索結果の１ページ目のログに絞り込むなどの前処理を行います。

ちなみに、本書では、テキストボックスに検索ワードを入力して検索する、フリーキーワード検索を前提にレポートやクエリを紹介しますが、カテゴリ検索でも流用可能なレポートもあるので、参考にしてください。

データ8.1.0.1:検索結果と詳細ページのアクセスログ(access_log)テーブル(拡張表示)

```
-[ RECORD 1 ]----------------------------------------------------------------------
stamp      | 2017-01-05 23:35:13
session    | 0CVKaz
action     | search
keyword    | sql
url        | http://www.example.com/search_result?q=sql
referrer   |
result_num | 16851
-[ RECORD 2 ]----------------------------------------------------------------------
stamp      | 2017-01-05 23:36:08
session    | 0CVKaz
action     | search
keyword    | postgresql
url        | http://www.example.com/search_result?q=postgresql
referrer   |
result_num | 889
-[ RECORD 3 ]----------------------------------------------------------------------
stamp      | 2017-01-05 23:37:17
session    | 0CVKaz
action     | detail
keyword    | NULL
url        | http://www.example.com/detail?id=170
referrer   | http://www.example.com/search_result?q=postgresql
result_num |
```

8-1-1 NoMatch率とそのワードを集計する

SQL SUM(CASE ～)、AVG(CASE ～)　　分析 NoMatch率、NoMatchワード

ユーザーが検索を実行した結果、何もヒットしない状況は、ユーザーにマイナスの印象を与えてしまいます。再度、別な条件で検索してくれれば良いのですが、検索をやり直さずに離脱してしまう可能性もあります。検索結果が０件の場合、そもそも入力された検索ワードに対応する商品やコンテンツがない場合は仕方ありませんが、検索エンジンが正しくワードを把握できず、揺らぎを吸収できていない場合は、何らかの対応を施し、検索結果が０件になることを回避しましょう。

NoMatch率を集計する

本書では、検索総数の中で検索結果が0件を返した検索結果の割合を、「NoMatch率」と呼びます。NoMatch率は下記の定義で求めます。

NoMatch率　＝検索結果が0の検索回数（NoMatch数）　/ 検索総数

検索エンジンが正しくワードを把握できなかったり、検索ワードの揺らぎを吸収できていない可能性を調べるため、検索結果が0件の割合（NoMatch率）を集計するクエリを、下記のコード例に示します。

コード8.1.1.1：NoMatch率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
 -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、substringで日付部分を抽出する
   substring(stamp, 1, 10) AS dt
 -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、substrが利用できる
 -- substr(stamp, 1, 10) AS dt
 , COUNT(1) AS search_count
 , SUM(CASE WHEN result_num = 0 THEN 1 ELSE 0 END) AS no_match_count
 , AVG(CASE WHEN result_num = 0 THEN 1.0 ELSE 0.0 END) AS no_match_rate
FROM
 access_log
WHERE
 action = 'search'
GROUP BY
 -- ■ PostgreSQL, Redshift, BigQueryの場合、
 --   SELECT句で定義した別名をGROUP BYに指定できる
 dt
 -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、
 --   SELECT句で別名を指定する前の式をGROUP BYに指定できる
 -- substring(stamp, 1, 10)
;
```

```
     dt     | search_count | no_match_count |  no_match_rate
------------+--------------+----------------+-----------------
 2017-01-05 |         3510 |            235 |         0.0669
 2017-01-06 |         3541 |            261 |         0.0737
```

NoMatchワードを集計する

NoMatchとなった際の検索ワードを抽出するクエリをコード8.1.1.2に示します。
どのようなワードが0件になったか集計して、シノニム辞書、ユーザー辞書の追加として適切なものがあれば、反映させましょう。

コード8.1.1.2：NoMatchワードを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
search_keyword_stat AS (
  -- 検索キーワード全体の集計結果
  SELECT
      keyword
    , result_num
    , COUNT(1) AS search_count
    , 100.0 * COUNT(1) / COUNT(1) OVER() AS search_share
  FROM
    access_log
  WHERE
    action = 'search'
  GROUP BY
    keyword, result_num
)
-- NoMatchワードの集計結果
SELECT
    keyword
  , search_count
  , search_share
  , 100.0 * search_count / SUM(search_count) OVER() AS no_match_share
FROM
  search_keyword_stat
WHERE
  -- 検索結果0件のキーワードに絞り込む
  result_num = 0
;
```

```
    keyword    | search_count | search_share | no_match_share
---------------+--------------+--------------+----------------
 pstgresql     |           13 |       0.1843 |         2.6209
 postgresql 10 |           25 |       0.3545 |         5.0403
...
```

ワンポイント

本項では、フリーワード検索を前提として説明していますが、条件を選択するカテゴリ検索でも、説明したNoMatch率は重要な指標となり得ます。0件の検索結果にユーザーが遭遇しないように工夫を施しましょう。

8-1-2 再検索率とそのワードを集計する

SQL LEAD関数、SUM(CASE～)、AVG(CASE～)　分析 再検索率、再検索ワード

検索結果に満足せず、新たな検索を実行したユーザーの行動には、検索を改善する大きなヒントがあります。複数回の検索を実行した際に入力された検索ワードを比較することで、ユーザーが求めているコンテンツの把握が可能になります。

再検索率を集計する

再検索率とは、ユーザーが検索結果が表示されたにも関わらず、どの結果もクリックせず新しい検索を実行した割合を示します。この再検索率をユーザーの行動ログから計算する方法を考えてみましょう。

検索結果の表示ログ（action=search）と、検索画面から詳細画面に遷移したログ（action=detail）を時系列順に並べ、各行に次行のアクションを付与します。下記コード例に、次行のアクションをウィンドウ関数を用いて取得するクエリを示します。

コード8.1.2.1：検索画面と詳細画面のアクセスログに次行のアクションを付与するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_action AS (
  SELECT
      stamp
    , session
    , action
    , LEAD(action)
        -- ■ PostgreSQL, Hive, Redshift, BigQueryの場合は下記
        OVER(PARTITION BY session ORDER BY stamp ASC)
        -- ■ SparkSQLの場合、フレーム指定が必要
        -- OVER(PARTITION BY session ORDER BY stamp ASC
        --   ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
      AS next_action
  FROM
    access_log
)
SELECT *
FROM access_log_with_next_action
ORDER BY
  session, stamp
;
```

```
        stamp        | session | action | next_action
---------------------+---------+--------+-------------
 2017-01-05 23:35:13 | 0CVKaz  | search | search  --> 再検索
```

```
 2017-01-05 23:36:08 | 0CVKaz  | search | detail
 2017-01-05 23:37:17 | 0CVKaz  | detail | detail
 2017-01-05 23:38:38 | 0CVKaz  | detail | search
 2017-01-05 23:40:10 | 0CVKaz  | search | detail
 2017-01-05 23:41:43 | 0CVKaz  | detail | search
 2017-01-05 23:43:10 | 0CVKaz  | search |
 2017-01-05 23:34:57 | 1QceiB  | search | search  --> 再検索
 2017-01-05 23:35:37 | 1QceiB  | search | detail
 2017-01-05 23:36:48 | 1QceiB  | detail | detail
 2017-01-05 23:37:27 | 1QceiB  | detail | search
 2017-01-05 23:38:52 | 1QceiB  | search | detail
 2017-01-05 23:39:50 | 1QceiB  | detail | search
...
```

上記コード例の出力結果では、actionとnext_action両方がsearchのレコードを再検索と見做して、このレコード数を再検索数とします。また、actionがsearchのレコードを集計したものを検索総数とします。検索総数に占める再検索数の割合が「再検索率」となります。

次のステップ 上記の定義に従って、再検索率を集計するクエリを下記のコード例に示します。

コード8.1.2.2：再検索率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_action AS (
  -- コード8.1.1.1参照
)
SELECT
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、substringで日付部分を抽出する
    substring(stamp, 1, 10) AS dt
  -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、substrが利用できる
  -- substr(stamp, 1, 10) AS dt
  , COUNT(1) AS search_count
  , SUM(CASE WHEN next_action = 'search' THEN 1 ELSE 0 END) AS retry_count
  , AVG(CASE WHEN next_action = 'search' THEN 1.0 ELSE 0.0 END) AS retry_rate
FROM
  access_log_with_next_action
WHERE
  action = 'search'
GROUP BY
  -- ■ PostgreSQL, Redshift, BigQueryの場合、
  --   SELECT句で定義した別名をGROUP BYに指定できる
  dt
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、
  --   SELECT句で別名を指定する前の式をGROUP BYに指定できる
  -- substring(stamp, 1, 10)
ORDER BY
```

```
  dt
;
```

```
     dt     | search_count | retry_count | retry_rate
------------+--------------+-------------+------------
 2017-01-05 |         3510 |         951 |     0.2709
 2017-01-06 |         3541 |         891 |     0.2516
```

再検索ワードを集計する

再検索ワードを集計することで、シノニム辞書の行き届いていない範囲が明確になります。また、コンテンツ名称の揺らぎや通称を見つけ出すきっかけにもなります。

例えば、映画のタイトルを想定すると分かりやすいかもしれません。仮に映画の続編に「マイナビクエスト２」と呼ばれているタイトルがあったとします。しかしながら、実は「マイナビクエスト データの覚醒」が正式名称である場合、シノニム辞書を設定していないと「マイナビクエスト２」ではヒットしません。再検索で正式名称を入力するユーザーもいるかもしれませんが、それ以前に離脱してしまうケースがほとんどでしょう。ユーザーは検索ワードが間違っていることに気付かないものです。ヒットさせるためには、「マイナビクエスト２」をシノニム辞書に追加すべきです。

したがって、ユーザーがどのような検索ワードから再検索に至ったのかを集計して、対応可能なものがあれば、シノニム辞書へ反映させましょう。下記に、再検索ワードを集計するクエリ例を示します。

コード 8.1.2.3：再検索ワードを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_search AS (
  SELECT
      stamp
    , session
    , action
    , keyword
    , result_num
    , LEAD(action)
        -- ■ PostgreSQL, Hive, Redshift, BigQueryの場合は下記
        OVER(PARTITION BY session ORDER BY stamp ASC)
        -- ■ SparkSQLの場合、フレーム指定が必要
        -- OVER(PARTITION BY session ORDER BY stamp ASC
        --   ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
      AS next_action
    , LEAD(keyword)
        -- ■ PostgreSQL, Hive, Redshift, BigQueryの場合は下記
        OVER(PARTITION BY session ORDER BY stamp ASC)
        -- ■ SparkSQLの場合、フレーム指定が必要
```

```
        -- OVER(PARTITION BY session ORDER BY stamp ASC
        --   ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
      AS next_keyword
    , LEAD(result_num)
        -- ■ PostgreSQL, Hive, Redshift, BigQueryの場合は下記
        OVER(PARTITION BY session ORDER BY stamp ASC)
        -- ■ SparkSQLの場合、フレーム指定が必要
        -- OVER(PARTITION BY session ORDER BY stamp ASC
        --   ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
      AS next_result_num
  FROM
    access_log
)
SELECT
    keyword
  , result_num
  , COUNT(1) AS retry_count
  , next_keyword
  , next_result_num
FROM
  access_log_with_next_search
WHERE
      action = 'search'
  AND next_action = 'search'
GROUP BY
  keyword, result_num, next_keyword, next_result_num
;
```

```
    keyword    | result_num | retry_count |    next_keyword   | next_result_num
---------------+------------+-------------+-------------------+----------------
 hive          |        730 |           7 | hive lateral view |              20
 spark         |        646 |          36 | spark sql         |             179
 pstgresql     |          0 |          13 | postgresql        |             889
 postgresql 10 |          0 |           5 | postgresql        |             889
 postgresql 10 |          0 |          20 | postgresql 9      |              64
...
```

ワンポイント

シノニム辞書を人力で管理することは、その膨大なデータ量から困難であったり、担当者では気付かない揺らぎも存在する可能性があります。再検索ワードを集計して、検索システムを継続的に改善する仕組みを検討しましょう。

関連項目 | 3-3-2 グループの中での順序を扱う

8-1-3 再検索ワードを分類して集計する

SQL LIKE演算子　　分析 再検索ワード

前項では、離脱せずに再検索を実行するユーザーの行動には、検索を改善する大きなヒントがあると説明しましたが、ユーザーが再検索を実行するケースは、検索結果に満足せずに何らかのモチベーションを持っていることを意味しています。どのような状況やモチベーションで再検索を実行するのか、ユーザーの行動にはパターンがあります

1 **【NoMatchからの条件変更】**
　検索結果が0件だったから、別のワードで検索した
2 **【検索の絞り込み】**
　検索結果が多すぎるので、別のワードを加えて絞り込んだ
3 **【検索ワードの変更】**
　検索にヒットはしているが、別のワードで検索した

上記のパターンで、どのようなワードから再検索を実行しているか明らかになれば、シノニム辞書やユーザー辞書に追加すべき項目を検討することが可能となり、検索の改善に繋がります。再検索ワードを上記のパターン別に集計する方法を紹介しましょう。

NoMatchからの条件変更

NoMatchから条件を変更している検索ワードを集計することで、シノニム辞書やユーザー辞書に追加すべきワードの候補が明らかになります。下記コード例に、NoMatchからの再検索ワードを集計するクエリを示します。

コード8.1.3.1：NoMatchからの再検索ワードを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_search AS (
  -- コード8.1.2.3参照
)
SELECT
    keyword
  , result_num
  , COUNT(1) AS retry_count
  , next_keyword
  , next_result_num
FROM
  access_log_with_next_search
WHERE
```

```
      action = 'search'
  AND next_action = 'search'
  -- NoMatchのログに絞り込む
  AND result_num = 0
GROUP BY
  keyword, result_num, next_keyword, next_result_num
;
```

```
  keyword   | result_num | retry_count | next_keyword | next_result_num
------------+------------+-------------+--------------+-----------------
 pstgresql  |          0 |          13 | postgresql   |             889
...
```

検索の絞り込み

再検索された検索ワードで元の検索ワードを含むものは、「検索の絞り込み」を行う再検索と判断できます。元の検索ワードから、頻繁に絞り込みが行われるワードに関しては、より上位に表示されるようにソート順を調整したり、予め絞り込みの検索候補を表示することで、ユーザーが求めるコンテンツに辿り着く手助けが可能です。

コード8.1.3.2：検索の絞り込み再検索ワードを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_search AS (
  -- コード8.1.2.3参照
)
SELECT
    keyword
  , result_num
  , COUNT(1) AS retry_count
  , next_keyword
  , next_result_num
FROM
  access_log_with_next_search
WHERE
      action = 'search'
  AND next_action = 'search'
  -- 元のワードを含む検索に絞り込む
  -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、concat関数が利用できる
  AND next_keyword LIKE concat('%', keyword, '%')
  -- ■ PostgreSQL, Redshiftの場合、|| 演算子が利用できる
  -- AND next_keyword LIKE '%' || keyword || '%'
GROUP BY
  keyword, result_num, next_keyword, next_result_num
;
```

```
   keyword   | result_num | retry_count |     next_keyword     | next_result_num
-------------+------------+-------------+----------------------+-----------------
 fab         |        646 |           1 | fabulous             |             179
 yamada taro |        367 |           1 | yamada taro football |             105
 english     |        343 |           1 | history of english   |             757
```

検索ワードの変更

まったく異なる検索ワードで再検索が実行されているケースは、元の検索ワードでの結果が少なく、シノニム辞書が行き届いていないと考えられます。また、商品名の揺らぎや通称、略称の気付きとなる場合もあります。

例えば、女性歌手は結婚後に名字が変わり、新しい名前で活動するケースもあります。こうした場合、旧姓でも商品はヒットしますが、欲しい商品がないと改姓後の名前で検索をします。これは限定的な例ですが、まったく違う検索ワードで検索するには何らかの理由があるはずです。集計することで、その糸口を探してみましょう。下記コード例に、異なる検索ワードでの再検索を集計するクエリを示します。

コード8.1.3.3：異なる検索ワードでの再検索を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_search AS (
  -- コード8.1.2.3参照
)
SELECT
    keyword
  , result_num
  , COUNT(1) AS retry_count
  , next_keyword
  , next_result_num
FROM
  access_log_with_next_search
WHERE
      action = 'search'
  AND next_action = 'search'
  -- 元のワードを含まない検索に絞り込む
  -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合、concat関数が利用できる
  AND next_keyword NOT LIKE concat('%', keyword, '%')
  -- ■ PostgreSQL, Redshiftの場合、|| 演算子が利用できる
  -- AND next_keyword NOT LIKE '%' || keyword || '%'
GROUP BY
  keyword, result_num, next_keyword, next_result_num
;
```

```
    keyword      | result_num | retry_count |   next_keyword | next_result_num
-----------------+------------+-------------+----------------+-----------------
 WITH clause     |        125 |           8 | CTE            |             312
 Window function |        562 |          13 | OLAP function  |             128
...
```

まとめ

本項で紹介した3つの方法で再検索ワードを集計すると、それぞれまったく異なる気付きが得られるはずです。それぞれ集計してみましょう。

8-1-4 検索離脱率とそのワードを集計する

SQL SUM(CASE～)、AVG(CASE～)　　分析 検索離脱率、検索離脱ワード

検索結果の表示後に、アクションを行っていない（離脱した）ユーザーは、その検索結果に対して満足していないと考えられます。検索結果の画面からユーザーが離脱する理由として、検索結果が0件、求めている検索結果ではなかったなどの理由が考えられます。検索しながらも離脱するユーザーがどれほど存在するのか集計して、改善の糸口がないか探りましょう。

検索離脱数の集計方法

前述の「8-1-2 再検索率とそのワードを集計する」(P.373) で紹介したコード8.1.2.1の出力結果で、actionがsearchでnext_actionがNULLのものを検索離脱として、このレコード数を検索離脱数とします。また、actionがsearchのレコードを検索総数とします。検索総数に占める検索離脱数の割合を「検索離脱率」と定義します。

この検索離脱率の定義に従い、検索離脱率を集計するクエリを下記コード例に示します。

コード8.1.2.1の出力結果(再掲)

```
        stamp        | session | action | next_action
---------------------+---------+--------+-------------
 2017-01-05 23:35:13 | 0CVKaz  | search | search
 2017-01-05 23:36:08 | 0CVKaz  | search | detail
 2017-01-05 23:37:17 | 0CVKaz  | detail | detail
 2017-01-05 23:38:38 | 0CVKaz  | detail | search
 2017-01-05 23:40:10 | 0CVKaz  | search | detail
```

```
 2017-01-05 23:41:43 | 0CVKaz  | detail | search
 2017-01-05 23:43:10 | 0CVKaz  | search |         --> 検索離脱
 2017-01-05 23:34:57 | 1QceiB  | search | search
 2017-01-05 23:35:37 | 1QceiB  | search | detail
 2017-01-05 23:36:48 | 1QceiB  | detail | detail
 2017-01-05 23:37:27 | 1QceiB  | detail | search
 2017-01-05 23:38:52 | 1QceiB  | search | detail
 2017-01-05 23:39:50 | 1QceiB  | detail | search
...
```

コード8.1.4.1：検索離脱率を集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_action AS (
  -- コード8.1.2.1参照
)
SELECT
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合.substringで日付部分を抽出する
    substring(stamp, 1, 10) AS dt
  -- ■ PostgreSQL, Hive, BigQuery, SparkSQLの場合.substrが利用できる
  -- substr(stamp, 1, 10) AS dt
  , COUNT(1) AS search_count
  , SUM(CASE WHEN next_action IS NULL THEN 1 ELSE 0 END) AS exit_count
  , AVG(CASE WHEN next_action IS NULL THEN 1.0 ELSE 0.0 END) AS exit_rate
FROM
  access_log_with_next_action
WHERE
  action = 'search'
GROUP BY
  -- ■ PostgreSQL, Redshift, BigQueryの場合、
  --   SELECT句で定義した別名をGROUP BYに指定できる
  dt
  -- ■ PostgreSQL, Hive, Redshift, SparkSQLの場合、
  --   SELECT句で別名を指定する前の式をGROUP BYに指定できる
  -- substring(stamp, 1, 10)
ORDER BY
  dt
;
```

```
     dt     | search_count | exit_count |  exit_rate
------------+--------------+------------+------------
 2017-01-05 |         3510 |       1324 |     0.3772
 2017-01-06 |         3541 |       1247 |     0.3521
```

検索離脱ワードを集計する

続いて、検索離脱となった際の検索ワードを抽出するクエリをコード8.1.4.2に示します。
検索離脱となった検索において、どのようなワードが離脱に繋がったかを調べ、シノニム辞書、ユーザー辞書の追加として適切なものがあれば、反映させましょう。

コード8.1.4.2：検索離脱ワードを集計するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_search AS (
  -- コード8.1.2.3参照
)
SELECT
    keyword
  , COUNT(1) AS search_count
  , SUM(CASE WHEN next_action IS NULL THEN 1 ELSE 0 END) AS exit_count
  , AVG(CASE WHEN next_action IS NULL THEN 1.0 ELSE 0.0 END) AS exit_rate
  , result_num
FROM
  access_log_with_next_search
WHERE
  action = 'search'
GROUP BY
  keyword, result_num
  -- キーワード全体の離脱率を計算したあと、離脱率が0より大きいキーワードに絞り込む
HAVING
  SUM(CASE WHEN next_action IS NULL THEN 1 ELSE 0 END) > 0
;
```

```
  keyword   | search_count | exit_count | exit_rate | result_num
------------+--------------+------------+-----------+------------
 hive       |          212 |         23 |    0.1084 |        730
 spark sql  |          123 |         12 |    0.0975 |        179
 postgresql |          351 |         34 |    0.0968 |        889
 redshift   |          298 |         31 |    0.1040 |        821
...
```

ワンポイント

検索から離脱したユーザーが検索結果に満足しなかった理由として、欲しい商品が上位に表示されないなど、表示順が関係するケースもあります。なお、並び順の評価指標に関しては、「8-1-6 検索結果の網羅性を指標化する」(P.384) 以降で後述します。

8-1-5 検索ワードに関する指標の集計を効率化する

前項まで、NoMatch率と再検索率、検索離脱率の指標を紹介しましたが、各指標と検索ワードを算出するために、毎回類似したクエリを作成するのは非効率です。下記コード例に、集計を効率化するため中間データを作成するSQLを紹介します。

コード8.1.5.1：検索に関する指標を集計しやすくする中間データを作成するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
access_log_with_next_search AS (
  -- コード8.1.2.3参照
)
, search_log_with_next_action (
  SELECT *
  FROM
    access_log_with_next_search
  WHERE
    action = 'search'
)
SELECT *
FROM search_log_with_next_action
ORDER BY
  session, stamp
;
```

```
 session | action |  keyword   | result_num | next_action | next_keyword | next_result_num
---------+--------+------------+------------+-------------+--------------+----------------
 0CVKaz  | search | pstgresql  |          0 | search      | postgresql   |             889
 0CVKaz  | search | postgresql |        889 | detail      | NULL         |
 0CVKaz  | search | postgresql |        889 | detail      | NULL         |
 0CVKaz  | search | hive       |        730 |             |              |
 1QceiB  | search | spark      |        646 | search      | spark sql    |             179
 1QceiB  | search | spark sql  |        179 | detail      | NULL         |
...
```

ワンポイント

上記の出力結果があれば、NoMatch数や再検索数、検索離脱数、またそのワードの集計は容易になります。この出力結果をテーブルとして保存するか、検索に関する指標を集計する前処理として、定型化したWITH句を準備しておくことで、作業効率が格段に上がります。

関連項目	9-1-4 データを活用しやすい状態に整える

8-1-6 検索結果の網羅性を指標化する

SQL FULL OUTER JOIN、SUMウィンドウ関数

分析 再現率

前項までは、ユーザーの検索ログを用いて検索機能の利用を評価しましたが、本項以降では、検索ワードに対する個別アイテムの評価を人力で行ない、定量的な指標を用いて検索エンジンそのものの精度を評価する手法を紹介します。

本項で用いるサンプルデータは、下記データ例に示す、検索ワードに対する検索結果のランキング（データ8.1.6.1）と検索ワードに対する正解アイテム（データ8.1.6.2）です。なお、正解アイテムとは、ある検索ワードに対してどのアイテムがヒットして欲しいかを予め列挙したデータです。

データ8.1.6.1：検索キーワードに対する検索結果のランキング（search_result）テーブル

```
 keyword  | rank |  item
----------+------+---------
 sql      |    1 | book001
 sql      |    2 | book005
 sql      |    3 | book012
 sql      |    4 | book004
 sql      |    5 | book003
 sql      |    6 | book010
 sql      |    7 | book024
 sql      |    8 | book025
 sql      |    9 | book050
 sql      |   10 | book100
 postgres |    1 | book002
 postgres |    2 | book004
 postgres |    3 | book012
 postgres |    4 | book008
 postgres |    5 | book003
 postgres |    6 | book010
 postgres |    7 | book035
 postgres |    8 | book040
 postgres |    9 | book077
 postgres |   10 | book100
 hive     |    1 | book200
```

データ8.1.6.2：検索キーワードに対する正解アイテム（correct_result）テーブル

```
 keyword  |  item
----------+---------
 sql      | book003
 sql      | book005
 sql      | book008
 sql      | book010
```

```
 sql      | book025
 sql      | book100
 postgres | book008
 postgres | book010
 postgres | book030
 postgres | book055
 postgres | book066
 postgres | book100
 hive     | book200
 redshift | book300
```

再現率(Recall)を用いて検索の網羅性を評価する

検索エンジンを定量的に評価するための代表的な指標に、再現率（Recall）と適合率（Precision）があります。前者の再現率とは、あるキーワードに対して用意された正解アイテムのうち、何割のアイテムを検索結果として返すことができたかを意味します。例えば、あるキーワードに対する正解アイテムが10件あり、検索結果内に4件の正解アイテムが含まれているケースの再現率は40%となります。

再現率を計算するために、まずは検索結果と正解アイテムを結合し、どのアイテムが正解アイテムに含まれているかを判定するクエリを、下記のコード例に示します。

コード8.1.6.1：検索結果と正解アイテムを結合するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
search_result_with_correct_items AS (
  SELECT
      COALESCE(r.keyword, c.keyword) AS keyword
    , r.rank
    , COALESCE(r.item, c.item) AS item
    , CASE WHEN c.item IS NOT NULL THEN 1 ELSE 0 END AS correct
  FROM
      search_result AS r
    FULL OUTER JOIN
      correct_result AS c
      ON r.keyword = c.keyword
      AND r.item = c.item
)
SELECT *
FROM
  search_result_with_correct_items
ORDER BY
  keyword, rank
;
```

```
 keyword | rank |  item   | correct
---------+------+---------+---------
 sql     |    1 | book001 |       0
 sql     |    2 | book005 |       1
 sql     |    3 | book012 |       0
 sql     |    4 | book004 |       0
 sql     |    5 | book003 |       1
 sql     |    6 | book010 |       1
 sql     |    7 | book024 |       0
 sql     |    8 | book025 |       1
 sql     |    9 | book050 |       0
 sql     |   10 | book100 |       1
 sql     |      | book008 |       1
...
```

上記の出力結果で、correctカラムのフラグが1であるアイテムが、正解アイテムに含まれているアイテムです。ここで、OUTER JOINを用いて検索結果に含まれていない正解アイテムのレコードを残しているのは、再現率を計算するために正解アイテムの総数を求める必要があるためです。

続いて、下記コード例に、検索結果の上位n件における再現率を計算するクエリを示します。

上位n件における正解アイテムのヒット数は、SUMウィンドウ関数を利用して、累計の正解アイテムのフラグを合計することで求めます。この値を正解アイテムの総数で割ることで、再現率を算出します。なお、検索結果に含まれないアイテムのレコードに関しては再現率を計算できませんが、本項では便宜上再現率を0として表示しています。

コード8.1.6.2：検索結果の上位n件における再現率を計算するクエリ

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
, search_result_with_recall AS (
  SELECT
      *
    -- 検索結果の上位から、正解データに含まれるアイテムの数を累計する
    , SUM(correct)
        -- rankがNULLの場合は、ソート順で最後に位置させたいため、便宜上、十分大きな値に変換する
        OVER(PARTITION BY keyword ORDER BY COALESCE(rank, 100000) ASC
          ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cum_correct
    , CASE
        -- 検索結果に含まれないアイテムのレコードは、便宜上再現率を0とする
        WHEN rank IS NULL THEN 0.0
        ELSE
          100.0
          * SUM(correct)
```

```
            OVER(PARTITION BY keyword ORDER BY COALESCE(rank, 100000) ASC
              ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
          / SUM(correct) OVER(PARTITION BY keyword)
      END AS recall
  FROM
    search_result_with_correct_items
)
SELECT *
FROM
  search_result_with_recall
ORDER BY
  keyword, rank
;
```

```
 keyword  | rank |  item   | correct | cum_correct |   recall
----------+------+---------+---------+-------------+----------
 hive     |    1 | book200 |       1 |           1 |   100.00
 postgres |    1 | book002 |       0 |           0 |     0.00
 postgres |    2 | book004 |       0 |           0 |     0.00
 postgres |    3 | book012 |       0 |           0 |     0.00
 postgres |    4 | book008 |       1 |           1 |    16.66
 postgres |    5 | book003 |       0 |           1 |    16.66
 postgres |    6 | book010 |       1 |           2 |    33.33
 postgres |    7 | book035 |       0 |           2 |    33.33
 postgres |    8 | book040 |       0 |           2 |    33.33
 postgres |    9 | book077 |       0 |           2 |    33.33
 postgres |   10 | book100 |       1 |           3 |    50.00
 postgres |      | book030 |       1 |           6 |     0.00
 postgres |      | book055 |       1 |           6 |     0.00
 postgres |      | book066 |       1 |           6 |     0.00
 redshift |      | book300 |       1 |           1 |     0.00
 sql      |    1 | book001 |       0 |           0 |     0.00
 sql      |    2 | book005 |       1 |           1 |    16.66
 sql      |    3 | book012 |       0 |           1 |    16.66
 sql      |    4 | book004 |       0 |           1 |    16.66
 sql      |    5 | book003 |       1 |           2 |    33.33
 sql      |    6 | book010 |       1 |           3 |    50.00
 sql      |    7 | book024 |       0 |           3 |    50.00
 sql      |    8 | book025 |       1 |           4 |    66.66
 sql      |    9 | book050 |       0 |           4 |    66.66
 sql      |   10 | book100 |       1 |           5 |    83.33
 sql      |      | book008 |       1 |           6 |     0.00
```

再現率の値を集約して比較しやすくする

前述のコード例で算出した再現率は、検索結果のすべてのレコードに対して再現率を求めていますが、これらの値を同時にすべて眺めて、検索エンジンを評価することは困難です。検索エンジンを定量的に評価して、複数の検索エンジンを比較するには、1つの代表的な値に集約することが望ましいです。

検索エンジンの一般的なインターフェースでは、検索結果の上位n件がランキング形式で表示されることがほとんどです。したがって、各検索ワードにおける再現率の値は、検索結果上でユーザーに表示されるアイテムの件数に対する値を用いるのが好ましいといえます。

下記コード例では、検索結果の表示結果がデフォルトで5件と仮定して、キーワードごとの再現率を計算します。検索結果が5件以上の場合は、上位5件の場合の再現率、検索結果が5件未満の場合は、検索結果全体に対する再現率、検索結果が存在しない場合は再現率0の値を取得します。

コード8.1.6.3：検索結果上位5件の再現率をキーワードごとに取得するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
, search_result_with_recall AS (
  -- コード8.1.6.2参照
)
, recall_over_rank_5 AS (
  SELECT
      keyword
    , rank
    , recall
      -- 検索結果の順位が大きい順に番号を振る
      -- 検索結果に出ていないアイテムの順位は、便宜上0として扱う
    , ROW_NUMBER()
        OVER(PARTITION BY keyword ORDER BY COALESCE(rank, 0) DESC)
      AS desc_number
  FROM
    search_result_with_recall
  WHERE
    -- 検索結果の順位が5以下または検索結果に含まれないアイテムのみを表示
    COALESCE(rank, 0) <= 5
)
SELECT
    keyword
  , recall AS recall_at_5
FROM recall_over_rank_5
-- 検索結果上位5件のうち、最も順位が大きいレコードに絞る
WHERE desc_number = 1
;
```

```
  keyword  | recall_at_5
----------+-------------
 hive     |       100.00
 postgres |        16.66
 sql      |        33.33
 redshift |         0.00
```

検索ワードごとの再現率が1つの値に集約できたら、最後にそれらの再現率を平均して、検索エンジン全体の平均再現率を計算しましょう。下記コード例に、検索エンジン全体の平均再現率を計算するクエリを示します。

コード8.1.6.4：検索エンジン全体の平均再現率を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
, search_result_with_recall AS (
  -- コード8.1.6.2参照
)
, recall_over_rank_5 AS (
  -- コード8.1.6.3参照
)
SELECT
  AVG(recall) AS average_recall_at_5
FROM recall_over_rank_5
-- 検索結果上位5件のうち、最も順位が大きいレコードに絞る
WHERE desc_number = 1
;
```

```
 average_recall_at_5
---------------------
               37.50
```

まとめ

再現率は、正解アイテムに含まれるアイテムをどれだけ網羅的に取得できたかを示す指標です。この指標は、医学関連や法律関連など必要な情報を漏れなく検索したい場合などで重視される指標です。

関連項目	3-3-2 グループの中での順序を扱う

8-1-7 検索結果の妥当性を指標化する

SQL FULL OUTER JOIN、SUMウィンドウ関数

分析 適合率

前項で説明した再現率と同じく、検索結果の評価に用いられる指標が適合率（Precision）です。この適合率は単に精度とも呼ばれることがあります。適合率の値は、検索結果に含まれるアイテムの何割が正解アイテムであるかを表します。例えば、検索結果上位10件に5件の正解アイテムが含まれている場合、その適合率は50%となります。

適合率（Precision）を用いて検索の妥当性を評価する

下記コード例に、検索結果の上位n件における適合率を計算するクエリを示します。基本的な考え方は、前項で説明した再現率と同じで、分母の計算のみが検索結果順位までの累計アイテム数に変わっています。検索結果に含まれないアイテムのレコードも同様に、適合率0として計算しています。

コード8.1.7.1：検索結果の上位n件における適合率を計算するクエリ

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
, search_result_with_precision AS (
  SELECT
      *
    -- 検索結果の上位から、正解データに含まれるアイテムの数を累計する
    , SUM(correct)
        -- rankがNULLの場合は、ソート順で最後に位置させたいため、便宜上、十分大きな値に変換する
        OVER(PARTITION BY keyword ORDER BY COALESCE(rank, 100000) ASC
          ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cum_correct
    , CASE
        -- 検索結果に含まれないアイテムのレコードは、便宜上適合率を0とする
        WHEN rank IS NULL THEN 0.0
        ELSE
          100.0
          * SUM(correct)
             OVER(PARTITION BY keyword ORDER BY COALESCE(rank, 100000) ASC
               ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
          -- 再現率と比べて、分母がその検索結果順位までの累計アイテム数になる
          / COUNT(1)
              OVER(PARTITION BY keyword ORDER BY COALESCE(rank, 100000) ASC
                ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      END AS precision
  FROM
    search_result_with_correct_items
```

```
)
SELECT *
FROM
  search_result_with_precision
ORDER BY
  keyword, rank
;
```

```
 keyword   | rank |  item   | correct | cum_correct | precision
-----------+------+---------+---------+-------------+-----------
 hive      |    1 | book200 |       1 |           1 |    100.00
 postgres  |    1 | book002 |       0 |           0 |      0.00
 postgres  |    2 | book004 |       0 |           0 |      0.00
 postgres  |    3 | book012 |       0 |           0 |      0.00
 postgres  |    4 | book008 |       1 |           1 |     25.00
 postgres  |    5 | book003 |       0 |           1 |     20.00
 postgres  |    6 | book010 |       1 |           2 |     33.33
 postgres  |    7 | book035 |       0 |           2 |     28.57
 postgres  |    8 | book040 |       0 |           2 |     25.00
 postgres  |    9 | book077 |       0 |           2 |     22.22
 postgres  |   10 | book100 |       1 |           3 |     30.00
 postgres  |      | book030 |       1 |           6 |      0.00
 postgres  |      | book055 |       1 |           6 |      0.00
 postgres  |      | book066 |       1 |           6 |      0.00
 redshift  |      | book300 |       1 |           1 |      0.00
 sql       |    1 | book001 |       0 |           0 |      0.00
 sql       |    2 | book005 |       1 |           1 |     50.00
 sql       |    3 | book012 |       0 |           1 |     33.33
 sql       |    4 | book004 |       0 |           1 |     25.00
 sql       |    5 | book003 |       1 |           2 |     40.00
 sql       |    6 | book010 |       1 |           3 |     50.00
 sql       |    7 | book024 |       0 |           3 |     42.85
 sql       |    8 | book025 |       1 |           4 |     50.00
 sql       |    9 | book050 |       0 |           4 |     44.44
 sql       |   10 | book100 |       1 |           5 |     50.00
 sql       |      | book008 |       1 |           6 |      0.00
```

適合率の値を集約して比較を容易にする

適合率に関しても、再現率と同様に1つの代表的な値に集約しましょう。下記コード例に、検索結果上位5件の適合率を検索ワードごとに取得するクエリを示します。このコード例も再現率の計算に用いたクエリとほとんど同じです。

なお、検索結果上位n件における適合率は、「P@n」と記述されることもあります。

コード8.1.7.2：検索結果上位5件の適合率をキーワードごとに取得するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
, search_result_with_precision AS (
  -- コード8.1.7.1参照
)
, precision_over_rank_5 AS (
  SELECT
      keyword
    , rank
    , precision
      -- 検索結果の順位が大きい順に番号を振る
      -- 検索結果に出ていないアイテムの順位は、便宜上0として扱う
    , ROW_NUMBER()
        OVER(PARTITION BY keyword ORDER BY COALESCE(rank, 0) DESC)
      AS desc_number
  FROM
    search_result_with_precision
  WHERE
    -- 検索結果の順位が5以下または検索結果に含まれないアイテムのみを表示
    COALESCE(rank, 0) <= 5
)
SELECT
    keyword
  , precision AS precision_at_5
FROM precision_over_rank_5
-- 検索結果上位5件のうち、最も順位が大きいレコードに絞る
WHERE desc_number = 1;
```

```
 keyword  | precision_at_5
----------+----------------
 postgres |          20.00
 sql      |          40.00
 hive     |         100.00
 redshift |           0.00
```

最後に、検索ワードごとの適合率を取得できたら、最後に適合率を平均して、検索エンジン全体の平均適合率を計算します。下記コード例に平均適合率を計算するクエリを示します。

コード8.1.7.3：検索エンジン全体の平均適合率を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
```

```
, search_result_with_precision AS (
  -- コード8.1.7.1参照
)
, precision_over_rank_5 AS (
  -- コード8.1.7.2参照
)
SELECT
    AVG(precision) AS average_precision_at_5
FROM precision_over_rank_5
-- 検索結果上位5件のうち、最も順位が大きいレコードに絞る
WHERE desc_number = 1
;
```



```
 average_precision_at_5
------------------------
                 40.00
```

まとめ

適合率は、検索結果の上位に表示されるアイテムの妥当性を表す指標です。この指標は、Web検索など、膨大な検索対象から妥当な検索結果にいち早く辿り着きたいケースで重視される指標です。

関連項目	3-3-2 グループの中での順序を扱う

8-1-8 検索結果の順位を考慮した指標を計算する

SQL AVG関数　　分析 MAP(Mean Average Precision)

本節では、検索エンジンを評価するための最も基本的な指標、再現率と適合率を紹介しています。この2つの指標はシンプルで使い勝手が良いものですが、現実の検索エンジンを評価したり、より複雑な条件を検討したいケースでは、いくつか不十分な点があります。

以下に、再現率と適合率の主な課題を挙げます。

1. 検索結果におけるランキングの順位が考慮されない
2. 正解と不正解のアイテムを0と1の2値でしか表現できない
3. そもそもすべてのアイテムに対して、予め正解アイテムを用意できない

1 ランキング順位が考慮されない件ですが、例えば、検索結果上位10件の適合率が40%である検索エンジンAとBがあり、検索エンジンAは1件目～4件目が正解アイテムであるのに対し、検索エンジンBは7件目～10件目が正解アイテムであるとします。このケースでは、より上位に正解アイテムが位置する検索エンジンAを高く評価すべきですが、P@10値を比較すると、両者の評価は互角になります。

検索結果の順位を考慮した指標には、MAP（Mean Average Precision）やMRR（Mean Reciprocal Rank）などがあります。

2 正解と不正解の択一的にしか表現できない件ですが、前項までで取り扱った正解アイテムは、どのアイテムが検索にヒットしても対等に扱われますが、場合によっては「とても関連があるアイテム」「少しだけ関連があるアイテム」など、段階的なスコアで正解アイテムを扱いたいケースがあります。
例えば、ユーザーからの5段階評価のレビューデータが既に存在する場合、そのスコアを正解データとして扱うには、適合率や再現率を適用すると、せっかくユーザーに入力してもらった5段階評価のスコアを活用しきれません。

段階的なスコアを考慮して正解アイテムを扱う指標には、DCG（Discounted Cumulated Gain）やNDCG（Normalized DCG）が有名です。

3 全アイテムに対して正解アイテムを用意できない件は、より現実の検索エンジン評価に即した課題です。そもそも検索エンジンが必要となるサービスでは、検索対象となるアイテムが数十万件、場合によっては数千万件以上のデータ量となるケースもあります。こうした大量アイテムに対して、すべてのアイテムとキーワードの組み合わせに対して、「正解」「不正解」を人力でラベル付けすることは、実質不可能です。
ユーザーの行動ログなどから機械的に正解アイテムを作成することも考えられますが、サービスを利用中のユーザーが単独で確認できるアイテム数は、せいぜい数百件が限度でしょう。再現率や適合率を適用しても極端に低い値となり、参考にならない場合が多々あります。

検索対象のアイテム数に対して、正解アイテム数が限定されている場合に使用できる指標としては、BPREF（Binary Preferece）があります。

MAPを用いて検索結果の順位を考慮した評価を行う

まずは、検索結果の順位を考慮して評価するために、MAP（Mean Average Precision）と呼ばれる指標を実際に計算してみましょう。

MAPの定義は、「それぞれの正解アイテムがランキング中に初めて登場した順位における適合率の平均」です。ただし、正解アイテムがランキングに登場しない場合は、便宜上適合率を0として扱います。

MAPを用いると、例えば、正解アイテム数が4件かつP@10=40%の検索結果で、上位1件目～4件目が正解アイテムの場合、MAPの値は100×((1／1)＋(2／2)＋(3／3)＋(4／4))／4＝100となります。

一方、上位7件目～10件目が正解アイテムの場合、同じくP@10は40%ですが、MAPの値は100×(1／7)＋(2／8)＋(3／9)＋(4／10))／4＝28.15…となり、前者より低く評価されます。

MAPを計算するために、まずは正解アイテムごとの適合率を抽出します。前項までのクエリを活用すれば、correctカラムのフラグが1であるレコードを抽出するだけです。下記のコード例に、正解アイテムごとの適合率を抽出するクエリを示します。

コード8.1.8.1：正解アイテムごとの適合率を抽出するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
, search_result_with_precision AS (
  -- コード8.1.7.1参照
)
SELECT
    keyword
  , rank
  , precision
FROM
  search_result_with_precision
WHERE
  correct = 1
;
```

```
 keyword  | rank | precision
----------+------+-----------
 hive     |    1 |    100.00
 postgres |    4 |     25.00
 postgres |    6 |     33.33
 postgres |   10 |     30.00
 postgres |      |      0.00
 postgres |      |      0.00
 postgres |      |      0.00
 redshift |      |      0.00
 sql      |    2 |     50.00
 sql      |    5 |     40.00
 sql      |    6 |     50.00
 sql      |    8 |     50.00
 sql      |   10 |     50.00
 sql      |      |      0.00
```

続いて、検索ワードごとに適合率の平均を計算します。

コード8.1.8.2：検索キーワードごとに適合率の平均を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
, search_result_with_precision AS (
  -- コード8.1.7.1参照
)
, average_precision_for_keywords AS (
  SELECT
      keyword
    , AVG(precision) AS average_precision
  FROM
    search_result_with_precision
  WHERE
    correct = 1
  GROUP BY
    keyword
)
SELECT *
FROM
  average_precision_for_keywords
;
```

```
 keyword  | average_precision
----------+-------------------
 hive     |            100.00
 redshift |              0.00
 sql      |             40.00
 postgres |             14.72
```

最後に、検索ワードごとの適合率の平均を取得し、検索エンジン自体のMAPを計算します。下記コード例に検索エンジンのMAPを計算するクエリを示します。

コード8.1.8.3：検索エンジンのMAPを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
search_result_with_correct_items AS (
  -- コード8.1.6.1参照
)
, search_result_with_precision AS (
  -- コード8.1.7.1参照
)
```

```
, average_precision_for_keywords AS (
  -- コード8.1.8.2参照
)
SELECT
  AVG(average_precision) AS mean_average_precision
FROM
  average_precision_for_keywords
;
```

```
 mean_average_precision
------------------------
                  38.68
```

検索評価におけるその他の指標を知る

本項では、代表的な検索指標としてMAPを計算するクエリを紹介しましたが、よく使用されるランキング指標を、下表に紹介します。

表8.1.8.1：検索評価に用いられる代表的なランキング指標

指標名	説明
Precision（適合率）	検索結果のランキング中、何割のアイテムが正しく表示されたかを示す。Web検索のような、膨大で重複を含む情報から関連する情報を素早く獲得したい場面で重視される。 $Precision = p = \frac{\lvert R \cap A \rvert}{\lvert A \rvert}$
Recall（再現率）	ユーザーが欲しいと思うアイテム中、何割のアイテムが正しく表示されたかを示す。法律系や医療系を対象とした検索など、結果に漏れがあると困る場面で重視される。 $Recall = r = \frac{\lvert R \cap A \rvert}{\lvert R \rvert}$
P@n	ランキングの上位n件におけるPrecision(適合率)。
MAP (Mean Average Precision)	それぞれのクエリに対するAverage Precisionの平均値。 $MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q}$
MRR (Mean Reciprocal Rank) (平均逆順位)	ランキング中、初めて正解のアイテムが出た順位の逆数の平均。 $MRR = \frac{1}{\lvert Q \rvert} \sum_{i=1}^{\lvert Q \rvert} \frac{1}{rank_i}$
E-Measure	PrecisionとRecallを重み付きで組み合わせた単一指標。 PrecisionとRecallの重みを1：1にするとF-Measureになる。 $F_\beta = \frac{(1+\beta^2) \cdot (p \cdot r)}{(\beta^2 \cdot p + r)}$

F-Measure	PrecisionとRecallの調和平均 $F_1 = 2 \cdot \frac{p \cdot r}{p + r}$
NDCG （Normalized Discounted Cumulated Gain）	アイテムの評価を複数レベルで設定し、ランキング中の順位を加味した指標。正解データを作るのが大変だが、信頼度は高い。 $DCG_k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$ $nDCG_k = \frac{DCG_k}{idealDCG_k}$
BPREF （Binary Preferences）	アイテム間の相対的な関心度を基準にした指標。アイテム空間が非常に巨大なケースで、完全な評価データの作成が困難な場合に有効。 $bpref = \frac{1}{R} \sum_{r} \left(1 - \frac{\min(number\ of\ n\ above\ r, R)}{\min(N, R)}\right)$

まとめ

検索エンジンを定量的に評価する指標は、本書で紹介しきれないほど数多くのものが考案されています。本書で紹介したクエリを参考にしながら、NDCGやBPREFなどその他の指標に関しても、その定義や適用範囲を調べて、クエリを考えてみてください。

8-2

データマイニング

データマイニングとは、大量のデータから、特定のパターンやルールなど有用な知識を抽出する手法全般を指す言葉です。代表的なデータマイニングの手法には、相関ルール抽出やクラスタリング、相関分析などが挙げられます。データマイニング手法の多くは再帰的な処理やヒューリスティックな処理が必要となるため、単純にSQLだけで実現することは困難です。通常の実務では、RやPythonなどのパッケージを用いるケースが多いでしょう。

本書では、データマイニングの代表的な手法として、相関ルール抽出の1手法である「アソシエーション分析」を取り上げ、アソシエーション分析の限定的なロジックをSQLで試してみます。ライブラリを使用せずに、データ加工の流れをすべてクエリとして追い掛けることで、データマイニングの理解を深めましょう。

サンプルデータ

本節では、下記データ例に示す購入詳細ログデータを、データマイニングの対象となるログとして利用します。

データ8.2.0.1：購入詳細ログ(purchase_detail_log)テーブル

```
       stamp       | session  | purchase_id | product_id
-------------------+----------+-------------+------------
 2016-11-03 18:10 | 989004ea |           1 | D001
 2016-11-03 18:10 | 989004ea |           1 | D002
 2016-11-03 20:00 | 47db0370 |           2 | D001
 2016-11-04 13:00 | 1cf7678e |           3 | D002
 2016-11-04 15:00 | 5eb2e107 |           4 | A001
 2016-11-04 15:00 | 5eb2e107 |           4 | A002
 2016-11-04 16:00 | fe05e1d8 |           5 | A001
 2016-11-04 16:00 | fe05e1d8 |           5 | A003
 2016-11-04 17:00 | 87b5725f |           6 | A001
 2016-11-04 17:00 | 87b5725f |           6 | A003
 2016-11-04 17:00 | 87b5725f |           6 | A004
 2016-11-04 18:00 | 5d5b0997 |           7 | A005
 2016-11-04 18:00 | 5d5b0997 |           7 | A006
 2016-11-04 19:00 | 111f2996 |           8 | A002
 2016-11-04 19:00 | 111f2996 |           8 | A003
```

```
2016-11-04 20:00 | 3efe001c |           9 | A001
2016-11-04 20:00 | 3efe001c |           9 | A003
2016-11-04 21:00 | 9afaf87c |          10 | D001
2016-11-04 21:00 | 9afaf87c |          10 | D003
2016-11-04 22:00 | d45ec190 |          11 | D001
2016-11-04 22:00 | d45ec190 |          11 | D002
2016-11-04 23:00 | 36dd0df7 |          12 | A002
2016-11-04 23:00 | 36dd0df7 |          12 | A003
2016-11-04 23:00 | 36dd0df7 |          12 | A004
2016-11-05 15:00 | cabf98e8 |          13 | A002
2016-11-05 15:00 | cabf98e8 |          13 | A004
2016-11-05 16:00 | f3b47933 |          14 | A005
```

8-2-1 アソシエーション分析

SQL COUNTウィンドウ関数　　分析 支持度、確信度、リフト

アソシエーション分析は、データマイニングの1分野である「相関ルール抽出」の代表的な手法の1つで、「商品Aを買った人の60%は商品Bも同時に買う」などの相関ルールを、大量のデータから発見する手法です。ここでの相関ルールとは、「商品AとBが同時に買われやすい」という単純な「相関」ではなく、「AならばB」の因果関係を持つルールである点に注意してください。つまり、「AならばB」が成立するから「BならばA」が成り立つとは限りません。

本来のアソシエーション分析は、「商品の組み合わせ（A、B、C…）を買った人のn%は、商品の組み合わせ（X、Y、Z…）を買う」など、商品の組み合わせに関するルールも発見できる汎用的なアルゴリズムですが、完全なアソシエーション分析には再帰的な繰り返し処理が必要となるため、本書では2商品間の相関ルールをマイニングする問題に単純化して考えます。

アソシエーション分析に用いられる指標

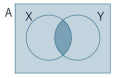
まず、アソシエーション分析に用いられる主な指標、支持度（Support）、確信度・信頼度（Confidence）、リフト（Lift）を紹介します。

支持度（Support）

支持度とは、全データ中で抽出された相関ルールがどれくらいの確率で発生するかを意味する値です。

例えば、100件の購入ログに「商品Xと商品Yを同時に購入したログ」が20件ある場合は、支持度は20%となります（ここでの購入ログは、同時に購入した複数の商品が1件のログに保存されているものとします）。

$$\frac{|X \cap Y|}{|A|}$$

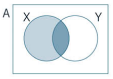


確信度・信頼度（Confidence）

確信度（または信頼度）とは、ある条件を満たす場合に、ある結果がどれくらいの確率で発生するかを意味する値です。

例えば、100件の購入ログに、商品Xを購入しているレコードが50件あり、そのうち商品Yも購入しているレコードが20件あれば、確信度は20／50＝40%となります。

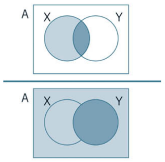
$$\frac{|X \cap Y|}{|X|}$$



リフト（Lift）

リフトとは、ある条件を満たす場合に、結果の確率（つまり確信度）と前提条件なしでその結果が起こる確率の比率を意味する値です。前述の例で、100件の購入ログに、商品Xを購入しているログが50件、商品XとYを同時に購入しているログが20件、商品Yのみを購入しているログが0件だった場合、確信度は20／50＝40%、商品Yの購入確率は20／100＝20%であるので、リフトは40%／20%＝2.0となり、商品Xを購入した場合、商品Yの購入確率が2倍になることが分かります。一般的に、リフトの値が1.0以上となると、良いルールと考えられます。

$$\frac{\left(\frac{|X \cap Y|}{|X|}\right)}{\left(\frac{|Y|}{|A|}\right)}$$



2商品の関連についてアソシエーション分析を行なう

実際にアソシエーション分析で用いる指標を計算して、2商品の関連に関してアソシエーション分析を行います。商品Aと商品Bのアソシエーション分析で必要となる情報は、「購入ログの総数」「商品Aの

購入数」「商品Bの購入数」「商品AとBの同時購入数」の4つです。

下記コード例に示すクエリを用いて、「購入ログの総数」「商品Xの購入数」をそれぞれカウントし、全レコードに展開します。各商品IDに対して、購入ログの総数(purchase_count)と商品購入数(product_count)をカウントできます。

コード8.2.1.1：購入ログ数と商品別購入数をカウントするクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
purchase_id_count AS (
  -- 購入詳細ログからユニークな購入ログ数を計算
  SELECT COUNT(DISTINCT purchase_id) AS purchase_count
  FROM purchase_detail_log
)
, purchase_detail_log_with_counts AS (
  SELECT
      d.purchase_id
    , p.purchase_count
    , d.product_id
    -- 商品別購入数を計算
    , COUNT(1) OVER(PARTITION BY d.product_id) AS product_count
  FROM
      purchase_detail_log AS d
    CROSS JOIN
      -- 購入ログ数をすべてのレコードと結合
      purchase_id_count AS p
)
SELECT
  *
FROM
  purchase_detail_log_with_counts
ORDER BY
  product_id, purchase_id
;
```

```
 purchase_id | purchase_count | product_id | product_count
-------------+----------------+------------+---------------
           1 |             14 | D001       |             4
           1 |             14 | D002       |             3
           2 |             14 | D001       |             4
           3 |             14 | D002       |             3
           4 |             14 | A001       |             4
           4 |             14 | A002       |             4
...
          12 |             14 | A004       |             3
          13 |             14 | A002       |             4
          13 |             14 | A004       |             3
          14 |             14 | A005       |             2
```

次のステップ 続いて、同時に購入された商品のペアを作成し、その組み合わせごとに購入数をカウントするクエリを下記コード例に示します。同時に購入された商品のペアを作成したいので、購入IDで自己結合を行ない、同じ商品のペアを排除しています。

コード8.2.1.2：商品の組み合わせごとに購入数をカウントするクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
purchase_id_count AS (
  -- コード8.2.1.1 参照
)
, purchase_detail_log_with_counts AS (
  -- コード8.2.1.1 参照
)
, product_pair_with_stat AS (
  SELECT
     l1.product_id AS p1
   , l2.product_id AS p2
   , l1.product_count AS p1_count
   , l2.product_count AS p2_count
   , COUNT(1) AS p1_p2_count
   , l1.purchase_count AS purchase_count
  FROM
      purchase_detail_log_with_counts AS l1
    JOIN
      purchase_detail_log_with_counts AS l2
      ON l1.purchase_id = l2.purchase_id
  WHERE
    -- 同じ商品の組み合わせは排除する
    l1.product_id <> l2.product_id
  GROUP BY
      l1.product_id
    , l2.product_id
    , l1.product_count
    , l2.product_count
    , l1.purchase_count
)
SELECT
  *
FROM
  product_pair_with_stat
ORDER BY
  p1, p2
;
```

```
  p1  |  p2  | p1_count | p2_count | p1_p2_count | purchase_count
------+------+----------+----------+-------------+----------------
 A001 | A002 |        4 |        4 |           1 |             14
 A001 | A003 |        4 |        5 |           3 |             14
```

```
A001 | A004 |       4 |       3 |          1 |           14
A002 | A001 |       4 |       4 |          1 |           14
A002 | A003 |       4 |       5 |          2 |           14
A002 | A004 |       4 |       3 |          2 |           14
A003 | A001 |       5 |       4 |          3 |           14
A003 | A002 |       5 |       4 |          2 |           14
A003 | A004 |       5 |       3 |          2 |           14
A004 | A001 |       3 |       4 |          1 |           14
A004 | A002 |       3 |       4 |          2 |           14
A004 | A003 |       3 |       5 |          2 |           14
A005 | A006 |       2 |       1 |          1 |           14
A006 | A005 |       1 |       2 |          1 |           14
D001 | D002 |       4 |       3 |          2 |           14
D001 | D003 |       4 |       1 |          1 |           14
D002 | D001 |       3 |       4 |          2 |           14
D003 | D001 |       1 |       4 |          1 |           14
```

上記コード例の出力結果から、アソシエーション分析に必要となる、「購入ログの総数」「商品Aの購入数」「商品Bの購入数」「商品AとBの同時購入数」の４つの情報を商品ペアごとに計算できました。

次のステップ 指標の定義に従って、アソシエーション分析の指標を計算します。下記コード例に、支持度、確信度、リフトを計算するクエリを示します。

コード8.2.1.3：支持度、確信度、リフトを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
purchase_id_count AS (
  -- コード8.2.1.1 参照
)
, purchase_detail_log_with_counts AS (
  -- コード8.2.1.1 参照
)
, product_pair_with_stat AS (
  -- コード8.2.1.2 参照
)
SELECT
    p1
  , p2
  , 100.0 * p1_p2_count / purchase_count AS support
  , 100.0 * p1_p2_count / p1_count AS confidence
  ,   (100.0 * p1_p2_count / p1_count)
    / (100.0 * p2_count / purchase_count) AS lift
FROM
  product_pair_with_stat
ORDER BY
  p1, p2
;
```

```
  p1  |  p2  |   support   |  confidence  |      lift
------+------+-------------+--------------+-------------
 A001 | A002 |  7.14285714 |  25.00000000 |  0.87500000
 A001 | A003 | 21.42857142 |  75.00000000 |  2.10000000
 A001 | A004 |  7.14285714 |  25.00000000 |  1.16666666
 A002 | A001 |  7.14285714 |  25.00000000 |  0.87500000
 A002 | A003 | 14.28571428 |  50.00000000 |  1.40000000
 A002 | A004 | 14.28571428 |  50.00000000 |  2.33333333
 A003 | A001 | 21.42857142 |  60.00000000 |  2.10000000
 A003 | A002 | 14.28571428 |  40.00000000 |  1.40000000
 A003 | A004 | 14.28571428 |  40.00000000 |  1.86666666
 A004 | A001 |  7.14285714 |  33.33333333 |  1.16666666
 A004 | A002 | 14.28571428 |  66.66666666 |  2.33333333
 A004 | A003 | 14.28571428 |  66.66666666 |  1.86666666
 A005 | A006 |  7.14285714 |  50.00000000 |  7.00000000
 A006 | A005 |  7.14285714 | 100.00000000 |  7.00000000
 D001 | D002 | 14.28571428 |  50.00000000 |  2.33333333
 D001 | D003 |  7.14285714 |  25.00000000 |  3.50000000
 D002 | D001 | 14.28571428 |  66.66666666 |  2.33333333
 D003 | D001 |  7.14285714 | 100.00000000 |  3.50000000
```

上記コード例の出力結果に、「商品p1を買った人が商品p2も買う」相関ルールに対する支持度、確信度、リフトを計算できました。この結果には、考えられる相関ルールがすべて網羅されているので、支持度、確信度、リフトを複合的に検討しながら、有用なルールを選んでいきます。

まとめ

本項で紹介したアソシエーション分析は、2商品間の相関ルールのみに着目した限定的な分析です。本項で紹介したクエリを用いて、相関ルール抽出の理解を深め、より発展的な分析に挑戦してみてください。

8-3 レコメンド

レコメンドシステムとは、ユーザーの趣味嗜好を分析して、ユーザー個別に興味を持ちそうな情報を表示することと考えがちですが、本書ではもう少し広義に捉え、ユーザーに価値がある情報を推薦することと定義します。

普段の生活で何らかの決断を下す際に、クチコミサイトのレビューやランキング、書評などを参考にする方もいるはずです。このような決断を補助もしくは促進するものの1つにレコメンドシステムがあります。特に昨今はサービスで取り扱われる情報や商品は情報過多となり、ユーザーが検索しても臨む情報が得られないシーンが多々あります。そんなときもレコメンドシステムは効果を発揮します。レコメンドシステムを利用することで、ユーザーがまだ知らない商品や情報を認知させ、購入のきっかけを作ることができます。

本節では、後述のItem to Itemのレコメンドリスト、User to Itemのレコメンドリストの作成手順を紹介します。

サンプルデータ

本節では、レコメンドリストを作成するために、下記データ例に示すアイテムへの閲覧／購入ログを用いて集計します。

データ8.3.0.1:アイテムへの閲覧／購入ログ(action_log)テーブル

```
        stamp        | user_id |  action  | product
---------------------+---------+----------+---------
 2016-11-03 18:00:00 | U001    | view     | D001
 2016-11-03 18:01:00 | U001    | view     | D002
 2016-11-03 18:02:00 | U001    | view     | D003
 2016-11-03 18:03:00 | U001    | view     | D004
 2016-11-03 18:04:00 | U001    | view     | D005
 2016-11-03 18:05:00 | U001    | view     | D001
 2016-11-03 18:06:00 | U001    | view     | D005
 2016-11-03 18:10:00 | U001    | purchase | D001
 2016-11-03 18:10:00 | U001    | purchase | D005
 2016-11-03 19:00:00 | U002    | view     | D001
 2016-11-03 19:01:00 | U002    | view     | D003
 2016-11-03 19:02:00 | U002    | view     | D005
```

```
2016-11-03 19:03:00 | U002    | view     | D003
2016-11-03 19:04:00 | U002    | view     | D005
2016-11-03 19:10:00 | U002    | purchase | D001
2016-11-03 19:10:00 | U002    | purchase | D005
2016-11-03 20:00:00 | U003    | view     | D001
2016-11-03 20:01:00 | U003    | view     | D004
2016-11-03 20:02:00 | U003    | view     | D005
2016-11-03 20:10:00 | U003    | purchase | D004
2016-11-03 20:10:00 | U003    | purchase | D005
```

8-3-1 レコメンドシステムを広義に捉える

分析 レコメンドシステム

レコメンドシステムは時に定義や解釈が異なり混乱を招くことがあります。これは本節冒頭で述べた広義な解釈と狭義な解釈の差にあります。レコメンドシステムが意味する範囲が広いこと、その目的が多岐にわたること、また、レコメンドシステムの知識不足から混乱を招いているためでしょう。

本項では、広義にレコメンドシステムを捉えて、どのようなレコメンドシステムがあるのかを、その種類やモジュール、効果や精度の観点から整理します。

レコメンドの種類

レコメンドには大きく2つの種類があります。1つは、表示中のアイテムや購入したアイテムに対して、別アイテムを推薦する「Item to Item」、もう1つは、ユーザーの過去の行動やデモグラフィック情報から、趣味嗜好を類推してアイテムを推薦する「User to Item」です。

表8.3.1.1：レコメンドの種類

種類	内容	表示例
Item to Item	アイテムに関連した、別のアイテムを提案する	この商品を見ている人はこんな商品も見ています。
User to Item	ユーザー個人に最適化したアイテムを提案する	あなたへのオススメ

「あなたへのオススメ」と表示されても、ユーザーが直近に購入したアイテムを軸に、該当アイテムの購入ユーザーが次に購入しているものを推薦するロジックであれば、Item to Itemのレコメンドです。

また、Item to ItemとUser to Itemは共に、必ずしも商品や記事単位である必要はなく、ECサイトのよく売れている商品カテゴリ、飲食店予約サイトでの店舗エリアコードも推薦の対象となります。

なお、User to Itemのユーザー行動を元に趣味嗜好を推測するシステムでは、登録から間もないユーザーは趣味嗜好を推測する十分なデータが揃っていないことがあり、注意が必要です。

モジュールの種類

普段使用しているECサイトには、どのようなレコメンドモジュールがあるのか、広義にレコメンドを捉えて整理します。

表 8.3.1.2：モジュールの種類

モジュール	内容	表示例
リマインド	ユーザーの過去の行動から、再度アイテムを提案する	最近見た商品 もう一度買う
ランキング	閲覧数、購入数等に基づき、人気のあるアイテムを提案する	人気ランキング 急上昇ランキング
コンテンツベース	アイテムに付属する情報を元に、別のアイテムを提案する	この俳優が出演している他の作品
レコメンド	サービスを利用するユーザー全体の行動履歴から、次に見るアイテム、関連するアイテムを推測し、提案する	この商品を見ている人はこんな商品も見ています。
パーソナライズレコメンド	ユーザー個人の行動履歴から趣味嗜好を推測し、興味のありそうなアイテムを提案する	あなたへのオススメ

高度な計算ロジックを用いることなく、過去に表示した情報を閲覧履歴として表示するだけのモジュールも、リマインドを目的とするレコメンドモジュールと捉えることができます。

例えば、売上や閲覧数の上位を占めるものを表示するだけでも、ランキングを使用したレコメンドといえます。売上や閲覧数の上位を集計する方法は既に、「4-2 多面的な軸を使ってデータを集約する」（P.112）や「6-1-2 ページ毎の訪問者数、訪問回数、ページビューを集計する」（P.247）で紹介しています。また、季節性を重視するランキングであれば、「4-2-3 ファンチャートで商品の売れ行きの伸び率を見る」（P.119）で紹介した伸び率をソートスコアとして採用することで、急上昇ランキングといえるモジュールを作成できます。

なお、レコメンドとパーソナライズレコメンドは、後述の「8-3-2 このアイテムを見た人はこのアイテムも見ています」と「8-3-3 あなたにオススメの商品」で、その定義や集計方法を解説します。

レコメンドの効果

レコメンドシステムは情報過多な状況から、ユーザーが価値ある情報に辿り着きやすくする効果も当然ありますが、ECサイトでのレコメンド効果に着目して下記の3つに分類します。

表 8.3.1.3：レコメンドの効果

効果	内容	表示例
ダウンセル	価格が高く、購入を悩んでいるユーザーにより安いアイテムを提案し、購入数を上げる	過去モデルで価格が安い商品があります
クロスセル	付属品、関連商品を一緒に購入してもらうことにより、購入単価を上げる	一緒に購入されている商品
アップセル	より上位のモデルや高機能なアイテムを提案し、購入単価を上げる	この商品には新しいモデルがあります

レコメンドの効果を解説する例によく用いられるのがハンバーガーショップでのやり取りです。

「ご一緒にポテトはいかがですか？」はクロスセル、「＋50円でポテトをMサイズからLサイズに変更ができますが、いかがですか？」はアップセルです。ダウンセルはハンバーガーショップではあまり見掛けませんが、「ポテトとドリンクのサイズがSサイズになりますが、こちらの商品はお手頃な価格です」などのやり取りです。

この他にもレコメンドで得られる効果はさまざまですが、1訪問当たりの遷移率や閲覧数を上げるため、新着商品や人気商品以外のあまり閲覧されない商品を敢えて推薦し、新しい発見を与える（意外性重視）など、レコメンドシステムを構築する際には、何を目的とするのかを明確にすることが重要です。

データによって得られる精度、効果の違い

レコメンドシステムを構築する際に重要なのは、どのようなデータを元にレコメンドシステムを構築するかです。従来はデータベースに格納されている購入データや会員データしか利用できませんでしたが、昨今ではビッグデータの技術進歩で商品の閲覧ログまでもが取り扱えます。利用できる各種のデータをどのように収集し、どのように利用するかで、得られる効果が変わることを説明します。

データの明示的獲得と暗黙的獲得

レコメンドシステム構築時に使用するデータを収集する方法には、ユーザー行動から好みを推測する暗黙的なデータ獲得と、ユーザーに好みを直接聞く明示的なデータ獲得の2種類があります。この2つの違いを説明します。

表8.3.1.4：データの明示的獲得と暗黙的獲得

データ元	内容	データ例
暗黙的なデータ獲得	ユーザーの行動から、好みを推測する。 データ量は多いが、評価の正確性に欠ける。	購入データ 閲覧ログ
明示的なデータ獲得	ユーザーに好みを聞く。 データ量は少なくなるが、評価に段階を設けることができ、評価の正確さは高い。	クチコミ時の評価点

暗黙的なデータ獲得は、「購入したのであれば、きっとこの商品を好きに違いない」との前提の元、購入データをレコメンドシステムのデータソースとして採用します。しかし、商品の購入が必ずしも好みであるかは不明です。例えば、「DVDを購入したけれども面白くなかった」ことも実際にはあり得ます。

一方で明示的なデータ獲得は、クチコミやレビュー時の5段階評価で、ユーザーの好みを正確なデータとして収集可能です。しかし、訪問先や購入商品のすべてにクチコミやレビューを必ず行うとは限らず、暗黙的なデータ獲得よりもデータが少なくなります。

また、レコメンドシステム構築時に、何を元にレコメンドしているのかユーザーに明らかにすることで、納得感を得られるケースがあります。明示的に獲得したデータはその根拠を示しても問題ありませんが、暗黙的に獲得したデータの場合、プライバシー侵害ではないかと嫌悪感を抱くユーザーも少なからずいることに注意する必要があります。

閲覧ログと購入ログの違い

暗黙的なデータを利用する場合、閲覧ログと購入ログのどちらをデータソースとして採用するかで、推薦される情報が変わる場合があります。

表8.3.1.5：閲覧ログと購入ログを用いた際の効果の違い

データ元	内容	データ例
閲覧ログ	あるアイテムに類似したアイテムが提案されやすく、選択肢を多く与える効果がある	スーツケースに対して、サイズ、形状、色の異なるものを提案する
購入ログ	クロスセル可能なアイテムが提案されやすく、購入単価を引き上げる効果がある	スーツケースに対して、一緒に利用するパスポートケースを提案する

商品購入時に、サイズ違いや色違い、他メーカーの類似品などと比較して購入を決断するユーザーがいます。そのユーザー行動を記録した閲覧ログを元にレコメンドシステムを構築すると、ある商品に対して、サイズや形状、色などが異なる類似商品がレコメンドされるケースがあります。閲覧商品が好みでない場合、類似商品を提案することで、購入してもらえるきっかけを用意できます。

一方、購入ログを元にレコメンドシステムを構築すると、一緒に購入されている商品、次に良く購入する商品など、閲覧ログとは異なる結果が得られるケースがあります。

まとめ

レコメンドシステムには、さまざまな目的や効果、モジュールがあります。レコメンドシステムの定義をメンバー間できちんと揃えた上で、何を目的にどんな効果をレコメンドに期待するかを具体化して取り組むと良いでしょう。

8-3-2 このアイテムに興味がある人はこんなアイテムも見ています

SQL SUM(CASE〜)、ROW_NUMBER関数

分析 ベクトルの内積、2ノルム正規化、コサイン類似度

ユーザーのアクセスログを用いて作成できるレコメンドシステムの中でも、比較的シンプルに実装でき、かつ効果の高いレコメンドが、「このアイテムに興味がある人はこんなアイテムも見ています」などのItem to Itemレコメンドです。

Item to Itemレコメンドが、User to Itemレコメンドと比較してシンプルに実装できて効果的な理由は、一般的なサービスでは、ユーザーの流動性よりもアイテムの流動性が低く、データの蓄積が容易なためです。例えば、今日サイトにアクセスしたユーザーが、翌日もアクセスする保証はありませんが、今日発売中のアイテムを明日も継続して発売するかはサイト側でコントロールできます。

アイテムが継続的にサイトで公開されていれば、公開期間のアクセスログも継続的に蓄積されるので、ユーザーへのレコメンドと比較して、ログ収集のハードルが低くなります。

また、ユーザーの興味や関心は時間と共に変化しますが、アイテム間の関連は時間が経っても大きくは変化しません。比較的古いアクセスログでも、ある程度の精度を保つことが可能なことを意味します。ただし、ニュース記事やオークションなど、アイテム自体が流動的なケースでは、上記のメリットを享受できないので注意が必要です。

アクセスログを用いてアイテム間の相関度を計算する

実際にアクセスログを用いてアイテム間の相関度を計算しましょう。

本項で使用するアクセスログには、ユーザーごとのアイテムに対する閲覧と購入アクションが蓄積されています。前述の「8-3-1 レコメンドシステムを広義に捉える」（P.407）でも説明した通り、閲覧ログと購入ログの特徴をうまく活用すれば、より効果的なレコメンドを作成できます。

閲覧ログと購入ログを同時に扱う汎用的なクエリを紹介し、両者の重みを変更することで、閲覧ベースのレコメンドや購入ベースのレコメンドに応用できる形にしましょう。下記のコード例に、ユーザーとアイテムの組み合わせに対して、閲覧数と購入数を集計し、両者を組み合わせたスコアを計算するクエリを示します。

コード例では、ユーザー単位のアイテム閲覧数と購入数を3:7の割合で重み付けした平均を取り、該当ユーザーのアイテムへの興味をスコアとして定義しています。

コード8.3.2.1: 閲覧数と購入数を組み合わせたスコアを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
ratings AS (
  SELECT
      user_id
    , product

      -- 商品の閲覧数
    , SUM(CASE WHEN action = 'view'     THEN 1 ELSE 0 END) AS view_count

      -- 商品の購入数
    , SUM(CASE WHEN action = 'purchase' THEN 1 ELSE 0 END) AS purchase_count

      -- 閲覧数と購入数を3:7の割合で重み付き平均する
    ,   0.3 * SUM(CASE WHEN action = 'view'     THEN 1 ELSE 0 END)
      + 0.7 * SUM(CASE WHEN action = 'purchase' THEN 1 ELSE 0 END)
      AS score
  FROM
    action_log
  GROUP BY
    user_id, product
```

```
)
SELECT *
FROM
  ratings
ORDER BY
  user_id, score DESC
;
```

```
 user_id | product | view_count | purchase_count | score
---------+---------+------------+----------------+-------
 U001    | D005    |          2 |              1 |   1.3
 U001    | D001    |          2 |              1 |   1.3
 U001    | D003    |          1 |              0 |   0.3
 U001    | D004    |          1 |              0 |   0.3
 U001    | D002    |          1 |              0 |   0.3
 U002    | D005    |          2 |              1 |   1.3
 U002    | D001    |          1 |              1 |   1.0
 U002    | D003    |          2 |              0 |   0.6
 U003    | D005    |          1 |              1 |   1.0
 U003    | D004    |          1 |              1 |   1.0
 U003    | D001    |          1 |              0 |   0.3
```

次のステップ ユーザーのアイテムに対する興味の度合いを数値化できたら、それらのスコアを組み合わせて、アイテム間の類似度を計算します。下記コード例に、アイテム間の類似度を計算して、ランキング形式で表示するクエリを示します。

コード8.3.2.2:アイテム間の類似度を計算しランキングを作成するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
ratings AS (
  -- コード8.3.2.1 参照
)
SELECT
    r1.product AS target
  , r2.product AS related
    -- 両方のアイテムを閲覧または購入しているユーザー数
  , COUNT(r1.user_id) AS users
    -- スコア同士の掛け算を合計して、関連度を計算する
  , SUM(r1.score * r2.score) AS score
    -- 商品の関連度順
  , ROW_NUMBER()
      OVER(PARTITION BY r1.product ORDER BY SUM(r1.score * r2.score) DESC)
    AS rank
FROM
    ratings AS r1
```

```
  JOIN
    ratings AS r2
    -- 共通のユーザーが存在する商品のペアを作成する
    ON r1.user_id = r2.user_id
WHERE
  -- 同じアイテムのペアは排除する
  r1.product <> r2.product
GROUP BY
  r1.product, r2.product
ORDER BY
  target, rank
;
```

```
 target | related | users | score | rank
--------+---------+-------+-------+------
 D001   | D005    |     3 |  3.29 |    1
 D001   | D003    |     2 |  0.99 |    2
 D001   | D004    |     2 |  0.69 |    3
 D001   | D002    |     1 |  0.39 |    4
 D002   | D001    |     1 |  0.39 |    1
 D002   | D005    |     1 |  0.39 |    2
 D002   | D003    |     1 |  0.09 |    3
 D002   | D004    |     1 |  0.09 |    4
 D003   | D005    |     2 |  1.17 |    1
 D003   | D001    |     2 |  0.99 |    2
 D003   | D004    |     1 |  0.09 |    3
 D003   | D002    |     1 |  0.09 |    4
 D004   | D005    |     2 |  1.39 |    1
 D004   | D001    |     2 |  0.69 |    2
 D004   | D002    |     1 |  0.09 |    3
 D004   | D003    |     1 |  0.09 |    4
 D005   | D001    |     3 |  3.29 |    1
 D005   | D004    |     2 |  1.39 |    2
 D005   | D003    |     2 |  1.17 |    3
 D005   | D002    |     1 |  0.39 |    4
```

アイテム間の類似度を計算するには、両方のアイテムを閲覧もしくは購入しているユーザーが存在する必要があります。したがって、ratingsテーブルをユーザーIDで自己結合し、共通のユーザーが存在するアイテムのペアを作成します。さらに、同じアイテムのペアを排除した上で、アイテムのペアで集約し、両者の類似度を集計します。

類似度の集計では、2つのアイテムに対するユーザーのスコアを掛け算し、全ユーザーで足し合わせる計算を行っています。「ベクトルの内積」と呼ばれる値ですが、ここでのベクトルは、あるアイテムに対して付与されたユーザースコアを列挙した、「数字の集まり」と考えてください。スコアの掛け算で、両アイテムが共に高いスコアの場合に、類似度が高くなるスコアを定義できます。

スコアの正規化を行なう

前述の単純なベクトルの内積を用いた類似度は、実際にレコメンドする際には精度にいくつか課題があります。1つはアクセス数が多いアイテムの類似度が相対的に高くなってしまう問題です。例えば、あるユーザーが特定アイテムに対して数百回や数千回アクセスしているケースでは、アクセスログからそのまま内積を用いて類似度を計算すると、その特定アイテムのスコアが常に高くなります。

また、スコアの値がどれくらいの類似度を表しているか、その意味付けが難しいことです。例えば、アイテムAとアイテムBの内積による類似度が3.7である場合に、そのスコアが高いのか低いのかを単純に判断できません。他のアイテムと比べて、相対的にスコアの位置付けを都度調べる必要があります。

上述の問題に対処するために用いられるのが「ベクトルの正規化」です。ベクトルの正規化とは、ベクトルをすべて同じ大きさに揃えることです。ベクトルの大きさ自体にもさまざまな定義がありますが、よく利用されるのは、ベクトルの各数値を2乗して足し合わせたものの平方根を取った値です。

これは2点間の距離を計算するために用いられる、ユークリッド距離の定義と同様です。ベクトルの大きさを「ノルム」(Norm) とも呼び、ノルムでベクトルの各数値を割ることで、ベクトルのノルムを1に揃えることが可能です。2乗和の平方根でベクトルを正規化することを、「2ノルム正規化」もしくは「L2正規化」と呼びます。

下記のコード例に、ユーザーのアイテムに対するスコアの集合をアイテムのベクトルと見立てて、2ノルム正規化を行うクエリを示します。ベクトルのノルム計算には、SUM() OVER()関数を用いてベクトル全体の値の2乗和を求め平方根を取ります。求められたノルムで各スコアを割ることで、正規化されたスコア (norm_score) を計算します。

コード8.3.2.3:アイテムベクトルを2ノルム正規化するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
ratings AS (
  -- コード8.3.2.1 参照
)
, product_base_normalized_ratings AS (
  -- アイテムを基準にした正規化を行う
  SELECT
      user_id
    , product
    , score
    , SQRT(SUM(score * score) OVER(PARTITION BY product)) AS norm
    , score / SQRT(SUM(score * score) OVER(PARTITION BY product)) AS norm_score
  FROM
    ratings
)
SELECT *
FROM
```

```
  product_base_normalized_ratings
;
```

```
 user_id | product | score |        norm          |       norm_score
---------+---------+-------+----------------------+------------------------
 U001    | D001    |   1.3 |   1.667333200053307 | 0.77968818707528714825
 U002    | D001    |   1.0 |   1.667333200053307 | 0.59976014390406703712
 U003    | D001    |   0.3 |   1.667333200053307 | 0.17992804317122011113
 U001    | D002    |   0.3 |   0.3000000000000000 | 1.00000000000000000000
 U002    | D003    |   0.6 |   0.6708203932499360 | 0.89442719099991587713
 U001    | D003    |   0.3 |   0.6708203932499360 | 0.44721359549995793856
 U003    | D004    |   1.0 |   1.0440306508910550 | 0.95782628522115140913
 U001    | D004    |   0.3 |   1.0440306508910550 | 0.28734788556634542274
 U001    | D005    |   1.3 |   2.0928449536456350 | 0.62116402733774553208
 U002    | D005    |   1.3 |   2.0928449536456350 | 0.62116402733774553208
 U003    | D005    |   1.0 |   2.0928449536456350 | 0.47781848256749656314
```

正規化を行うとベクトルの大きさは必ず1になるので、上記コード例の出力結果では、ベクトルの値が1つしかないD002のアイテムのnorm_scoreが1.0に正規化されていることが分かります。

次のステップ 続いて、正規化されたスコアを用いて、もう一度アイテムの類似度を計算します。

下記コード例に、正規化されたスコアでアイテムの類似度を計算するクエリを示します。コード例では、同じアイテムのペアも残した状態で類似度を計算しています。

コード8.3.2.4:正規化されたスコアでアイテムの類似度を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
ratings AS (
  -- コード8.3.2.1 参照
)
, product_base_normalized_ratings AS (
  -- コード8.3.2.3 参照
)
SELECT
    r1.product AS target
  , r2.product AS related
    -- 両方のアイテムを閲覧または購入しているユーザー数
  , COUNT(r1.user_id) AS users
    -- スコア同士の掛け算を合計して、関連度を計算する
  , SUM(r1.norm_score * r2.norm_score) AS score
    -- 商品の関連度順
  , ROW_NUMBER()
      OVER(PARTITION BY r1.product ORDER BY SUM(r1.norm_score * r2.norm_score) DESC)
    AS rank
```

```
FROM
    product_base_normalized_ratings AS r1
  JOIN
    product_base_normalized_ratings AS r2
    -- 共通のユーザーが存在する商品のペアを作成する
    ON r1.user_id = r2.user_id
GROUP BY
  r1.product, r2.product
ORDER BY
  target, rank
;
```

```
 target | related | users | score | norm_score | rank
--------+---------+-------+-------+------------+------
 D001   | D001    |     3 |  2.78 | 0.99999999 |    1  -- 自分自身との相関が必ず1になる
 D001   | D005    |     3 |  3.29 | 0.94283662 |    2
 D001   | D003    |     2 |  0.99 | 0.88512893 |    3
 D001   | D002    |     1 |  0.39 | 0.77968818 |    4
 D001   | D004    |     2 |  0.69 | 0.39638156 |    5  -- 正規化前と順序が変わっている
 D002   | D002    |     1 |  0.09 | 1.00000000 |    1  -- 自分自身との相関が必ず1になる
 D002   | D001    |     1 |  0.39 | 0.77968818 |    2
 D002   | D005    |     1 |  0.39 | 0.62116402 |    3
 D002   | D003    |     1 |  0.09 | 0.44721359 |    4
 D002   | D004    |     1 |  0.09 | 0.28734788 |    5
 D003   | D003    |     2 |  0.45 | 0.99999999 |    1  -- 自分自身との相関が必ず1になる
 D003   | D001    |     2 |  0.99 | 0.88512893 |    2  -- 正規化前と順序が変わっている
 D003   | D005    |     2 |  1.17 | 0.83337899 |    3  -- 正規化前と順序が変わっている
 D003   | D002    |     1 |  0.09 | 0.44721359 |    4
 D003   | D004    |     1 |  0.09 | 0.12850588 |    5
 D004   | D004    |     2 |  1.09 | 1.00000000 |    1  -- 自分自身との相関が必ず1になる
 D004   | D005    |     2 |  1.39 | 0.63615727 |    2  -- 正規化前と順序が変わっている
 D004   | D001    |     2 |  0.69 | 0.39638156 |    3
 D004   | D002    |     1 |  0.09 | 0.28734788 |    4
 D004   | D003    |     1 |  0.09 | 0.12850588 |    5
 D005   | D005    |     3 |  4.38 | 0.99999999 |    1  -- 自分自身との相関が必ず1になる
 D005   | D001    |     3 |  3.29 | 0.94283662 |    2
 D005   | D003    |     2 |  1.17 | 0.83337899 |    3
 D005   | D004    |     2 |  1.39 | 0.63615727 |    4  -- 正規化前と順序が変わっている
 D005   | D002    |     1 |  0.39 | 0.62116402 |    5
```

上記コード例の出力結果を確認すると、同じアイテムのペアの類似度が、必ず１であることが分かります。ベクトルを正規化することで自分自身との内積の値は必ず1となり、内積の最大値も1となります。

また、ベクトルすべての値が正の値であれば内積の最小値は０となります。つまり、内積の値が０のときがまったく類似性のないアイテム、１のときが完全に一致するアイテムとして、アイテム間の類似度を定量的に意味付けすることが可能です。

なお、ベクトルを2ノルム正規化して内積を取った値は、2つのベクトルをn次元上にマッピングしたときになす角度のコサインに一致するため、「コサイン類似度」とも呼ばれます。

まとめ

本項で紹介しているアイテム間の類似度を計算する手法ではベクトルの内積を用いていますが、アイテムのベクトル作成は、アイテムの閲覧数と購入数の重み付き平均を取るだけの単純なものです。実際のサービスでレコメンドに活用するには、対象サービスの特性を考慮して、ベクトルを作成するロジックを工夫してください。

関連項目 | 8-4-1 複数の値をバランスよく組み合わせてスコアを計算する

8-3-3 あなたにオススメの商品

SQL SUM(CASE〜)、ROW_NUMBER関数

分析 ベクトルの内積、2ノルム正規化、コサイン類似度

前項「8-3-2 このアイテムに興味がある人はこんなアイテムも見ています」(P.410)で説明した「Item to Item」レコメンドに続き、「あなたにオススメの商品」を計算するロジックとして、「User to Item」レコメンドの作成方法を紹介します。

User to Itemレコメンドは、ユーザーに紐付くアイテムのレコメンドなので、特定アイテムに紐付く必要があるItem to Itemと比べると、Webサイトのトップページやユーザーのマイページ、検索結果のNoMatchページなど各種ページでの表示に加えて、メルマガ配信やプッシュ通知など、さまざまな場面で活用できます。

Item to Itemはアイテムとアイテムの類似度を計算するだけですが、User to Itemでは、まずユーザーとユーザーの類似度を計算し、類似ユーザーが興味のあるアイテムを取得するという2段階の計算が必要になります。

下記コード例に、ユーザー同士の類似度を計算するクエリを紹介します。前項で紹介した2ノルム正規化を適用しますが、ユーザーのベクトルの内積を計算するため、ユーザーごとのベクトルのノルムを計算している点に注意してください。

コード8.3.3.1:ユーザー同士の類似度を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
ratings AS (
  -- コード8.3.2.1 参照
)
, user_base_normalized_ratings AS (
```

```
  -- ユーザーを基準にした正規化を行う
  SELECT
      user_id
    , product
    , score
      -- PARTITION BY user_id でユーザーごとのベクトルのノルムを計算
    , SQRT(SUM(score * score) OVER(PARTITION BY user_id)) AS norm
    , score / SQRT(SUM(score * score) OVER(PARTITION BY user_id)) AS norm_score
  FROM
    ratings
)
, related_users AS (
  -- 傾向の近いユーザーを発見する
  SELECT
      r1.user_id
    , r2.user_id AS related_user
    , COUNT(r1.product) AS products
    , SUM(r1.norm_score * r2.norm_score) AS score
    , ROW_NUMBER()
        OVER(PARTITION BY r1.user_id ORDER BY SUM(r1.norm_score * r2.norm_score) DESC)
      AS rank
  FROM
      user_base_normalized_ratings AS r1
    JOIN
      user_base_normalized_ratings AS r2
      ON r1.product = r2.product
  WHERE
    r1.user_id <> r2.user_id
  GROUP BY
    r1.user_id, r2.user_id
)
SELECT *
FROM
  related_users
ORDER BY
  user_id, rank
;
```

```
 user_id | related_user | products |    score    | rank
---------+--------------+----------+-------------+------
 U001    | U002         |        3 | 0.950085981 |    1
 U001    | U003         |        3 | 0.720499196 |    2
 U002    | U001         |        3 | 0.950085981 |    1
 U002    | U003         |        2 | 0.633719279 |    2
 U003    | U001         |        3 | 0.720499196 |    1
 U003    | U002         |        2 | 0.633719279 |    2
```

次のステップ 類似ユーザーのランキングを計算したら、続いて類似ユーザーが興味のあるアイテム一覧を取得し、アイテムのランキングを作成します。

下記コード例に、上位の類似ユーザーからおすすめアイテムを取得するクエリを示します。類似ユーザー全員を用いてアイテム一覧を取得すると、膨大な計算量が必要になる割に、低類似度のユーザーが興味のあるアイテムは最終的なランキングでも下位に位置します。類似度が上位n件のユーザーに絞り込むことで、効率的におすすめ商品を計算できます。

ただし、作成された「おすすめ商品」のアイテムには、ユーザーが既に購入済みの商品が含まれる可能性があります。元データにユーザーの購入ログも含まれているので、購入済み商品をフィルタするとより効果的なレコメンドになります。

コード8.3.3.2: 上位の類似ユーザーからおすすめアイテムを取得するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
ratings AS (
  -- コード8.3.2.1 参照
)
, user_base_normalized_ratings AS (
  -- コード8.3.3.1 参照
)
, related_users AS (
  -- コード8.3.3.1 参照
)
, related_user_base_products AS (
  SELECT
      u.user_id
    , r.product
    , SUM(u.score * r.score) AS score
    , ROW_NUMBER()
        OVER(PARTITION BY u.user_id ORDER BY SUM(u.score * r.score) DESC)
      AS rank
  FROM
      related_users AS u
    JOIN
      ratings AS r
      ON u.related_user = r.user_id
  WHERE
    u.rank <= 1
  GROUP BY
    u.user_id, r.product
)
SELECT *
FROM
  related_user_base_products
ORDER BY
  user_id
;
```

```
 user_id | product |     score    | rank
---------+---------+--------------+------
 U001    | D005    | 1.2351117764 |    1
 U001    | D001    | 0.9500859819 |    2
 U001    | D003    | 0.5700515891 |    3
 U002    | D001    | 1.2351117764 |    1
 U002    | D005    | 1.2351117764 |    2
 U002    | D002    | 0.2850257945 |    3
 U002    | D003    | 0.2850257945 |    4
 U002    | D004    | 0.2850257945 |    5
 U003    | D005    | 0.9366489560 |    1
 U003    | D001    | 0.9366489560 |    2
 U003    | D004    | 0.2161497590 |    3
 U003    | D002    | 0.2161497590 |    4
 U003    | D003    | 0.2161497590 |    5
```

次のステップ ユーザーのアイテム購入数を用いて、既に購入されているアイテムをフィルタするクエリを、下記のコード例に示します。未購入の商品を結果に残すため、User to Item結果にLEFT JOINで購入数を結合後、アイテムの購入が0またはNULLのアイテムに絞り込んで、ランキングを作成します。

コード8.3.3.3:すでに購入されているアイテムをフィルタするクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
ratings AS (
  -- コード8.3.2.1 参照
)
, user_base_normalized_ratings AS (
  -- コード8.3.3.1 参照
)
, related_users AS (
  -- コード8.3.3.1 参照
)
, related_user_base_products AS (
  -- コード8.3.3.2 参照
)
SELECT
    p.user_id
  , p.product
  , p.score
  , ROW_NUMBER()
      OVER(PARTITION BY p.user_id ORDER BY p.score DESC)
    AS rank
FROM
    related_user_base_products AS p
  LEFT JOIN
    ratings AS r
    ON p.user_id = r.user_id
```

```
    AND p.product = r.product
WHERE
  -- 対象ユーザーがまだ買っていない商品に絞り込む
  COALESCE(r.purchase_count, 0) = 0
ORDER BY
  p.user_id
;
```

```
 user_id | product |     score    | rank
---------+---------+--------------+------
 U001    | D003    | 0.5700515891 |    1
 U002    | D004    | 0.2850257945 |    1
 U002    | D003    | 0.2850257945 |    2
 U002    | D002    | 0.2850257945 |    3
 U003    | D001    | 0.9366489560 |    1
 U003    | D003    | 0.2161497590 |    2
 U003    | D002    | 0.2161497590 |    3
```

User to Itemは登録して間もないユーザーやゲストユーザーに関しては、趣味嗜好を推測するための十分なデータが揃っていないため注意が必要です。対応策として考えられるのは、登録期間が短いユーザーには、User to Itemではない他のレコメンドロジック（ランキングやコンテンツベース）を適用することです。また、ゲストユーザー全員を1人の仮想ユーザーと見立てて、User to Itemを作成して対応することも可能です。

まとめ

本項で紹介した、ユーザーの行動ログから類似アイテムやオススメ商品を計算する手法は、「協調フィルタリング」と呼ばれる概念の1手法です。協調フィルタリングには、機械学習を活用する発展的な手法も多く考案されているので、本項の手法を足掛かりに、各種のレコメンドに挑戦してください。

8-3-4 レコメンドシステムを改善するポイント

レコメンドシステムは、「サービスの利用者が欲しい情報と推測したもの」と「サービスの利用者が欲しいもの」の関係が一致した際に、最大の効果を発揮します。しかし、レコメンドシステムを構築したといっても、必ずしもその関係が一致するとは限りません。継続的にレコメンドシステムのロジックを改善し、さまざまな検証を繰り返して、精度を高めていくことが重要です。

本項では、レコメンドシステムを構築して、継続的に運用を進めていく際、どのように精度を高めるかを説明します。

値・リストの操作による改善ポイント

レコメンドシステムの精度を高める手法として、主に「重み付け」「フィルタ」「ソート」が挙げられます。それぞれの手法をどのようなシーンで活用するか、下記にまとめます。

重み付け

閲覧ログと購入データ双方をデータソースとして収集する際、どちらをより評価するか重み付けします。例えば、閲覧1ポイントに対して購入は3ポイントなど、スコアの重み付けをすることで、レコメンドされる内容が変わります。「8-4 スコアの計算」(P.427) で後述する紹介する手法を用いれば、より発展的な重み付けを使ったレコメンドのチューニングが可能です。

フィルタ

関連を導き出すには不十分なデータや異常値を除外し、より正しい関連が導き出せるデータを残すことで、レコメンドの精度を高めることができます。残念なことに、特定商品をより多く閲覧させる目的で、スパム攻撃やサクラ攻撃を実行するなど、悪意を持つユーザーも存在します。そのため、異常値を除外することもレコメンドの精度を高める上では重要な作業です。

また、ユーザーのデモグラフィック情報を元に、男性と女性で個別にレコメンドスコアを計算することや、レコメンドするアイテムを同一カテゴリや同一の都道府県でフィルタして、より根拠や納得感があるレコメンドを作成することも改善ポイントです。

ソート

最終的なレコメンド結果は、何らかの定義に基づくスコアでソートしたランキング形式で提供されるケースがほとんどです。スコアの計算方法は、レコメンドがどのような需要を満たす目的であるかによって変化します。例えば、新しいアイテムを多くレコメンドするには、新規性 (Novelty) のスコアを定義、レコメンド経由での売上額を上げたければ、アイテムの金額や期待購入率を考慮したスコアを定義します。目的に応じて指標を組み合わせてソートすることも改善ポイントの1つです。

また、限定的な例ですが、飲食店のレビューサイトのレコメンドでは、評価は高いが所在地が遠い店舗、評価はそこそこだが近い店舗をそれぞれ評価する場合、起点となる位置 (現在地と店舗の緯度経度など) から、レコメンドリスト内にある他の情報から距離を導き出し、距離÷評価で求めた値を元にソートして、ユーザーに適切な選択肢を提示することも可能です。

構築手順における改善ポイント

レコメンドシステムの構築には、大別すると下記の4フェーズがあります。各フェーズにおける、前述の重み付け、フィルタ、ソートを活用して、どのような改善が可能か整理しましょう。

- データの収集
- データの加工
- スコアの計算
- データの返却

データ収集時における対象の拡充

各種データが存在しなければ、レコメンドシステムの構築は不可能です。構築したいレコメンドのロジックとモジュールに対して、収集すべきデータと元にするデータを定めて、収集する必要があります。

購入データしか存在しない状態よりも、クチコミデータや閲覧データ、商品データ、デモグラフィックデータなど、さまざまなデータを取り扱える方がより精度の高いレコメンドシステムを構築可能です。また、閲覧ログを用いてレコメンドシステムを構築した結果、あまり効果がない場合に、レコメンドシステムを諦めるのではなく、別の手段を検討して即実施することも可能です。

データ収集の段階で、精度が高くより多くのデータを集めることが最初の改善ポイントです。

データ加工時における信頼性の担保と評価期間、重み付けの調整

データ収集後に、スコアを計算しやすい形式に加工すると共に、データの異常値を排除し精度を高めることで、レコメンド精度を向上できます。加工済みデータが信頼できないと、以降の作業にも支障を来します。

また、過去や直近などでの季節による移り変わりや趣味嗜好の変化、家族構成の変更などが生じるケースがあります。評価期間の限定や対象期間で重み付けを変更することで、より現状に即したレコメンド結果を用意できます。

スコア計算時におけるロジック

広義でレコメンドを捉えるとランキングもロジックの1つです。相関の計算よりも単純に売上順に並べるロジックの方が高いケースもあります。また、本節で紹介した2つのレコメンドロジック以外にも、数多くロジックが存在します。採用するロジックによって精度や内容を大きく変更することも可能です。

また、ロジックを変更せずとも、デモグラフィック情報（性別・年齢など）に応じて、各スコアを計算することで異なる結果が得られるケースもあります。

データ返却時におけるフィルタとソート

スコアを計算した結果をそのままAPIとして返却し、サービス側で表示することも可能ですが、この段階でフィルタとソートを実行することで、より効果的なレコメンドリストにすることが可能です。

例えば、当日予約する飲食店を探すシーンでは、仮に遠方の飲食店に相関があっても、ユーザーは今注目している飲食店からさほど遠くない場所で予約することが推測できます。こうしたケースでは、検索条件、表示している飲食店と同一エリア、近い距離でフィルタすることで、よりユーザーの行動意欲を高めることができます。

また、スコア計算時に求めたデモグラフィック情報を考慮したスコアで並び替えたり、新規性や多様性、意外性などを考慮したリストに並び替えることで、高い効果を期待することも可能です。

なお、既に購入済みの商品をレコメンドしても、購入を繰り返す消耗品でなければ、効果は期待できません。購入済みの商品を除外することで、より多くの発見を提供できます。

まとめ

レコメンドの改善には、スコア計算のロジック変更だけではなく、各フェーズにもさまざまな改善手段があります。少しの操作で内容がガラリと変わるケースもあります。本項に記載しているものはあくまでも一例に過ぎません。導入して効果がないと諦めるのではなく、さまざまな工夫を施し、改善していきましょう。

8-3-5 表示時におけるポイント

前項では、レコメンドシステムでのデータ収集から返却に至るまでのシーンで、可能な改善手段を説明していますが、レコメンドリストの表示側にも改善の余地はあります。

表示するページ、位置、タイミングを検討する

各種のサービスでレコメンドモジュールが表示されるページとその位置を比較すると、意外な発見があります。レコメンドモジュールが閲覧商品の詳細情報よりも上部に表示されているサービスは、より多くの閲覧でより多くの選択肢を提供する考えの現れでしょう。また、ページ下部に配置されている場合は、購入行為を毀損せずにユーザーの離脱を回避することが第一目的でしょう。

例えば、カートへの商品追加後にカート中の商品を閲覧するページへ遷移した場合、カート内一覧画面に表示する適切なレコメンドを考えてみましょう。

閲覧ログを中心に作成したレコメンドリストでは、類似する色や形、サイズの商品がレコメンドされますが、似通った商品を2つも購入するシーンは少ないはずです。ここでは閲覧ログではなく購入データをベースにして、一緒に購入されている商品、一緒に利用する可能性がある商品をレコメンドする方が効果的です。どこのページに何を表示するかも、レコメンドの重要な要素です。

レコメンドリストの根拠を伝える

単純に「オススメ商品」のタイトルでアイテムをレコメンドするのではなく、「この商品を買った人はこんな商品も買っています」、「あなたが購入した商品〜〜に基づくオススメ」と表示することで、レコ

メンドリストの根拠をユーザーに伝えることができ、より効果が上がる場合があります。

また、知人が評価しているものをより評価する傾向（認知バイアス）や、より多くの人が良いと評価するものを良いと判断する傾向（バンドワゴン効果）があることから、ソーシャルメディアでは、上記の情報を表示してレコメンドするモジュールも存在します。

クロスセルを行いやすくする工夫をする

頻繁に同時購入される商品や関連性が極めて高い商品であれば、単にレコメンドするのではなく、「両商品をカートに追加する」ボタンを設置することで、より購入される場合があります。

サービスに合わせて表示する

例えば、購入金額5,000円以上で送料無料のサービスがある場合、現在カート内の金額が5,000円未満の際には、「下記商品を購入すれば送料無料」と表示することで、同時購入で送料無料の5,000円以上になるように、商品価格をフィルタしてクロスセルを狙うことも、より高い効果を得る改善案と考えられます。

まとめ

本項でまとめた通り、レコメンドはロジックの改善だけではなく、レコメンドモジュールを表示する側で実施できる改善も数多くあります。

8-3-6 レコメンドに関する指標を知る

レコメンドシステムの構築時には、どのような指標を確認すればよいでしょうか。Webサイトでの CTR、CVR、レコメンド経由の売上を集計することも1つの手段です。ただし、この方法ではロジックに関する指標がないため、レコメンドシステムの改善に何を考慮に入れれば良いのか不明確で、手探りとなってしまいます。

Microsoft Researchの技術レポート、Shani Guyらによる「Evaluating Recommender Systems（2009）」※に、レコメンドシステムを構築する際の評価方法や評価指標が詳細にまとめられています。オンラインで無料でアクセスできるため、興味のある場合は参照ください。次表に、レポートで取り上げられているレコメンドの評価指標を抜粋して解説を加えています。

レコメンドを実際にサービスとして提供する前に、あらかじめ評価できる指標は、必ず事前に確認しておくことをおすすめします。

※ https://www.microsoft.com/en-us/research/publication/evaluating-recommender-systems/

表8.3.6.1：レコメンドシステムの代表的な評価指標一覧

指標名	説明
Coverage	全体のユーザー数やアイテム数に対して、レコメンドが提供できているカバー率。
Confidence	(システム的な)レコメンドアイテムの信頼度。 あるアイテムをレコメンドするために使用したデータ量が多ければ、Confidenceも高くなる。
Trust	(ユーザー的な)レコメンドアイテムの信頼度。 基本的にユーザーテストするしかない。
Novelty	レコメンドされるアイテムの新規性。
Serendipity	発見的なアイテムがレコメンドされているか。
Diversity	レコメンドされるアイテムの多様性。
Utility	サービスに対するレコメンドの有益性。
Risk	リスクを含むアイテム(株など)のレコメンドをする場合に考慮が必要。
Robustness	DOS攻撃などがあった場合にレコメンド内容が変にならないか。
Privacy	レコメンド内容から個人情報が推測できるような内容が出ていないか。
Adaptivity	アイテムの更新や流行の変化に対して対応できるか。
Scalability	データ量が増えた時に対応できるか。

例えば、上表の「Coverage」(カバー率)は、サービス全体のユーザー数やアイテム数と、レコメンド計算結果でのユーザー数やアイテム数を求められると、計算することが可能です。Coverageが期待値より低い場合はレコメンドアルゴリズムの改善、表示されなかった場合は代替モジュールの設置を検討しましょう。さらに、前述の「8-1-8 検索結果の順位を考慮した指標を計算する」(P.393)で紹介したランキング指標もレコメンドに当てはめることが可能です。

また、レコメンド結果の精度だけでなく、レコメンドへの特定商品の表示など、恣意的な操作がどの程度制御可能か、データ量が増大した際の処理時間の増加予測など、システム的な思考が必要になる場合もあります。評価指標が数多く存在するため、個々の解説は割愛しますが、上記で説明したような視点でレコメンドシステムを構築していきましょう。

関連項目 | 8-1-8 検索結果の順位を考慮した指標を計算する

8-4 スコアの計算

本書では、行動ログやマスタデータなど各種データを用いて、さまざまな値を計算しその結果に基づいて並べ替えを実行する処理を何度も行っています。ある値に基づき並べ替えを実行した結果は「ランキング」と呼ばれ、分析レポートに用いられるだけでなく、検索結果のランキングや、商品のレコメンドなど、さまざまな場面で活用されます。

ランキングで最も重要になるのは、データを並べ替えるための「スコア」をどのように計算するかです。前項までにさまざまな場面に応じて利用できる指標を紹介し、データをスコアリングする方法を実践していますが、本節では、スコアの計算手法をさらに深掘りして、複数の指標を組み合わせたり、値の範囲を変化させるなど、より実践的なスコア計算を紹介します。

8-4-1 複数の値をバランスよく組み合わせてスコアを計算する

SQL SUM(CASE～), ROW_NUMBER関数

分析 相加平均、相乗平均、調和平均、加重平均

データ分析で定量的な判断が必要な際、複数の指標を比較しながら総合的に判断するケースが多々あります。例えば、「8-1-6 検索結果の網羅性を指標化する」(P.384)と「8-1-7 検索結果の妥当性を指標化する」(P.390) で紹介した「再現率」と「適合率」は、一方が高いともう一方が低くなる傾向のトレードオフ関係にあり、両者をバランスよく検討する必要があります。

この他にも、「6-2-5 検索条件毎のユーザー行動を可視化する」(P.279) で紹介した「CTR」と「CVR」、「8-2-1 アソシエーション分析」(P.400) で紹介した「支持度」と「確信度」なども、複数の指標を総合的に判断しなければならない例です。

本項では、複数の値をバランスよく組み合わせて1つの値に集約し、比較検討を容易にする加工手法を紹介します。「8-1-6検索結果の網羅性を指標化する」(P.384) と「8-1-7 検索結果の妥当性を指標化する」(P.390) で紹介した、「再現率」と「適合率」を組み合わせて、検索エンジンの精度を定量的に評価する例を示します。

サンプルデータは、検索エンジンのパスごとに計算した再現率と適合率の一覧を使用します。データの保持方法は、次に示すデータ例の通り、列持ちと行持ちの2種類を取り扱います。

データ8.4.1.1：列持ちで保存した再現率と適合率のデータ

```
   path   | recall | precision
----------+--------+-----------
 /search1 |   40.0 |      60.0
 /search2 |   60.0 |      40.0
 /search3 |   50.0 |      50.0
 /search4 |   30.0 |      60.0
 /search5 |   70.0 |       0.0
```

データ8.4.1.2：行持ちで保存した再現率と適合率のデータ

```
   path   |   index   | value
----------+-----------+-------
 /search1 | recall    |  40.0
 /search1 | precision |  60.0
 /search2 | recall    |  60.0
 /search2 | precision |  40.0
 /search3 | recall    |  50.0
 /search3 | precision |  50.0
 /search4 | recall    |  30.0
 /search4 | precision |  60.0
 /search5 | recall    |  70.0
 /search5 | precision |   0.0
```

3種類の平均

複数の値を組み合わせて集約した値を計算する方法に平均がありますが、一口に平均といってもいくつかの種類があります。本項では一般的によく利用される3種類、「相加平均」と「相乗平均」、「調和平均」を紹介します。

相加平均

相加平均は算術平均とも呼ばれ、各値を足し算して個数で割った値を意味します。一般的に「平均」として呼ばれるものはこの相加平均です。

相乗平均

相乗平均は幾何平均とも呼ばれ、各値を掛け算して個数のn乗根を取った値を意味します。n乗根を取るため、相乗平均を実数として求めるためには、各値は正の数である必要があります。

相乗平均がよく利用されるシーンとして、成長率や利率の平均が挙げられます。例えば、100万円を銀行Aに預金した場合、初年度は3%、2年目は1%の利子が付き、銀行Bでは一定の2%の利子が付くと仮定します。ここで2年間お金を銀行に預けるとしたら、どちらの銀行が得でしょうか。

相加平均で利子の平均を求めると、どちらの銀行も2%であり、一見違いはないように見えます。し

かし、実際に2年後の金額を計算すると、銀行Aは100万円×1.03×1.01=104.03万円、銀行Bでは100万円×1.02×1.02=104.04万円となり、銀行Bが僅かに得です。

この例で利子の相乗平均を求めると、銀行Aが（1.03×1.01）^（1÷2）＝1.01995…となる一方、銀行Bは（1.02×1.02）^（1÷2）＝1.02と、銀行Bの平均利子が高いことが分かります。この通り、ある指標を何度も掛け算する場面では、相加平均より相乗平均が適しています。

調和平均

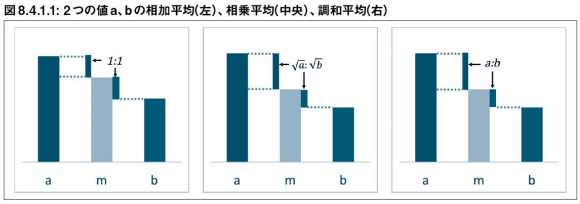
調和平均は各値の逆数の相加平均を計算し、その逆数を取った値を意味します。逆数を取る必要があるため、各値は0でない数である必要があります。馴染みのない計算ですが、割合を意味する値の平均を算出するケースに使用されます。

頻出する例は平均速度の計算です。速度は移動する道程を要する時間で割った割合です。例えば、往路を時速40kmで復路を時速60kmで移動した際の平均速度を求めてみましょう。仮に相加平均を取った「時速50km」は間違いで、正解は「時速48km」です。

極端な例として、東京から広島までの往路は、新幹線を使い時速200kmで4時間かけて移動し、復路は徒歩（時速4km）で10日間かけて移動したとき、相加平均での平均速度は（200＋4）÷2＝102と、明らかに相加平均ではうまく行かないことが分かります。新幹線での移動時間よりも徒歩での移動時間が遙かに長いため、平均速度も徒歩での移動速度に近い値となります。

2値の相加平均、相乗平均、調和平均の値をそれぞれ図示したものを下図に示します。

図 8.4.1.1: 2つの値a、bの相加平均(左)、相乗平均(中央)、調和平均(右)



平均の計算

前述の再現率と適合率のデータを例に、相加平均、相乗平均、調和平均の各値を計算しましょう。

まず、下記コード例に、列持ちのデータに対して平均を計算するクエリを示します。相乗平均や調和平均を同時に求めるため、WHERE句で再現率と適合率のいずれも0ではない、即ち両者の積が0より

大きいものに限定します。

相乗平均の計算は、SQRT関数などを使って平方根を取っても構いませんが、3値以上の相乗平均にも対応させるため、POWER関数を用いて1/2の累乗として計算します。

コード8.4.1.1：列持ちのデータに対して平均を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    *
    -- 相加平均
  , (recall + precision) / 2 AS arithmetic_mean
    -- 相乗平均
  , POWER(recall * precision, 1.0 / 2) AS geometric_mean
    -- 調和平均
  , 2.0 / ((1.0 / recall) + (1.0 / precision)) AS harmonic_mean
FROM
  search_evaluation_by_col
  -- 値が0より大きいものに限定する
WHERE recall * precision > 0
ORDER BY path
;
```

```
   path   | recall | precision | arithmetic_mean | geometric_mean | harmonic_mean
---------+--------+-----------+-----------------+----------------+----------------
 /search1 |   40.0 |      60.0 |         50.0000 |        48.9897 |       47.9999
 /search2 |   60.0 |      40.0 |         50.0000 |        48.9897 |       47.9999
 /search3 |   50.0 |      50.0 |         50.0000 |        50.0000 |       50.0000
 /search4 |   30.0 |      60.0 |         45.0000 |        42.4264 |       40.0000
```

上記コード例の出力結果を確認すると、再現率と適合率が共に50.0で同じ値のとき、すべての平均値も50.0となり、両者が異なる値の場合は、「相加平均>相乗平均>調和平均」の大小関係があることが分かります。

引き続き、行持ちのデータに対して相加平均と相乗平均、調和平均の各値を求めます。次に示すコード8.4.1.2に、行持ちのデータに対して平均を計算するクエリのコード例を示します。

値が0となるpathを除外するため、WHERE句でvalueが0より大きいものに限定し、HAVING句で値の欠損がないpathに限定します。

相加平均は一般的な集約関数のAVGを用いるだけでよく、調和平均もAVGと逆数の計算を組み合わせるだけです。一方、複数レコードの値を掛け算する集約関数は一般的に標準で用意されていないため、相乗平均の計算には工夫が必要です。

ここで、対数（log）を利用すると、値の掛け算を対数の足し算に変形できることを思い出しましょう。対数を導入すると、相乗平均を対数の相加平均を用いて表すことが可能です。

例えば、2と8の相乗平均は、任意の底bを用いて、下式の通りに計算できます。

$$b^{\frac{\log_b(2)+\log_b(8)}{2}}$$

上述の通り、各値の常用対数をlog関数で求め、その相加平均をAVGで計算し、最後に10の指数関数に対数の相加平均を適用することで、相乗平均を簡単に求めることが可能です。

コード8.4.1.2：行持ちのデータに対して平均を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    path
  -- 相加平均
  , AVG(value) AS arithmetic_mean
  -- 相乗平均(対数の相加平均)
  -- ■ PostgreSQL, Redshiftの場合、常用対数にlog関数を用いる
  , POWER(10, AVG(log(value))) AS geometric_mean
  -- ■ Hive, BigQuery, SparkSQLの場合、常用対数にlog10関数を用いる
  -- , POWER(10, AVG(log10(value))) AS geometric_mean
  -- 調和平均
  , 1.0 / (AVG(1.0 / value)) AS harmonic_mean
FROM
  search_evaluation_by_row
  -- 値が0より大きいものに限定する
WHERE value > 0
GROUP BY path
  -- 値の欠損がないpathに限定する
HAVING COUNT(*) = 2
ORDER BY path
;
```

```
   path   | arithmetic_mean | geometric_mean | harmonic_mean
----------+-----------------+----------------+---------------
 /search1 |         50.0000 |        48.9897 |       47.9999
 /search2 |         50.0000 |        48.9897 |       47.9999
 /search3 |         50.0000 |        49.9999 |       50.0000
 /search4 |         45.0000 |        42.4264 |       40.0000
```

上記コード例の出力結果を確認すると、列持ちデータと同様の結果を得られることが分かります。

なお、再現率と適合率の調和平均は、両者を総合的に評価するためによく利用され、「F尺度」（F-measure）や「F1スコア」などと呼ばれています。詳細は「表8.1.8.1.検索評価に用いられる代表的なランキング指標」（P.397～398）を参照してください。

重み付き平均

ここまでの例では、再現率と適合率を平等に扱って平均を求めていますが、シチュエーションによっては、一方に重み付けした値を必要とする場合もあります。例えば、検索結果の網羅性を考慮しつつも、検索結果上位の妥当性を重視して検索エンジンを評価したいため、再現率と適合率を3:7の割合で組合せたいケースです。

相加平均の場合は、各値に任意の重みを掛け算した合計を、重みの合計で割ることで、重み付き相加平均を求めることが可能です。相乗平均の場合、各値に任意の重みを累乗し、各値の積に対して重みの合計の冪根（累乗根）を取ることで、重み付き相乗平均が得られます。また、調和平均の場合、各値の逆数の重み付き相加平均を計算して、その逆数を取ることで、重み付き調和平均を算出できます。

定義上、重みには任意の値を用いることが可能ですが、クエリでの計算に当たり、重みの合計が1.0となるように調整すれば、重みの合計での割り算や冪根の計算を省略できます。下記のコード例に、列持ちのデータに対して重み付き平均を計算するクエリを示します。コード例では、重みの合計を1.0にするため、再現率の重みを0.3、適合率の重みを0.7に設定しています。

コード8.4.1.3：列持ちのデータに対して重み付き平均を計算するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
SELECT
    *
    -- 重み付き相加平均
  , 0.3 * recall + 0.7 * precision AS weighted_arithmetic_mean
    -- 重み付き相乗平均
  , POWER(recall, 0.3) * POWER(precision, 0.7) AS geometric_mean
    -- 重み付き調和平均
  , 1.0 / ((0.3 / recall) + (0.7 / precision)) AS harmonic_mean
FROM
  search_evaluation_by_col
  -- 値が0より大きいものに限定する
WHERE recall * precision > 0
ORDER BY path
;
```

```
   path   | recall | precision | weighted_a_mean | weighted_g_mean | weighted_h_mean
---------+--------+-----------+-----------------+-----------------+-----------------
 /search1 |   40.0 |      60.0 |         54.0000 |         53.1280 |         52.1739
 /search2 |   60.0 |      40.0 |         46.0000 |         45.1738 |         44.4444
 /search3 |   50.0 |      50.0 |         50.0000 |         49.9999 |         50.0000
 /search4 |   30.0 |      60.0 |         51.0000 |         48.7351 |         46.1538
```

上記コード例の出力結果に、重み付きの相加平均、相乗平均、調和平均が求められています。重みを付けない平均に比べ、適合率側に値が偏っていることが分かります。

続いて、下記コード例に、行持ちのデータに対して重み付き平均を計算するクエリを示します。

重みをSELECT句のCASE式ですべて表現することも可能ですが、冗長なクエリとなるため、一時テーブル（weights）を用いて、重みのマスタデータを用意し、各値と結合する方法を採用しています。

なお、ここでAVG関数を用いると、レコードの行数で平均を取ってしまうため、代わりにSUM関数を用いている点に注意してください。

コード8.4.1.4：行持ちのデータに対して重み付き平均を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
weights AS (
  -- 重みのマスタテーブル(重みの合計が1.0になるよう調整する)
           SELECT 'recall'    AS index, 0.3 AS weight
  UNION ALL SELECT 'precision' AS index, 0.7 AS weight
)
SELECT
    e.path
  -- 重み付き相加平均
  , SUM(w.weight * e.value) AS weighted_a_mean
  -- 重み付き相乗平均
  -- ■ PostgreSQL, Redshiftの場合、常用対数にlog関数を用いる
  , POWER(10, SUM(w.weight * log(e.value))) AS weighted_g_mean
  -- ■ Hive, BigQuery, SparkSQLの場合、常用対数にlog10関数を用いる
  -- , POWER(10, SUM(w.weight * log10(e.value))) AS weighted_g_mean

  -- 重み付き調和平均
  , 1.0 / (SUM(w.weight / e.value)) AS weighted_h_mean
FROM
    search_evaluation_by_row AS e
  JOIN
    weights AS w
    ON e.index = w.index
  -- 値が0より大きいものに限定する
WHERE e.value > 0
GROUP BY e.path
  -- 値の欠損がないpathに限定する
HAVING COUNT(*) = 2
ORDER BY e.path
;
```

```
   path   | weighted_a_mean | weighted_g_mean | weighted_h_mean
----------+-----------------+-----------------+-----------------
 /search1 |         54.0000 |         53.1280 |         52.1739
 /search2 |         46.0000 |         45.1738 |         44.4444
 /search3 |         50.0000 |         49.9999 |         50.0000
 /search4 |         51.0000 |         48.7351 |         46.1538
```

まとめ

本項での紹介例は、2つの値に対する相加平均、相乗平均、調和平均ですが、3つ以上の値に対しても、同様のクエリで平均を求めることが可能です。紹介した3つの平均を使いこなし、各種の指標を対象にしてデータ活用に役立ててください。

8-4-2 値の範囲が異なる指標を正規化して比較可能な状態にする

SQL MINウィンドウ関数、MAXウィンドウ関数　　分析 Min-Max正規化、シグモイド関数

前項では、複数の値を組み合わせて総合的な値を評価する手法を紹介しましたが、これはすべての値が同じスケールに収まっていることが前提です。しかし、実際には値の範囲や分布が異なり、単純な組み合わせが困難なケースも多数あります。

例えば、「8-2-1 アソシエーション分析」(P.400) で紹介した指標では、「支持度」「確信度」は0から100の値ですが、「リフト」は0以上の実数で1.0前後の値が多く出現します。また、「8-3-2 このアイテムに興味がある人はこんなアイテムも見ています」(P.410) の閲覧数と購入数も、一般的には購入数は0または1であり、閲覧数は0以上の整数です。この通り、値のスケールが大きく違う指標を組み合わせるには、「正規化」と呼ばれる前処理が必要になる場合があります。

本項では、下記データ例に示す、商品の閲覧数と購入数のデータを元に、各値を正規化して取り扱う手法を紹介します。商品の購入数は0または1であるのに対して、閲覧数は21や49と比較的大きな整数です。値の幅が異なる指標を組み合わせるには、両者の値を同じスケールで扱えるように正規化することが有効であるケースが多いです。例えば、閲覧数と購入数の値を、0から1までの実数で表現することを考えてみましょう。

データ8.4.2.1：商品の閲覧数と購入数(action_counts)テーブル

user_id	product	view_count	purchase_count
U001	D001	2	1
U001	D002	16	0
U001	D003	14	0
U001	D004	15	0
U001	D005	21	1
U002	D001	10	1
U002	D003	28	0
U002	D005	28	1
U003	D001	49	0
U003	D004	29	1
U003	D005	24	1

Min-Max正規化

異なる値の幅を持つ各指標を、0～1のスケールに正規化する方法として、「Min-Max正規化」があります。Min-Max正規化は、指標の最小値と最大値を求め、変換後の最小値が0、最大値が1になるように値を変形します。変換後の値は、各値から指標の最小値を引き算し、最大値と最小値の差で割った値になります。

下記コード例に、閲覧数と購入数にMin-Max正規化を適用するクエリを示します。レコード全体の最小値や最大値を求めるために、MINウィンドウ関数やMAXウィンドウ関数を用いています。

コード8.4.2.1：閲覧数と購入数にMin-Max正規化を適用するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id
  , product
  , view_count AS v_count
  , purchase_count AS p_count
  , 1.0 * (view_count - MIN(view_count) OVER())
    -- ■ PostgreSQL, Redshift, BigQuery, SparkSQLの場合､NULLIFで0除算を避ける
    / NULLIF((MAX(view_count) OVER() - MIN(view_count) OVER()), 0)
    -- ■ Hiveの場合､NULLIFの代わりにCASE式を用いる
    -- / (CASE
    --     WHEN MAX(view_count) OVER() - MIN(view_count) OVER() = 0 THEN NULL
    --     ELSE MAX(view_count) OVER() - MIN(view_count) OVER()
    --    END)
    AS norm_v_count
  , 1.0 * (purchase_count - MIN(purchase_count) OVER())
    -- ■ PostgreSQL, Redshift, BigQuery, SparkSQLの場合､NULLIFで0除算を避ける
    / NULLIF((MAX(purchase_count) OVER() - MIN(purchase_count) OVER()), 0)
    -- ■ Hiveの場合､NULLIFの代わりにCASE式を用いる
    -- / (CASE
    --     WHEN MAX(purchase_count) OVER() - MIN(purchase_count) OVER() = 0 THEN NULL
    --     ELSE MAX(purchase_count) OVER() - MIN(purchase_count) OVER()
    --    END)
    AS norm_p_count
FROM action_counts
ORDER BY user_id, product;
```

```
 user_id | product | v_count | p_count | norm_v_count | norm_p_count
---------+---------+---------+---------+--------------+--------------
 U001    | D001    |       2 |       1 |      0.00000 |      1.00000
 U001    | D002    |      16 |       0 |      0.29787 |      0.00000
 U001    | D003    |      14 |       0 |      0.25531 |      0.00000
 U001    | D004    |      15 |       0 |      0.27659 |      0.00000
 U001    | D005    |      21 |       1 |      0.40425 |      1.00000
 U002    | D001    |      10 |       1 |      0.17021 |      1.00000
```

```
U002    | D003    |      28 |       0 |      0.55319 |      0.00000
U002    | D005    |      28 |       1 |      0.55319 |      1.00000
U003    | D001    |      49 |       0 |      1.00000 |      0.00000
U003    | D004    |      29 |       1 |      0.57446 |      1.00000
U003    | D005    |      24 |       1 |      0.46808 |      1.00000
```

上記コード例の出力結果で、norm_v_countが正規化後の閲覧数、norm_p_countが正規化後の購入になります。購入数は元の値の最小値が0、最大値が1であるため、正規化後も値が変わっていません。

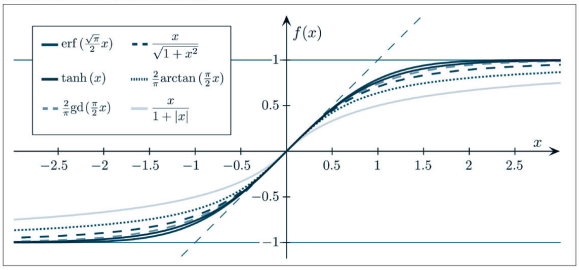
一方、閲覧数は、元々の最小値が2で最大値が49だったので、変換後は最小値と最大値が0と1になり、他のレコードは0～1に収まっていることが分かります。閲覧数と購入数のスケールを正規化で揃えることで、両者を比較・統合することが容易になります。

シグモイド関数による変換

前述のMin-Max正規化で、異なる指標のスケールを揃えることは可能ですが、いくつかの課題も残されています。1つは母集団の変化に伴い正規化後の値が変化することです。例えば、前述の閲覧数の例では、最大値が50のケースと100のケースで、閲覧数10のレコードの値が0.2や0.1になってしまいます。また、正規化の計算では最小値と最大値の値を求める必要があるため、何百億レコードものデータの正規化には、計算コストが掛かってしまいます。

上記の課題に対処するには、Min-Max正規化などの母集団の統計量を用いる正規化ではなく、値域が0～1に収まる特定の関数を用いてデータを変形する方法が考えられます。本項では、0以上の値を0～1の値に収めるため、シグモイド曲線を描く関数を利用します。シグモイド曲線とは、図8.4.2.1に示すS字型のカーブを描く曲線で、シグモイド曲線を描く関数はいくつか存在します。

図8.4.2.1シグモイド曲線を描く関数群※



※ https://commons.wikimedia.org/wiki/File:Gjl-t(x).svg

特に、「y = 1 / (1 + e^(-ax))」の関数がよく利用され、a=1の関数を標準シグモイド関数と呼びます。

コード例では、このシグモイド関数を、x=0のときにy=0、x=∞のときにy=1となるように変形して、パラメーター「a」の値（ゲインと呼ばれます）を変更することで、各指標の重みを調整します。

下記コード例に、シグモイド関数を用いた変形を行うクエリを示します。閲覧数に適用する関数はゲイン0.1、購入数に適用する関数はゲイン10に設定します。ゲインの値を大きくすることで、入力の値が少し大きくなっただけで、変換後の値も1に近付きやすくなります。

コード8.4.2.2：シグモイド関数を用いた変形を行うクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
SELECT
    user_id
  , product
  , view_count AS v_count
  , purchase_count AS p_count
    -- ゲイン0.1のシグモイド関数
  , 2.0 / (1 + exp( -0.1 * view_count)) - 1.0 AS sigm_v_count
    -- ゲイン10のシグモイド関数
  , 2.0 / (1 + exp(-10.0 * purchase_count)) - 1.0 AS sigm_p_count
FROM action_counts
ORDER BY user_id, product;
```

```
 user_id | product | v_count | p_count | sigm_v_count | sigm_p_count
---------+---------+---------+---------+--------------+--------------
 U001    | D001    |       2 |       1 |       0.0996 |       0.9999
 U001    | D002    |      16 |       0 |       0.6640 |       0.0000
 U001    | D003    |      14 |       0 |       0.6043 |       0.0000
 U001    | D004    |      15 |       0 |       0.6351 |       0.0000
 U001    | D005    |      21 |       1 |       0.7818 |       0.9999
 U002    | D001    |      10 |       1 |       0.4621 |       0.9999
 U002    | D003    |      28 |       0 |       0.8853 |       0.0000
 U002    | D005    |      28 |       1 |       0.8853 |       0.9999
 U003    | D001    |      49 |       0 |       0.9852 |       0.0000
 U003    | D004    |      29 |       1 |       0.8956 |       0.9999
 U003    | D005    |      24 |       1 |       0.8336 |       0.9999
```

上記コード例の出力結果に、シグモイド関数による変形後の閲覧数と購入数が求められています。購入数はゲイン10と大きくしているので、元の0～1の値とほぼ変わりません。閲覧数も、閲覧数が10のときに0.4621、49のときに0.9852と、0～1の範囲に収まっています。

Min-Max正規化とは異なり、元の閲覧数が2倍になれば変換後の値も2倍になる、線形の関係ではないことに着目してください。シグモイド関数を用いた場合、グラフが曲線になることから分かる通り、元の値が0に近いほど変換後の値への影響が大きく、元の値が大きければ、変換後の値はほぼ1に収束します。

この特徴は、閲覧数（ユーザーのアクション数）を取り扱うには直感的で都合が良いです。例えば、

閲覧数が1回の商品と10回の商品では、その商品への関心度はかなりの乖離がありますが、閲覧数が1001回の商品と1010回の商品では、ほとんど関心度に差はないと判断できます。したがって、シグモイド関数の非線形な変換を行うことで、人間の直観に近い形で指標を正規化できます。

まとめ

値の範囲（スケール）が異なる指標を取り扱う方法として、値の母集団に応じて正規化する手法と、特定の関数を適用して正規化する手法を紹介しています。いずれの方法が適しているかは、状況やデータの性質によってさまざまです。いろいろな応用分野で試してみてください。

関連項目 | 3-2-3 2つの値の比率を計算する

8-4-3 各データの偏差値を計算する

SQL stddev_pop関数　　分析 標準偏差、基準値、偏差値

前項では、値の範囲が異なる指標を正規化する方法を紹介しましたが、データの分布が正規分布に近いことが予め予測できる場合は、基準値や偏差値を用いることが多くあります。本項では、一般的な偏差値の定義と、SQLによる計算方法を紹介します。

標準偏差、基準値、偏差値

偏差値を扱う前に、標準偏差と基準値に関して説明します。

標準偏差

標準偏差は、データのばらつき具合を表す数値の1つ。標準偏差は下記の式で求めることができます。

$$\text{標準偏差} = \sqrt{\frac{(\text{個々のデータ} - \text{平均})^2\text{を足したもの}}{\text{データの個数}}} \quad \text{または} \quad \sqrt{\frac{(\text{個々のデータ} - \text{平均})^2\text{を足したもの}}{\text{データの個数-1}}}$$

前者の式は母集団の標準偏差を求める際に使用して、後者は母集団から一部を取り出した標本の標準偏差を求める際に使用します。標準偏差は最小値が0であり、データのばらつき具合が大きいほど、大きな値になります。SQLの関数では前者をstddev_pop関数で、後者をstddev関数で計算できます。

例えば、下表に示す国語と数学のテストの点数をまとめたものを確認すると、数学の方が点数にばらつきが見られ、標準偏差が大きくなっていることがわかります。

表8.4.3.1：データのばらつきの違いによる標準偏差

	生徒A	生徒B	生徒C	生徒D	生徒E	標準偏差
国語	69	87	65	73	61	8.94427191
数学	100	12	7	73	56	35.65725733

基準値

平均からの離れ具合やデータのばらつきの程度を元に、点数の価値を検討しやすくするデータ変換を基準化と呼びます。基準化されたデータを基準値と呼びます。基準値は下記の式で求められます。

$$\text{基準値} = \frac{(\text{個々のデータ}) - (\text{平均})}{\text{標準偏差}}$$

上表の国語と数学の点数を基準化した状態を下表に示します。生徒Dは国語と数学で同じ点数を取っていても、基準値で見ると異なることが分かります。

表8.4.3.2：表8.4.3.1を基準化した状態

	生徒A	生徒B	生徒C	生徒D	生徒E	平均	標準偏差
国語	-0.22360..	1.788854..	-0.670820..	0.223606..	-1.118033..	0	1
数学	1.413457..	-1.054483..	-1.194707..	0.656247..	0.179486..	0	1

基準化で求めた「基準値の平均は必ず0で、標準偏差は必ず1」となる特徴があります。この特徴から、異なる範囲のデータ（例：100点満点と200点満点のテスト）であっても、異なる単位のデータ（例：CTRとCVR）であっても比較が可能です。

偏差値

偏差値は条件の異なるデータを容易に比較する際に使われ、基準値を用いて計算します。偏差値は下記の式で求められます。

$$\text{偏差値} = (\text{個々のデータ}) \times 10 + 50$$

前述の国語と数学の点数を偏差値で示した状態が次表です。生徒Dは国語と数学で同じ点数を取っていても、基準値が異なるため当然、偏差値も異なることが分かります。また、基準値を元に計算しているため、「偏差値の平均は必ず50で、標準偏差は必ず10」となる特徴があります。

表8.4.3.3：偏差値

	生徒A	生徒B	生徒C	生徒D	生徒E	平均	標準偏差
国語	47.76393..	67.88854..	43.29179..	52.23606..	38.81966..	50	10
数学	64.13457..	39.45516..	38.05292..	56.56247..	51.79486..	50	10

偏差値を計算する

偏差値は基準値の定義と組み合わせると、「10×（個々の値－平均値）÷標準偏差＋50」として計算できます。個々のデータに対して平均値や標準偏差を用いて計算するため、ウィンドウ関数を用いて平均値や標準偏差を求める必要があります。下記コード例に、ウィンドウ関数を用いて教科ごとの標準偏差と基準値／偏差値を計算するクエリを示します。

コード8.4.3.1：標準偏差と基準値、偏差値を計算するクエリ

PostgreSQL Hive Redshift SparkSQL

```
SELECT
    subject
  , name
  , score
  -- 教科ごとの標準偏差を求める
  , stddev_pop(score) OVER(PARTITION BY subject) AS stddev_pop
  -- 教科ごとの平均点を求める
  , AVG(score) OVER(PARTITION BY subject) AS avg_score
  -- スコアごとの基準値を求める
  , (score - AVG(score) OVER(PARTITION BY subject))
    / stddev_pop(score) OVER(PARTITION BY subject)
    AS std_value
  -- スコアごとの偏差値を求める
  , 10.0 * (score - AVG(score) OVER(PARTITION BY subject))
    / stddev_pop(score) OVER(PARTITION BY subject)
    + 50
    AS deviation
FROM exam_scores
ORDER BY subject, name;
```

```
 subject | name  | score | stddev_pop | avg_score | std_value | deviation
---------+-------+-------+------------+-----------+-----------+-----------
  国語    | 生徒A  |    69 |     8.9442 |   71.0000 |   -0.2236 |   47.7639
  国語    | 生徒B  |    87 |     8.9442 |   71.0000 |    1.7888 |   67.8885
  国語    | 生徒C  |    65 |     8.9442 |   71.0000 |   -0.6708 |   43.2917
…
  数学    | 生徒B  |    12 |    35.6572 |   49.6000 |   -1.0544 |   39.4551
  数学    | 生徒C  |     7 |    35.6572 |   49.6000 |   -1.1947 |   38.0529
  数学    | 生徒D  |    73 |    35.6572 |   49.6000 |    0.6562 |   56.5624
  数学    | 生徒E  |    56 |    35.6572 |   49.6000 |    0.1794 |   51.7948
```

別解 BigQueryの場合は、stddev_pop関数をウィンドウ関数として使用できないため、別テーブルに標準偏差を計算しておき、各レコードと結合する方針を取ります。

コード8.4.3.2：標準偏差を別に計算し、基準値、偏差値を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
exam_stddev_pop AS (
  -- 別テーブルで教科ごとの標準偏差を求める
  SELECT
      subject
    , stddev_pop(score) AS stddev_pop
  FROM exam_scores
  GROUP BY subject
)
SELECT
    s.subject
  , s.name
  , s.score
  , d.stddev_pop
  , AVG(s.score) OVER(PARTITION BY s.subject) AS avg_score
  , (s.score - AVG(s.score) OVER(PARTITION BY s.subject))
    / d.stddev_pop
    AS std_value
  , 10.0 * (s.score - AVG(s.score) OVER(PARTITION BY s.subject))
    / d.stddev_pop
    + 50
    AS deviation
FROM
    exam_scores AS s
  JOIN
    exam_stddev_pop AS d
    ON s.subject = d.subject
ORDER BY s.subject, s.name;
```

まとめ

基準値や偏差値などの統計指標をSQLで計算する方法は、ウィンドウ関数の登場のおかげで格段に容易となりました。しかし、定義の分からないまま用意された関数を利用するだけでは、意味のあるデータ分析はできません。まずは指標の定義や意味をしっかりと確認して、データ分析に役立ててください。

関連項目 | 3-3-1 グループの特徴を捉える

8-4-4 巨大な数値の指標を直観的に分かりやすく加工する

SQL log関数　　分析 対数

「8-4-2 値の範囲が異なる指標を正規化して比較可能な状態にする」(P.434) では、シグモイド関数を用いて閲覧数を0～1の値に変形し、非線形な変換を行うことで、人間の直感に近いスコアを計算する方法を紹介しました。本項でも、人間の直感に近い値への変形を行う非線形な関数として、対数の応用を紹介します。

数値に関する人間の直感として、差の絶対値が同じでも、元の値が小さい場合（1や2など）と、元の値が大きい場合（10001と10002など）では、元の値が小さい方が、値の差を大きく感じます。ユーザーのアクション数や経過日数など、値が大きくなるほど値の差の重みが小さくなるデータは、対数を用いてデータを変換し、人間の直感に合ったスコアに加工する手法がよく利用されます。

本項で用いるサンプルデータとして、下記データ例に示す、ユーザーごとの商品の閲覧数（v_count）と購入数（p_count）をアクション日別に集計したものを用います。

データ8.4.4.1：日付別のユーザーのアクション数(action_counts_with_date)テーブル

```
     dt     | user_id | product | v_count | p_count
------------+---------+---------+---------+---------
 2016-10-03 | U001    | D001    |       1 |       0
 2016-11-03 | U001    | D001    |       1 |       1
 2016-10-03 | U001    | D002    |      16 |       0
 2016-10-03 | U001    | D003    |      14 |       0
 2016-10-03 | U001    | D004    |      15 |       0
 2016-10-03 | U001    | D005    |      19 |       0
 2016-10-25 | U001    | D005    |       1 |       0
 2016-11-03 | U001    | D005    |       1 |       0
 2016-11-05 | U001    | D005    |       0 |       1
 2016-10-03 | U002    | D001    |      10 |       0
 2016-11-30 | U002    | D001    |       0 |       1
 2016-11-20 | U002    | D003    |      28 |       0
 2016-11-20 | U002    | D005    |      28 |       0
 2016-11-30 | U002    | D005    |       0 |       1
 2016-11-01 | U003    | D001    |      49 |       0
 2016-11-01 | U003    | D004    |      29 |       0
 2016-11-03 | U003    | D004    |       0 |       1
...
```

「8-3-2 このアイテムに興味がある人はこんなアイテムも見ています」(P.410) では、すべての期間の閲覧数や購入数を単純に合計して、商品スコアを算出していますが、本項では日付の古いアクションほど重みを小さくする工夫を施します。まずは次ページのコード例に示す通り、ユーザーごとの最終アクセス日と、各レコードとの日数差を計算してみましょう。

コード8.4.4.1：ユーザーごとの最終アクセス日と各レコードとの日数差を計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
action_counts_with_diff_date AS (
  SELECT *
    -- ユーザーごとの最終アクセス日と各レコードとの日数差を計算する
    -- ■ PostgreSQL, Redshiftの場合､日付型同士の引き算ができる
    , MAX(dt::date) OVER(PARTITION BY user_id) AS last_access
    , MAX(dt::date) OVER(PARTITION BY user_id) - dt::date AS diff_date
    -- ■ BigQueryの場合､date_diff関数が利用できる
    -- , MAX(date(timestamp(dt))) OVER(PARTITION BY user_id) AS last_access
    -- , date_diff(MAX(date(timestamp(dt))) OVER(PARTITION BY user_id),
    --     date(timestamp(dt)), day) AS diff_date
    -- ■ Hive, SparkSQLの場合､datediff関数が利用できる
    -- , MAX(to_date(dt)) OVER(PARTITION BY user_id) AS last_access
    -- , datediff(MAX(to_date(dt)) OVER(PARTITION BY user_id), to_date(dt))
    --   AS diff_date
  FROM
    action_counts_with_date
)
SELECT *
FROM action_counts_with_diff_date;
```

```
     dt     | user_id | product | v_count | p_count | last_access | diff_date
------------+---------+---------+---------+---------+-------------+-----------
 2016-10-03 | U001    | D001    |       1 |       0 | 2016-11-05  |        33
 2016-11-03 | U001    | D001    |       1 |       1 | 2016-11-05  |         2
 2016-10-03 | U001    | D002    |      16 |       0 | 2016-11-05  |        33
 2016-10-03 | U001    | D003    |      14 |       0 | 2016-11-05  |        33
 2016-10-03 | U001    | D004    |      15 |       0 | 2016-11-05  |        33
 2016-10-03 | U001    | D005    |      19 |       0 | 2016-11-05  |        33
 2016-10-25 | U001    | D005    |       1 |       0 | 2016-11-05  |        11
 2016-11-03 | U001    | D005    |       1 |       0 | 2016-11-05  |         2
 2016-11-05 | U001    | D005    |       0 |       1 | 2016-11-05  |         0
 2016-10-03 | U002    | D001    |      10 |       0 | 2016-11-30  |        58
 2016-11-30 | U002    | D001    |       0 |       1 | 2016-11-30  |         0
 2016-11-20 | U002    | D003    |      28 |       0 | 2016-11-30  |        10
 2016-11-20 | U002    | D005    |      28 |       0 | 2016-11-30  |        10
 2016-11-30 | U002    | D005    |       0 |       1 | 2016-11-30  |         0
 2016-11-01 | U003    | D001    |      49 |       0 | 2016-12-01  |        30
 2016-11-01 | U003    | D004    |      29 |       0 | 2016-12-01  |        30
 2016-11-03 | U003    | D004    |       0 |       1 | 2016-12-01  |        28
 2016-11-01 | U003    | D005    |      24 |       0 | 2016-12-01  |        30
 2016-12-01 | U003    | D005    |       0 |       1 | 2016-12-01  |         0
```

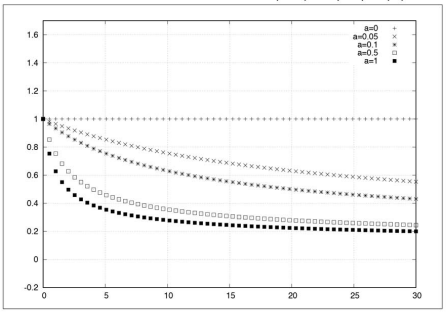
上記コード例の出力結果では、例えば、ユーザーU001の最終アクセス日は2016年11月5日で、商品D001に対するアクションは、最終アクセスの33日前に1度閲覧し、2日前に1度の閲覧と購入を実行しているデータになります。

次のステップ 最終アクセス日との日数差が大きいほど重みを小さくすることを考えると、最初に思い付くことは日数差の逆数を取ることです。しかし、単純に日数差の逆数を取ると、2日前のアクションの重みは1/2、30日前のアクションの重みは1/30と、極端に小さな値となります。そこで、日数差の対数を取り、日数差の増加に対して、重みの減衰を緩やかにする方法を採択しましょう。

また、日数差が0のときに重みが最大の1となり、パラメーターaの値で重みの変化率を調整可能にするため、下記に示す式で重みを定義します。

$$y = \frac{1}{\log_2(ax+2)}\{0 \leq x\}$$

図8.4.4.1:日数差の対数の逆数による重みのグラフ(a=0, 0.05, 0.1, 0.5, 1)



上図の＊でプロットされているグラフがパラメーターa=0.1のグラフです。パラメーターを0に近付けると重みの減り方が緩やかになり、a=0で重みが常に1となります。また、パラメーターを大きくするほど、重みの減り方が激しくなります。

下記コード例に、上記の式を用いて日数差による重みを計算するクエリを示します。

コード8.4.4.2：日数差による重みを計算するクエリ

```
WITH
action_counts_with_diff_date AS (
  -- コード8.4.4.1参照
)
, action_counts_with_weight AS (
```

```
  SELECT *
    -- 日数差による重みの計算(パラメーターa=0.1)
    -- ■ PostgreSQL, Hive, SparkSQLの場合、log(底, 値)関数が使用できる
    , 1.0 / log(2, 0.1 * diff_date + 2) AS weight
    -- ■ Redshiftの場合、log関数は常用対数となるので、底を2に変換するためlog(2)で割る
    -- , 1.0 / ( log(CAST(0.1 * diff_date + 2 AS double precision)) / log(2) )
    -- AS weight
    -- ■ BigQueryの場合、log(値, 底)関数が使用できる
    -- , 1.0 / log(0.1 * diff_date + 2, 2) AS weight
  FROM action_counts_with_diff_date
)
SELECT
    user_id
  , product
  , v_count
  , p_count
  , diff_date
  , weight
FROM action_counts_with_weight
ORDER BY
  user_id, product, diff_date DESC
;
```

user_id	product	v_count	p_count	diff_date	weight
U001	D001	1	0	33	0.41562
U001	D001	1	1	2	0.87911
U001	D002	16	0	33	0.41562
U001	D003	14	0	33	0.41562
U001	D004	15	0	33	0.41562
U001	D005	19	0	33	0.41562
U001	D005	1	0	11	0.61264
U001	D005	1	0	2	0.87911
U001	D005	0	1	0	1.00000
U002	D001	10	0	58	0.33744
U002	D001	0	1	0	1.00000
U002	D003	28	0	10	0.63092
U002	D005	28	0	10	0.63092
U002	D005	0	1	0	1.00000
U003	D001	49	0	30	0.43067
U003	D004	29	0	30	0.43067
U003	D004	0	1	28	0.44188
U003	D005	24	0	30	0.43067
U003	D005	0	1	0	1.00000

上記の出力結果から、最新アクセス日との日数差が0で重みが最大の1、日数差が大きくなるほど重みも小さくなることが分かります。ただし、日数差が大きくなるほど重みの減り方は緩やかになります。

次のステップ この重みを用いて、閲覧数や購入数からスコアを計算するクエリを下記のコード例に示します。日付単位の閲覧数や購入数に対して、該当日の重みを掛け算し、商品ごとにその値を合計してスコアを求めます。

コード8.4.4.3：日数差による重みを用いて閲覧数と購入数のスコアを計算するクエリ

```
WITH
action_counts_with_date AS (
  -- コード8.4.4.1参照
)
, action_counts_with_weight AS (
  -- コード8.4.4.2参照
)
, action_scores AS (
  SELECT
      user_id
    , product
    , SUM(v_count) AS v_count
    , SUM(v_count * weight) AS v_score
    , SUM(p_count) AS p_count
    , SUM(p_count * weight) AS p_score
  FROM action_counts_with_weight
  GROUP BY
     user_id, product
)
SELECT *
FROM action_scores
ORDER BY
  user_id, product;
```

```
 user_id | product | v_count | p_count | v_score  | p_score
---------+---------+---------+---------+----------+---------
 U001    | D001    |       2 |       1 |  1.29474 | 0.87911
 U001    | D002    |      16 |       0 |  6.65006 | 0.00000
 U001    | D003    |      14 |       0 |  5.81880 | 0.00000
 U001    | D004    |      15 |       0 |  6.23443 | 0.00000
 U001    | D005    |      21 |       1 |  9.38871 | 1.00000
 U002    | D001    |      10 |       1 |  3.37441 | 1.00000
 U002    | D003    |      28 |       0 | 17.66603 | 0.00000
 U002    | D005    |      28 |       1 | 17.66603 | 1.00000
 U003    | D001    |      49 |       0 | 21.10315 | 0.00000
 U003    | D004    |      29 |       1 | 12.48962 | 0.44188
 U003    | D005    |      24 |       1 | 10.33623 | 1.00000
```

上記コード例の出力結果から、アクションの日付に応じて重みを加えたスコアを算出可能です。同じ購入数の商品でも購入日が古い商品は、スコアも低くなっていることが分かります。

まとめ

閲覧数や購入数などカウントの値を用いてスコアを計算する際、対数を取ることで値の変化を緩やかに変換し、より人間の直感に合致するスコアを定義する手法を紹介しました。対数はほとんどの処理系で実装されている基本的な関数ですが、その応用範囲は広範囲に渡ります。

関連項目	3-2-5 日付・時刻を計算する

8-4-5 独自のスコアリング方法を定義してランキングを作成する

分析 加重平均

本節のまとめとして、具体的な施策のためのスコア計算例を紹介します。

下記のデータ例は、2016年1月から12月までの商品売上額を月別に集計したものです。このデータを用いて、2017年1月のおすすめ商品ランキングを作成するためのスコアを計算します。

データ8.4.5.1：月別の商品売上額(monthly_sales)テーブル

```
 year_month | item | amount
------------+------+--------
 2016-01    | D001 |  30000
 2016-01    | D002 |  10000
 2016-01    | D003 |   5000
 2016-01    | D004 |   3000
 2016-02    | D001 |  30000
 2016-02    | D002 |  30000
 2016-02    | D005 |  20000
 2016-02    | D003 |  10000
...
 2016-10    | D006 |  30000
 2016-10    | D003 |  30000
 2016-11    | D006 |  60000
 2016-11    | D004 |  40000
 2016-11    | D003 |  40000
 2016-11    | D001 |  40000
 2016-12    | D006 |  70000
 2016-12    | D003 |  60000
 2016-12    | D004 |  50000
 2016-12    | D001 |  40000
```

ランキング作成の方針は、1年間の季節で周期的に売れる商品と、直近のトレンド商品が上位に来るランキングを検討します。そこで、1年前の売上と直近1ヶ月の売上を重み付き平均でスコアリングします。下記コード例に四半期ごとの商品売上額と売上合計を集計するクエリを示します。

ランキングに使用するデータは、商品ごとの2016年の第1期（1月～3月）の売上額と、第4期（10月～12月）の売上額です。

コード8.4.5.1：四半期ごとの商品売上額と売上合計を集計するクエリ

PostgreSQL | Hive | Redshift | BigQuery | SparkSQL

```
WITH
item_sales_per_quarters AS (
  SELECT
      item
      -- 2016年第1期の商品売上を合計する
    , SUM(
        CASE WHEN year_month IN ('2016-01', '2016-02', '2016-03') THEN amount ELSE 0 END
      ) AS sales_2016_q1
      -- 2016年第4期の商品売上を合計する
    , SUM(
        CASE WHEN year_month IN ('2016-10', '2016-11', '2016-12') THEN amount ELSE 0 END
      ) AS sales_2016_q4
  FROM monthly_sales
  GROUP BY
    item
)
SELECT
    item
  -- 2016年第1期の商品売上
  , sales_2016_q1
  -- 2016年第1期の合計売上
  , SUM(sales_2016_q1) OVER() AS sum_sales_2016_q1
  -- 2016年第4期の商品売上
  , sales_2016_q4
  -- 2016年第4期の合計売上
  , SUM(sales_2016_q4) OVER() AS sum_sales_2016_q4
FROM item_sales_per_quarters
;
```

```
 item | sales_2016_q1 | sum_sales_2016_q1 | sales_2016_q4 | sum_sales_2016_q4
------+---------------+-------------------+---------------+-------------------
 D005 |         60000 |            254000 |         50000 |            550000
 D006 |             0 |            254000 |        160000 |            550000
 D002 |         80000 |            254000 |             0 |            550000
 D004 |          3000 |            254000 |        130000 |            550000
 D001 |         80000 |            254000 |         80000 |            550000
 D003 |         31000 |            254000 |        130000 |            550000
```

次のステップ 上記コード例の出力結果で、2016年第1期と第4期の合計売上を確認すると、第1期の254,000に対して第4期が550,000と、大きく差があることが分かります。これはサービス全体の売上が上昇傾向にあることを示します。したがって、第1期の商品売上額と第4期の商品売上額では、同じ金額でもその意味が異なってきます。

下記のコード例に、このような母数が異なる2つの指標を比較可能な状態にするため、「8-4-2 値の範囲が異なる指標を正規化して比較可能な状態にする」(P.434) で紹介したMin-Max正規化を用いて、第1期と第4期の商品売上額を正規化したスコアを計算するクエリを示します。

コード8.4.5.2：四半期ごとの商品売上額からスコアを計算するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
item_sales_per_quarters AS (
  -- コード8.4.5.1参照
)
, item_scores_per_quarters AS (
  SELECT
      item
    , sales_2016_q1
    , 1.0
      * (sales_2016_q1 - MIN(sales_2016_q1) OVER())
      -- ■ PostgreSQL, Redshift, BigQuery, SparkSQLの場合、NULLIFで0除算を避ける
      / NULLIF(MAX(sales_2016_q1) OVER() - MIN(sales_2016_q1) OVER(), 0)
      -- ■ Hiveの場合、NULLIFの代わりにCASE式を用いる
      -- / (CASE
      --     WHEN MAX(sales_2016_q1) OVER() - MIN(sales_2016_q1) OVER() = 0 THEN NULL
      --     ELSE MAX(sales_2016_q1) OVER() - MIN(sales_2016_q1) OVER()
      --   END)
      AS score_2016_q1
    , sales_2016_q4
    , 1.0
      * (sales_2016_q4 - MIN(sales_2016_q4) OVER())
      -- ■ PostgreSQL, Redshift, BigQuery, SparkSQLの場合、NULLIFで0除算を避ける
      / NULLIF(MAX(sales_2016_q4) OVER() - MIN(sales_2016_q4) OVER(), 0)
      -- ■ Hiveの場合、NULLIFの代わりにCASE式を用いる
      -- / (CASE
      --     WHEN MAX(sales_2016_q4) OVER() - MIN(sales_2016_q4) OVER() = 0 THEN NULL
      --     ELSE MAX(sales_2016_q4) OVER() - MIN(sales_2016_q4) OVER()
      --   END)
      AS score_2016_q4
  FROM item_sales_per_quarters
)
SELECT *
FROM item_scores_per_quarters
;
```

```
 item | sales_2016_q1 | score_2016_q1 | sales_2016_q4 | score_2016_q4
------+---------------+---------------+---------------+---------------
 D005 |         60000 |        0.7500 |         50000 |        0.3125
 D006 |             0 |        0.0000 |        160000 |        1.0000
 D002 |         80000 |        1.0000 |             0 |        0.0000
 D004 |          3000 |        0.0375 |        130000 |        0.8125
 D001 |         80000 |        1.0000 |         80000 |        0.5000
 D003 |         31000 |        0.3875 |        130000 |        0.8125
```

次のステップ 第1期と第4期の売上から正規化したスコアを計算したら、両者の重み付き平均を取り、1つのスコアに集約します。第1期と第4期の重みを7:3の割合に設定し、重み付き相加平均を計算してスコアリングします。下記コード例に重み付き平均でランキングを作成するクエリを示します。

コード8.4.5.3：四半期ごとの商品スコアの重み付き平均でランキングを作成するクエリ

PostgreSQL Hive Redshift BigQuery SparkSQL

```
WITH
item_sales_per_quarters AS (
  -- コード8.4.5.1参照
)
, item_scores_per_quarters AS (
  -- コード8.4.5.2参照
)
SELECT
    item
  , 0.7 * score_2016_q1 + 0.3 * score_2016_q4 AS score
  , ROW_NUMBER()
      OVER(ORDER BY 0.7 * score_2016_q1 + 0.3 * score_2016_q4 DESC)
    AS rank
FROM item_scores_per_quarters
ORDER BY rank
;
```

```
 item | score  | rank
------+--------+------
 D001 | 0.8500 |    1
 D002 | 0.7000 |    2
 D005 | 0.6187 |    3
 D003 | 0.5150 |    4
 D006 | 0.3000 |    5
 D004 | 0.2700 |    6
```

まとめ

本項で紹介した手順を踏むことで、周期要素のある商品とトレンド商品をバランスよく組み合わせたランキングが作成可能です。また、各種のスコア計算手法を組み合わせて、多くの分野に応用できます。

Chapter 9

知識に留めず行動を起こす

レポートやSQLの単なる習得に留めず、
実際に行動しサービス改善や利益に貢献し、
実際の業務で役立てる方法を紹介します。

どこから始めたらよいかわからない！
うまくいかない！
そんなときに最初の一歩を踏み出す
きっかけと解決の糸口を
提案します。

9-1 データ活用の現場

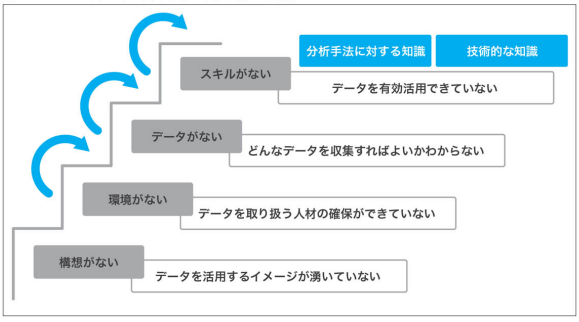
本書では、さまざまなレポート形式の紹介から指標の提案、関連するSQLを説明しています。しかし、分析の前段階で行き詰まったり、既に業務でデータ分析基盤を利用しているにも関わらず、想定通りに進めることができないケースもあるはずです。

ビッグデータに関する悩み

各所より「ビッグデータってどんなことをやるのか？」、「ビッグデータ分析基盤を導入したいので話を聞きたい」、「どうすればデータをうまく活用できるのか？」など、ビッグデータ分析に関わるさまざまな相談が執筆陣に寄せられます。

個々の相談内容はさまざまですが、大きく下記の4つに分類できます。それぞれの分類の悩みの原因を考えてみましょう。

図 9.1.0.1：ビッグデータに関して寄せられる悩みと分類



データを活用するイメージが湧いていない

データを活用するイメージが想像できず、ビッグデータ分析基盤をアクセス解析ツールの上位版と認識しているケースです。アクセス解析ツールの上位版としては相応のコストが掛かる上に、それに見合う効果が得られるのか疑問視しがちです。この場合、ビッグデータ分析基盤はもっとさまざまな用途に利用可能であることに気付いてないことが原因です。

データを取り扱う人材を確保できていない

「ビッグデータ分析基盤を導入したいが、対応できる人材がいない」ことを理由にするケースです。

詳細に状況を確認すると、「何をすればよいのか分からない」ことから始まり、最終的には「どんな役割の人材が必要なのかが分からない」ことが原因です。ビッグデータ分析基盤の運用に必要な人材を説明すると、ようやく理解して納得してもらうことが多々あります。

昨今の状況の通り、ビッグデータが流行のキーワードとなっていると、組織内にもビッグデータに携わりたいエンジニアが必ずいるはずです。「担当できる人材がいない」のではなく、「どんな役割の人材が必要」か分からないため、「人材を確保できない」との認識に繋がっているのでしょう。

どんなデータを収集すればよいか分からない

従来はGoogle Analyticsなどアクセス解析ツールに対してログを送信しているが、いざビッグデータ分析基盤を利用して、自らデータを収集する段階になると、どのようなログをどんな形式で出力、送信すればいいのか分からないケースです。

既存のアクセス解析ツールを新たなビッグデータ分析基盤に置き換えていく、進化させていくことを念頭に置くと、どうしてもWebサイトにおけるユーザーの行動ログなど大規模なデータと考えがちです。しかし、小規模で始めるには、「2-2-2 業務データ」(P.17参照)で説明した通り、サービスが保有する業務データの分析からスタートしても構いません。残念ながら、業務データからでもデータ分析が可能であることに気付いていないケースが多々あります。

有効にデータを活用できていない

データを有効に活用できていないケースのほとんどが、「データを活用するイメージが湧いていない」、「データを取り扱う人材を確保できていない」、「どんなデータを収集すればよいか分からない」ことに起因しています。技術的な側面に着目すると、さらに下記の2つに分類できます。

- 分析のプロセスや手法が分からない
- 習得すべき技術が分からない

分析のプロセスや手法が分からない状態では、収集すべきデータが後から判明して、手戻りが数多く発生しがちです。また、目的とする分析を実現するためのスキルがなければ、収集データは無駄になります。技術的な側面が理解できないがゆえ、その他の構想部分でも障壁となっているケースもあります。

まとめ

執筆陣に寄せられる悩みは、いずれか1つでも理解できていないと、全体的な構想を描くことが困難になります。ビッグデータチームやデータ分析、データ活用の全体的なイメージが掴めないがゆえに、最初の一歩を踏み出せないチームや組織が多々あります。

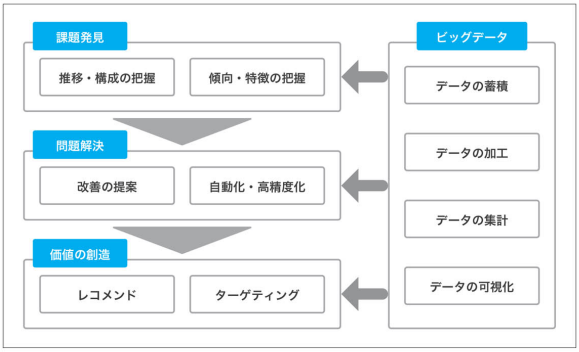
本章では、本書で紹介しているレポートやSQLなどが単なる知識に留まらず、データをより活用するにはどうすべきなのか、データ分析の現場での経験を元に説明しましょう。

9-1-1 データの活用方法を考える

ビッグデータ分析基盤を導入しても、「効果があるのか分からない」ことは、言い換えると「データを活用して何が可能になるのか分からない」状態でしょう。

ビッグデータ分析基盤が導入されているにも関わらず、まだ十分に有効活用できていないケースでは、現在取り組んでいる内容と、ビッグデータ活用の各ステップを照らし合わせて、どの段階にいるかを確認するのが第一です。段階を追ってチャレンジし、データ活用の範囲を広げましょう。

図 9.1.1.1：ビッグデータ活用のステップ



ビッグデータ活用の段階

ビッグデータの活用には段階があります。「課題発見」から「問題解決」、そして「価値の創造」の3段階です。段階が進むにつれて、関わる組織や取り扱う技術の幅も拡がっていきます。本項では、具体的に各段階で、どのようなことを実行するのかを紹介した上で、本書で説明している箇所のインデックスを用意します。

【課題発見】

レポートによって、問題箇所の特定、改善余地を発見する

運用するサービスに課せられている目標に対して、どのように改善を重ねていけばよいか、またその目標に対して現状がどれぐらい乖離しているのか、どのぐらい頑張らないといけないのかを可視化して、組織を鼓舞することもレポートの目的の1つです。

従来は分からなかった部分を数値化、さらに視覚化して、問題点を浮き彫りにします。そして、改善の余地がある箇所を発見することも、分析担当者のミッションの1つです。その結果、サービスにおけるミッションである、売上やユーザー数の向上に繋げることが最大の目的です。

関連項目	5-1-5 ペン図でユーザーのアクションを集計する 5-2-3 継続と定着に影響するアクションを集計する 6-2-6 フォールアウトレポートを用いてユーザーの回遊を可視化する 6-2-9 ユーザー行動の全体像を可視化する など

【問題解決】

既存の仕組みと組み合わせ、自動化や高精度化を図る

ユーザーの趣味嗜好はさまざまであるのに対して、サービスの運用は手作業や画一的な状態がゆえに、ユーザーの多様性を吸収できないシーンがあります。

例えば、メールマガジンにオススメ商品を掲載する際に、ユーザー個別に判断して取捨選択することは、実質的に手作業では不可能です。それゆえ、どのユーザーにも支持されるであろう商品を選択したり、季節的に人気の商品を選びがちです。少なくとも男性・女性の2パターンを作成することは可能であっても、さらにユーザー属性や個々のユーザーに細分化することは、手作業では難しくなります。

ビッグデータ分析基盤を用いると、こうした人力による作業の限界を自動化して、さらに高精度化することが可能です。

関連項目	4-2-3 ファンチャートで商品の売れ行きの伸び率を見る 4-2-2 ABC分析で売れ筋を判別する 5-1-3 年齢別区分ごとの特徴を抽出する 8-2-1 アソシエーション分析 など

ユーザーに提供する検索では、誤った語句で検索した際に「もしかして〜〜？」と表示する機能、もしくはカタカナと英語のいずれでもヒットするように商品を登録できる仕組みがあっても、サービスの成長に伴って、手作業で登録することは物量的に困難になっていきます。

例えば、ビッグデータ分析基盤を用いてユーザーの行動ログを元に、登録作業を自動化して既存の仕組みをより効果的に利用することも、ビッグデータの活用事例といえます。

関連項目	8-1-1 NoMatch率とそのワードを集計する 8-1-2 再検索率とそのワードを集計する など

大量のデータの中で特定条件に該当するデータを抽出し、既存システムと連携することで、より効果的に機能させることも可能です。

例えば、メールマガジンを配布する際に、ヘビーユーザーとライトユーザーで文面を切り替えたり、Webサイトに表示するバナーやモジュールを切り替えるなど、段階的に成長させることもビッグデータの活用では可能です。

関連項目	5-1-6 デシル分析でユーザーを10段階のグループに分ける 5-1-7 RFM分析でユーザーを3つの視点でグループに分ける など

【価値の創造】

システムの提供によって、新たな打ち手を提供する

ビッグデータを用いて提供するシステムとしては、個人の趣味嗜好や行動を把握し、成果に繋がるように広告や商品を表示するターゲティング、レコメンドやパーソナライズ化が思い浮かびます。既存の仕組みでは、処理に時間を要して、提供できなかったものが可能になってきています。

ターゲティングやレコメンド、パーソナライズ化の仕組みを社内で構築する組織もあれば、事業として他社に提供する企業もあります。タグを埋めれば最適な広告が表示される仕組み、レコメンド結果をAPIとして返却する仕組み、期待売上の高い順で並べるなど、従来は不可能であった新たな施策が、ビッグデータを活用することで実現可能となっています。

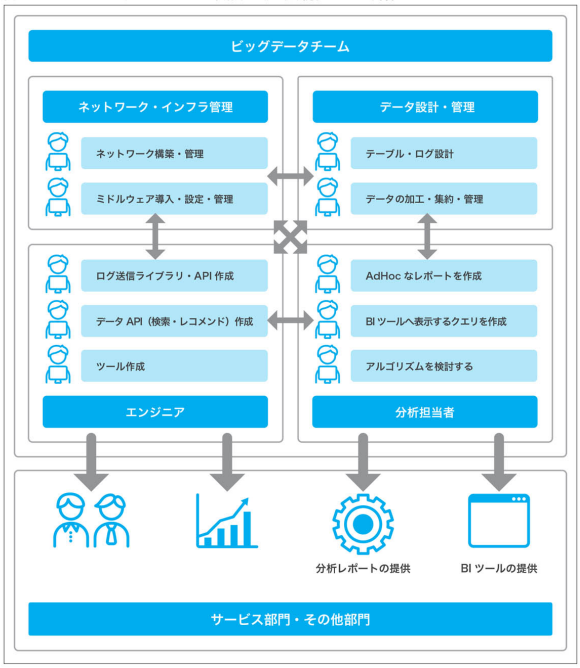
関連項目	8-3-2 このアイテムに興味がある人はこんなアイテムも見ています 8-3-3 あなたにオススメの商品 8-4-5 独自のスコアリング方法を定義してランキングを作成する など

担当サービスや組織の問題点に対してビッグデータで解決可能な余地がないか検討しましょう。また、未着手の領域があれば、関与と影響の輪を広げデータドリブンな組織運営に繋げていきましょう。

9-1-2 データに関わる登場人物を知る

業種や規模によって組織構成はさまざまであるため一般的とはいえませんが、ビッグデータチームにおける役割、登場人物、提供する内容を、下図にまとめます。

図 9.1.2.1：ビッグデータチームにおける役割、登場人物、提供していた内容



図中に登場する人物のアイコンは役割を意味しており、人数ではありません。チームの規模や所属する組織によって、複数の役割を１人が掛け持ちすることもあれば、複数人で１つの役割を担うこともあります。

前記の構成図を読者自身、もしくは所属する組織でのデータに関わる業務と登場人物を、試しに当てはめてみましょう。その上で、組織内でのデータを取り巻く環境や状況が好ましくない場合は、じっくりと問題点を考えてみます。もしかすると、下記にあげる問題点が浮かぶことがあるかもしれません。

- 分析以前にデータ・環境が揃っていない
- 実現したい目的に対して、その役割を担う人物が揃っていない

これはデータ分析に相対した際に一般的に耳にする事象です。それぞれの問題点に対して、執筆陣の考えを下記にまとめます。何らかの解決の糸口となれば幸いです。

分析以前にデータ・環境が揃っていない

「分析以前にデータが揃っていなかったり、そもそもそんな環境がない」ことを課題とするケースがありますが、データや環境が揃っていないことが実際の問題ではありません。サービスを実際に運用しているのであれば、少なくとも業務データは存在するはずです。おそらく「誰に何を働きかければよいのか分からない」ことが根幹にあると考えられます。

もちろん、「ビッグデータを活用して○○を実現したい！」と考えても、現実問題としてあなた１人で実行することは難しいかもしれません。図に示した通り、ビッグデータチームでやるべきことはたくさんあります。しかし、単独で実現できないのであれば、複数のメンバーと協力して進めるべきです。前図を参考にして、担当できるメンバー、担当してくれそうなメンバー、チームを巻き込んで、環境整備を推進しましょう。

やりたいことに対して、その役割を担う人物が揃っていない

経営側の視点を考えると、売上に直接的に関係する営業マンやエンジニアを雇うことには積極的になれるけれども、分析担当者やデータ周りを整備するエンジニアとなると、消極的になってしまうものです。売上規模が大きくないと、データに携わるメンバーを増員することは難しいことです。まして、上図の役割をすべて網羅するとなると、新たに必要なメンバーは…と考えてしまいがちです。

しかし、図の人物のアイコンは役割に過ぎず、必ずしもその人数が必要なわけではありません。例えば、ネットワーク・インフラを管理する役割はクラウドサービス（RedshiftやBigQueryなど）で代替し、メンバーの不足を補えます。BIツールは必ずしも自社で制作する必要はなく、その他のツールで代替しても構いません。また、昨今は分析用途の各種ライブラリも充実しているので、すべてを新たに作成しなくとも、さまざまなアルゴリズムで分析が可能です。

9-1-3 ログフォーマットを考える

サービスごとでデータを保存するテーブル形式を変更すると、集計ロジックも異なり、サービスが増えたり規模が拡大した際に、分析担当者のタスクが増大します。

あらかじめログ形式や格納テーブルを共通にすることで、サービス数の増加や規模拡大時に、分析担当者のタスクが増大することを防げます。

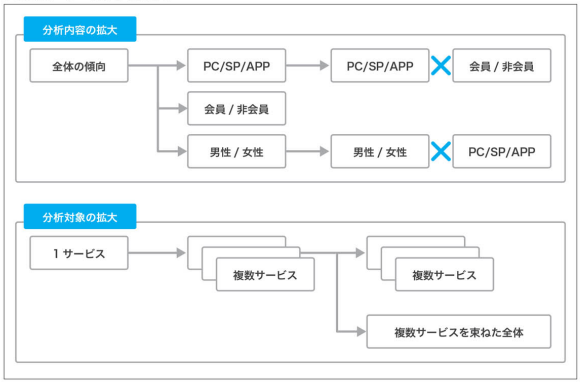
ログフォーマットを作成する際に考慮すること

データ分析における要求は常々拡大する状況です。

- 提供を受ける側が満たされると、新たな次の要望が出てくる
- 成功した事例が出てくると、便乗したくなる
- 勢いのある企業だと、新たなサービスが次々にローンチされる

上記の「拡大」には、「分析内容の拡大」と「分析対象の拡大」の2種類があります。

図 9.1.3.1：データ分析の広がり



レポート業務での技術的な対策を考えると、「分析内容の拡大」に対しては、CUBE関数を用いることで、対象データが増えても、クエリ自体を複数作成する必要はありません。「分析対象の拡大」に対しては、ROLLUP関数を用いることで、個別のサービス単位での集計と同時に、全体の集計が可能です。

つまり、分析の内容や対象が広がっても、あらかじめ将来を見越して集計作業が増大しないように、ログフォーマットや集計を検討しておくことで、分析担当者のタスク肥大化を抑えることができます。

ログフォーマットのサンプル

ECサイトでもコミュニティサイトでも利用できるベースとなる、サンプルフォーマットを下表にまとめます。必要であれば、運用するサイトの特性に合わせて項目を追加することも可能です。

表 9.1.3.1：ログフォーマットのサンプル

物理名	論理名	データ型
dt	日付	文字列
service_code	サービスコード	文字列
action	アクション	文字列
option	オプション	文字列
time	日時	文字列
short_session	訪問回数計測用session	文字列
long_session	訪問者計測用session	文字列
user_id	ユーザーID	文字列
url	URL	文字列
referrer	Referrer	文字列
user_agent	UserAgent	文字列
page_name	ページ名	文字列
view_type	表示タイプ	文字列
via	経由情報	文字列
segment	ABテスト用変数	文字列
detail	詳細情報群	配列 / コレクション / JSONなど
search	検索情報群	配列 / コレクション / JSONなど
show	表示情報群	配列 / コレクション / JSONなど
user	ユーザー情報群	配列 / コレクション / JSONなど
other	その他情報群	配列 / コレクション / JSONなど

上表にまとめた項目で、解説が必要と考えるものをいくつか選び説明します。

・dt

ログには出力時点の時刻まで含めて保存するのが慣例ですが、その時刻とは別に日付を格納します。ビッグデータを扱うミドルウェアには、データを分散させるためのキーを指定したり、パーティションを設定することで高速に動作するものがあります。

日付ごとに集計するシーンが多い場合はデータの分散に使用するため、時刻を除いた日付で保存し、

分散させるためのキーやパーティションに指定します。副産物として、日付＋時刻を毎回加工して、日付にする作業が必要なくなります。

・service_code

service_codeにはサービスを一意に識別する任意の文字列を設定します。現在は1サービスしか運営していなくとも、のちのち複数サービスを運営する際も、同様のログフォーマットで同じテーブルにデータを格納するために、将来を見越して設定します。

同一テーブルに複数サービスのデータが格納されるため、サービス単位の売上やユニークユーザー数、ページビューなどが、GROUP BYでservice_codeを指定するだけで集計できます。

・actionとoption

actionにはユーザー行動やシステム制御を示す文字列を設定します。

例：view / search / add_cart / clip / purchase / like / comment / recommend

optionにはactionを補足する文字列を設定します。

例：「いいね」アクションの対象をoptionに設定する action = like option=post（投稿） / comment（コメント） / photo（写真） /movie（動画）

このactionとoptionを集計することで、ユーザーの行動を識別・把握することが可能になります。

・short_session / long_session / user_id

「6-1-1 日次の訪問者数・訪問回数・ページビューを集計する」（P.245）で紹介した通り、非会員の行動を含む訪問者数・訪問回数を集計するためには、short_sessionとlong_sessionが必要になります。また、会員と非会員の行動には異なる特徴を確認できることがあります。そこで、会員と非会員が区別するためにuser_idを格納します。

・page_name

page_nameにはURLとは別のページを識別する文字列（ページ名）を格納します。URLには「6-1-3 流入元別に訪問回数やCVRを集計する」（P.251）で紹介した広告計測用のパラメーターや画面を制御するパラメーターが付与されます。そのためページに対するログの集計ではURLの加工が必要です。その加工を行わなくてもページ単位での集計を可能にするためページ名を付与します。

また、「6-3-2 入力〜確認〜完了までの遷移率を集計する」（P.307）で紹介した通り、同一URLでも表示する内容が異なる場合、URLだけではどの画面が表示されたのか判別できません。そのため、ページ名を定義してログに出力することで、その判別を可能にします。

・view_type

PCサイトの表示なのかSPサイトの表示なのかを示す文字列を格納します。

スマートフォンではSPサイトを初期表示にして、表示方法を切り替えることでPCサイトを閲覧できるサービスがあります。ユーザーエージェントがスマートフォンであっても、必ずしもSPサイトの閲覧とは限りません。また、URLがPCサイトとSPサイトで共通の場合は、いずれの閲覧か判別できません。閲覧先を示す文字列として表示タイプをログに出力することで判別が可能です。

・via

該当ページに遷移するきっかけとなった経由情報を格納します。例えば、商品詳細画面への遷移時、検索結果からの遷移なのか、レコメンドモジュールからの遷移なのかを、URLパラメーターに付与して、遷移先の詳細画面の表示ログに出力します。このviaの値を集計することで、機能の効果を可視化することが可能です。

・segment

segmentには、ABテスト時のパターンを判別する文字列を格納します。segmentの値別にユーザー行動を集計し、有意な差が認められるかを確認するために使用します。

・detail / search / show / user /other

前述のactionとoptionに対して情報を補足します。ログフォーマットの変更は容易ではないため、あらかじめ集計対象となりそうなもので付加情報が多いもの、配列型やコレクション型、JSON型など、フォーマットは変えずに吸収できる型を採用します。ミドルウェアによって使用できるものが異なるため、詳細は各ミドルウェアのマニュアルを確認してください。

上述のログフォーマットでECサイトでのカート追加時におけるスキーマ例を下表に示します。これを元に出力されるログサンプルをデータ例に示します。ミドルウェアによって、扱える形式は異なりますが、今回はカラムの区切り文字として「|」を、detailカラムにはJSON型を使用します。

表 9.1.3.2：カート追加時に商品の詳細情報をログに出力するスキーマ例

物理名	キー物理名	キー論理名	設定値
dt			2016-11-12
service_code			mynavi_book
action			add_cart
option			(空文字)
(途中省略)			
detail	product_id	商品ID	item013011
	category	商品カテゴリ	database
	price	価格	2980
	count	数量	1

データ 9.1.3.1：ログデータサンプル（カラム区切りとして「|」を使用）

```
2016-11-12|navi_book|add_cart||(途中省略)|{"product_id":"013011",
"category":"database","price":2980,"count":1
```

まとめ

事業規模の拡大に応じて分析担当者の増員が必要になる状態は、問題があるともいえます。先行しているサービスと同様のレポートや作業を求められた際に、即座に対応可能な状態を目指して、ログフォーマットを検討しましょう。

9-1-4 データを活用しやすい状態に整える

ログデータや業務データがビッグデータ分析基盤に保存されたところで、分析レポートの作成やレコメンドAPIの開発に着手する前に、必ずデータをチェックして、不要なデータや異常値を取り除いて、新たなテーブルとして保存することをオススメします。

高頻度で集計するデータは必ず取り扱いが容易な状態に加工して、新たなテーブルとして保存しましょう。この作業を怠ると、下記に挙げるトラブルが発生します。

- 集計クエリが膨大な行数となり、記述が大変になる
- 集計クエリが複雑なため、応答速度が遅くなる
- 不要なデータが含まれているため、分析の精度や信頼性が低くなる
- データ加工の手順が担当によって異なり、結果も異なる

以降、データを扱いやすい状態に整える手順を説明します。

データを3つの階層に分けて扱う

データの種類を3層に区切り、ログ層、集約層、集計層と定義します。あくまでもデータ概念を示す層であり、ビッグデータを扱うミドルウェアにある概念ではありません。

まずは、各階層に格納されているデータとその用途を説明します。

ログ層

ログ層は、サービスにおける行動ログやサービスを構築・運用している業務データそのものを転送して保存する階層です。この階層のデータを用いて分析すると、精度が高くなく、複雑なクエリをその都度作成する羽目になります。集計層へのデータ保存が失敗した時や間違っていた時に、やり直すためのデータをログ層に貯蔵します。ログ層に該当するデータに対して集計は行いません。

集約層

集約層にはログデータや業務データを加工したデータが保存されます。分析担当者やビッグデータ分析基盤を利用して構築するサービスは、集約層のデータにアクセスして集計します。BIツールを使用する際も、この集約層にあるデータを操作します。

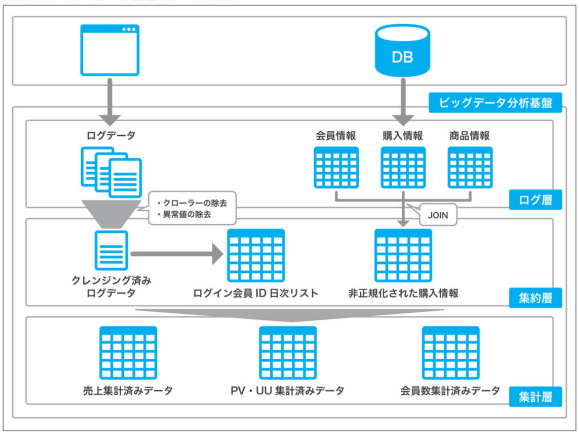
集計層

日々チェックする指標など高頻度でチェックするものは、あらかじめ集計した結果をデータとして保存することで、複数の分析担当者が確認のたびに集計クエリを実行する必要がなくなり、作業時間の短縮やシステムに対する負荷軽減にも繋がります。

各階層に対する操作

ログ層、集約層、集計層の各階層に対して、実施する操作とデータの流れを下図に示します。

図 9.1.4.1：3つのデータ階層とデータの流れ



ログ層から集約層へ

ログデータ

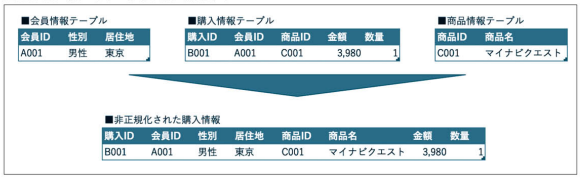
Webサイトにおける行動ログには、クローラーのログが混入するケースがあります。ユーザーのアクセス数と同等もしくは超える件数が混入するWebサイトもあります。機械的にアクセスするクローラーのログがユーザー行動を捉える妨げとなります。本書で紹介した、データ検証のクエリや、不要なデータや異常値を除外するクエリを用いて、分析対象データを限定し集約層へデータを保存します。

関連項目	7.2.2 クローラーを除外する 7.2.3 データの妥当性を確認する 7.2.4 特定のIPからのデータを除外する 7.3.1 マスタデータの重複を検出する 7.3.2 ログの重複を検出する

業務データ

一般的に業務データはリレーショナル構造であるため、1つのテーブルだけでは十分な情報を取得できません。例えば、購入情報の把握には、購入情報の他に会員情報と商品情報を結合しないと、データの全容が分かりません。購入情報など高頻度で集計するテーブルは、はじめから結合した状態で保存して集約層に保存します。複雑なクエリを作成する手間も省け、データの理解や経験が浅い担当者でも取り扱えます。

図 9.1.4.2 : 扱いやすいように非正規化する



関連項目	3.4.1 複数のテーブルを縦に並べる 3.4.2 複数のテーブルを横に並べる

集約層から集約層へ

集約層に保存されているデータは不要なデータが除去されているため、ログ層に保存したデータの集計結果よりも、精度・信頼性が高くなります。

高頻度での使用が想定されるデータを抜き出して、別のテーブルとして集約層に保存します。例えば、当日にログインしたユーザーIDのみを重複なしの状態で格納するテーブルなどです。以降の会員に対する分析は、このテーブルと他のテーブルを紐付けるだけで済み、処理も高速になります。

また、集計に時間が掛かったり、SQLの経験が浅い人が取り扱うには難しいデータは、中間データを保存することで、作業時間の短縮や負荷軽減に繋がります。SQLのカウントや合計など集計結果ではなく、「8-1-5 検索ワードに関する指標の集計を効率化する」で紹介したコード8.1.5.1（P.383）など、カウント・集計する前のデータを保存するイメージです。

関連項目	5-2-7 訪問頻度からユーザーの属性を定義し集計する 5-2-8 訪問種別を定義して成長指数を集計する 8-1-5 検索ワードに関する指標の集計を効率化する

集約層から集計層へ

集約層に保存されたデータで高頻度にチェックする指標をあらかじめ集計して保存します。

例えば、売上や会員登録数、訪問者数、訪問回数、ページビューなどをあらかじめ集計して保存しておきましょう。ただし、集計済みのデータであるため、ドリルダウンが必要な場合は集約層に対して集計を行なうか、集計層にドリルダウンが可能な状態で保存し直すことが必要になります。

関連項目	4-1-1 日別の売上を集計する 4-1-3 当月売上の累計を求める 5-2-1 登録数の推移と傾向を見る 5-2-2 継続率と定着率を算出する 5-2-8 訪問種別を定義して成長指数を集計する 6-1-1 日次の訪問者数・訪問回数・ページビューを集計する

本項で説明した通り、3つの階層を意識して、加工・集積して分析することで、作業効率が格段に上昇します。

9-1-5 データ分析のプロセスを習得する

データ分析の重要性を理解しつつもデータ分析の手法が分からず、うまくデータを活用できないと相談されることがあります。データがうまく活用されず、サービスでの施策は経営者やプロデューサーによる「勘・経験・度胸」に頼っている状況とのことです。

「勘・経験・度胸」でも、担当者が生まれつきの感性を持ち、実施する施策はすべて効果的で、売上も順調であれば無理に分析する必要はありません。しかしながら、そのような状態では担当者が離職す

ると途端にサービスが伸び悩んだり、縮小してしまう恐れがあります。

このような事態に陥らないために、データを活用し根拠を持ってサービスの改善を進めましょう。データ分析のやり方がわからない方でも、データを活用できる人材、組織になるために、データ分析における4つのプロセスを紹介します。

データ分析の4つのプロセス

データ分析で大事にしているプロセスがあります。レポート作成の仕事を分解すると、ほとんどが下記のプロセスのいずれかに該当するはずです。

1. まず大雑把に把握すること
2. グラフ化すること
3. 複数の数値で比較すること
4. 仮説を立てて検証すること

逆説的ではありますが、上記のプロセスさえ押さえていれば、大半のレポート作成業務が遂行できるといえます。各項目に関して、どのような理由で挙げているかを紐解きます。

まず大雑把に把握すること

データ分析を任された担当者は、「より効果的な施策を提案したい」「この記事のような高度な分析をして、何らかの糸口を探ってみよう」などと考えるかもしれません。しかし、最初から詳細なもの、高度なレポートの作成を試みると必ず失敗します。

数値を算出したものの、サービス全体で高いのか低いのか把握できなかったり、そもそもの注力箇所を見誤る恐れもあります。後から誤りに気付き、全体の傾向を再度分析する手戻りが発生します。

大雑把に全体像を把握して、気になる箇所を詳細に掘り下げると手戻りも発生しません。レポートを受け取る側も順を追って消化できるので、まずは大雑把に把握することを心掛けましょう。

グラフ化すること

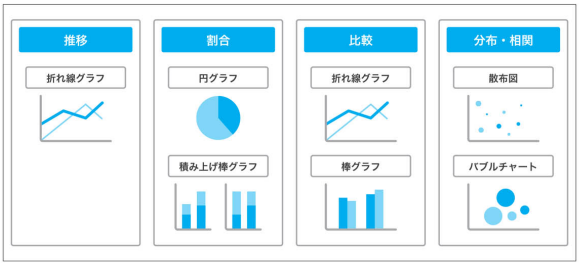
数値を羅列しても、どこが上昇してどこで下降しているのか、どのような特性があるのかなど、その特徴は捉えにくいものです。グラフ化することで、推移や割合、比較、分布、相関などの視認性が飛躍的に向上します。

例えば、「4-1-5 Zチャートで業績の推移を見る」(P.104) で紹介したZチャートには、グラフの傾き、膨らみに意味があります。また、「4-2-4 ヒストグラムで購入価格帯を集計する」(P.122) ではヒストグラムでデータ分布を表現する方法を紹介しています。同じ平均値でもデータの分布が異なるパターンがあります。

「5-2-5 利用日数に応じた定着率を集計する」(P.199) や「5-3-1 ユーザーのアクション間隔を集計する」(P.228) で紹介したグラフは、数値の羅列では捉えにくい、変化するタイミングや鈍化するタイミング、縮小傾向になるタイミングなどを明確に伝えてくれます。数値の羅列から脳内でグラフを描くことは困難です。グラフによる可視化で新たな発見もあるので、データ抽出とグラフによる可視化はセットで実行することをおすすめします。

また、データから取得したい情報で使用するグラフの種類が異なります。どのようなグラフが求める情報に最適かを判断して、効果的に活用しましょう。

図 9.1.5.1：利用目的と使用するグラフ例



複数の数値で比較すること

「4-1-6 売上を把握するための大事なポイント」(P.109) で紹介した通り、売上だけを集計していると、売上の上昇・下降しか分からず、その理由までは分かりません。

サービスの結果である売上に対して、その元となる購入回数や購入単価などの周辺データと見比べることで、売上の「なぜ」が分かります。売上のレポートを求められたら、周辺データも含めてレポートすることが望ましいでしょう。

また、「5-1-3 年齢別区分ごとの特徴を抽出する」(P.143) で紹介した通り、データのある側面だけに注目すると誤って判断することもあります。必ずさまざまなデータで比較することが大切です。

複数の数値で比較する際には、下記の5つの手法で行います。

- 関連する指標で割って比較する
- 関連する複数の指標を並べて比較する
- データを細分化して確認する

- 先月、前年のデータと比較する
- 同様の指標を他のサービスと比較する

関連する指標で割って比較する

「4-1-1 日別の売上を集計する」(P.95) で紹介した通り、売上を購入ユーザー数で割れば、1人あたりの平均購入額が分かります。より売上に寄与するページはどこかを探るために、「6-2-3 成果に結びつくページを把握する」(P.269) や「6-2-4 ページの価値を算出する」(P.273) では、訪問回数を成果数や売上で除算して、新しい指標を提案しています。

このように、データをカウントもしくは合計して、その総数を他の値で割ることで新しい指標を定義でき、新たな新たな発見も得られます。

関連する複数の指標を並べて比較する

「4-1-6 売上を把握するための大事なポイント」(P.109) で紹介した通り、売上が下降傾向にあれば、関連するデータ分析で原因の特定や改善案を検討します。関連する指標として、訪問回数やCVR、1人あたりの平均購入額などを確認することで、原因を探ることができます。

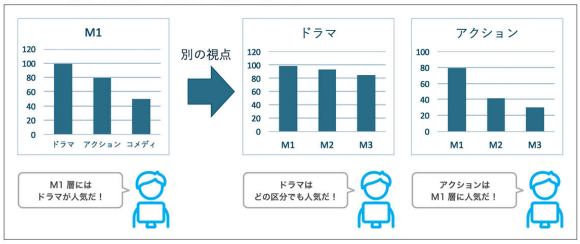
データを細分化して確認する

全体像を大雑把に把握してグラフ化したら、データの構成要素に落とし込み、どのように構成されているのか、どの部分が変化しているのかを把握します。

例えば、「5-2-1 登録数の推移と傾向を見る」(P.170) では、登録数をデバイス別に分解して集計し、「4-2-4 ヒストグラムで購入価格帯を集計する」(P.122) ではヒストグラムで購入されている商品の価格帯別に集計する方法を紹介しています。

データを細分化していくことで、状況の変化における要因を絞り込んでいくことが可能です。

図 9.1.5.2：複数の図を見ることで異なる判断が可能になる



先月、昨年のデータと比較する

長く運用しているサービスであれば、先月のデータはもちろん、前年データとの比較も可能です。数値の変動が季節トレンドによるものであれば、売上や成果数が上がっても開発・運用しているチームの成果ではありません。逆に売上が下がっても、季節トレンドで許容される下落幅であれば、慌てる必要はありません。

「4-1-4 月別売上の昨対比を求める」(P.102) で昨年と比較する方法を、「4-2-3 ファンチャートで商品の売れ行きの伸び率を見る」(P.119) で先月と比較する方法を紹介しています。先月や昨年のデータと比較することで、新たな発見や考察が得られます。

同様の指標を他のサービスと比較する

担当するサービスの指標を見ても、悪い状態なのか良い状態なのか判断しかねる場面があります。そのような時は、所属する組織が他のサービスを展開しているのであれば、そのサービスと比較して良い状態か悪い状態か判断することができます。

仮説を立てて検証すること

大雑把にデータを把握して、複数の数値と比較している際に、さまざまな疑問や仮説が浮かび上がります。仮説を立てて検証することで、「こんな傾向があるから、こういうことをやってみよう」などの施策に結び付きます。データ分析のみで終わらせずに、サービスに対して施策を実施して、効果の有無をしっかり検証しましょう。

「5-2-9 指標を改善する手順を身に付ける」(P.225) で紹介した通り、「達成していない人を達成している人に寄せる」ことを意識することで、現状を伝えるデータ分析から、現状を変えるためのデータ分析ができるようになります。

9-1-6 分析のはじめの一歩を踏み出す

前節「9-1-5 データ分析のプロセスを習得する」でプロセスは理解したものの、どこから手を付けてよいのか分からないときもあるでしょう。本項では2つのやり方を紹介します。

KGI・KPIを設定して、関連する箇所を分析する

まず、KGIとKPIを説明しましょう。

> **KGI (Key Goal Indicator)**：重要目標達成指標
> 組織やサービスにおける最終的なゴールとなる目標を定量化した指標

KPI（Key Performance Indicator）：重要業績評価指標
KGIを達成するための中間指標

KGIは、組織やサービスで「月間売上○○万円」「ユーザー数○○万人」など、具体的な数値を設定します。「顧客満足度を向上する」「日本一の○○になる」といった曖昧なものではなく、計測可能な指標を設定します。

KPIはKGIを達成するために必要不可欠な過程を数値化したものです。KGIを因数分解したものがKPIといえば分かりやすいでしょうか。KPIもKGIと同様に計測可能な指標を設定します。売上をKGIとして設定した場合は、KPIは下記の式で表現できます。

図 9.1.6.1：KGIを因数分解してKPIを設定する

仮にKGIを「売上2倍の○○万円」と設定した場合、KPIである訪問者数とCVR、客単価のどれか1つを2倍にすればKGIを達成できます。または、訪問者数とCVR、客単価をそれぞれ1.3倍ずつ増やせば、売上が2倍になると考えることもできます。KPIの増加がKGIに影響する式であるべきです。式を作成したら、KPIに注目して下記のことを実施していきます。

- KPIがどのような推移・構成になっているのか確認する
- KPIを達成している人と達成していない人で比較する
- KPIをさらに因数分解し、推移・構成を探る

そのための分析手法を本書から選択して進めていくのはどうでしょうか。

表 9.1.6.1：KPIの分析における分析の一部を本書から抜粋

KPI	関連項目
訪問者数	6-1-2 ページ毎の訪問者数・訪問回数・ページビューを集計する 6-1-3 流入元別に訪問回数やCVRを集計する 6-2-1 入口ページと出口ページを把握する
CVR	6-2-3 成果に結び付くページを把握する 6-2-6 フォールアウトレポートを用いてユーザーの回遊を可視化する 6-3-2 入力～確認～完了までの遷移率を集計する
購入単価	4-2-1 カテゴリ別の売上と小計を計算する 4-2-4 ヒストグラムで購入価格帯を集計する 5-1-7 RFM分析でユーザーを3つの視点でグループに分ける

全体像を把握して、改善箇所を発見し深掘りする

「6-2-9 ユーザー行動の全体像を可視化する」(P.301) で紹介したサービスの俯瞰図を作成しましょう。アクセス解析ツールが提供するレポートや分析チームの作成レポートは断片的なもので、全体が捉えられないことはないでしょうか。ことわざに「木を見て森を見ず」とありますが、「木」ばかりを見ていて、「森」が見えていない状態では、本当に注力すべき箇所がどこか分かりません。

さまざまなレポートを日々確認して分かっている状態でも、チームメンバー全員が同じ認識を持っているとは限りません。一目でサービスの状況が分かる俯瞰図を作成することで、チーム全員で問題点を共有でき、議論がぶれることなく、サービス改善の意見交換が可能になります。チームのメンバーからさまざまな疑問や改善案が上がるようになれば、チームはもちろんサービスの活気に繋がります。また、俯瞰図を作成していく段階で、問題点が浮き彫りになったり、疑問や仮説も出てくることでしょう。原因究明や施策立案まであと一歩です。

俯瞰図すらどこから作成すべきか分からない場合は、本項のKGIとKPIに関連する箇所から開始しましょう。まずは全体の売上額やユーザー数、ページビューなどでも構いません。「9-1-5 データ分析のプロセスを習得する」(P.466) で紹介した通り、全体を飛び越えて詳細なものを作ろうとしてもうまくいかないものです。焦らずに1つずつ紐解いていきましょう。

9-1-7 相手の職種・役職に応じたレポートを作成する

提出したレポートが活用されていないと感じたり、レポートを読み上げても相手に伝わっていないと感じることがあるはずです。これは「作成したレポートに提出先との意識に乖離がある」ことが原因と考えられます。本項では、レポート提出先との乖離をなくすために心掛けるべきことを紹介します。

レポートを誰に提出するのか

レポートの提出先が誰であるかが重要です。分析担当者はさまざまな職種や役職とやり取りする必要があります。当然のことですが、やり取りする相手の職種や役職によって、相手が期待するものは大きく異なります。やり取りする職種・役職別に例を挙げます。

- 役員・経営層
- 事業部長、プロデューサー
- サービス企画・開発担当者
- マーケティング部門

役員・経営層の場合

複数サービスを運営している役員や経営層であれば、どこに注力するべきなのか、事業として存続させるべきなのか常日頃から大局的に考えています。したがって、「このページの離脱率～」や「エントリーフォームの入力～確認～完了までの遷移率は～」などのレポートは粒度が細かすぎます。

規模感の把握、他社との比較、売上の推移、売上の構成、ユーザー数などを期待しているはずです。本書に記載した内容では、下記のレポートをサービスごとに集計し、比較可能なレポートを提出すると、今までとは違う反応があるかもしれません。

関連項目	4-1-5 Zチャートで業績の推移を見る 5-2-7 訪問頻度からユーザーの属性を定義し集計する 5-2-8 訪問種別を定義して成長指数を集計する など

事業部長・プロデューサーの場合

役員・経営層とは異なり、1つもしくは複数のサービスを管轄している、重要なポジションにいる役職では、もう少し粒度の細かいものが好まれるかもしれません。このポジションは、担当するサービスの売上やユーザー数を拡大することを重要なミッションとして課せられています。つまり、KGIを強く意識するポジションと考えられます。

そこで、KGIの達成に注目すべき箇所を特定したり、全体の問題点を探るレポートを提出する、さらには、売上を上げる具体的な施策をはじめ、データを活用した新たな戦略を提案するのが好ましいと考えられます。

本書記載の内容では、役員・経営層に提出するレポートの他に、下記のレポートや分析結果を提出すると、話に耳を傾けてくれるかもしれません。

関連項目	4-2-2 ABC分析で売れ筋を判別する 4-2-3 ファンチャートで商品の売れ行きの伸び率を見る 5-1-7 RFM分析でユーザーを3つの視点でグループに分ける 6-2-9 ユーザー行動の全体像を可視化する 8-3-2 このアイテムに興味がある人はこんなアイテムも見ています など

サービス企画・開発担当者の場合

前述の事業部長・プロデューサーと比較すると、このポジションはKPIを強く意識することとなります。事業部長・プロデューサーがKGIを達成するために、所属チームのメンバーに「○○数、○○率を何%改善するように！」と指示し、その責務を果たすポジションです。

したがって、全体の推移や構成を示すレポートではなく、より具体的に「何をすればよいか」を示すレポートや、細かいユーザー行動やページ単位での離脱率、直帰率、遷移率などのレポートを提出すると、具体的なアクションに結び付くため、重宝されるはずです。

本書に記載した内容で示すと、下記に挙げる施策の立案に直接結び付くレポートが最適です。

関連項目	5-2-3 継続と定着に影響するアクションを集計する 5-2-4 アクション回数に応じた定着率を集計する 6-2-2 離脱率と直帰率を計算する 6-2-5 検索条件毎のユーザー行動を可視化する 6-2-6 フォールアウトレポートを用いてユーザーの回遊を可視化する など

マーケティング部門の場合

広告の出稿や外部とのアライアンスを推進するマーケティング部門では、投下した予算に対して、どれだけ売上に結び付いたのかが重要です。効果のない広告やアライアンスに予算を投じるよりも、効果の高い広告に予算を投下して売上の拡大を図り、さらに大きな予算を確保して、より大規模に広告を展開しようと日々拡大を目指しています。

マーケティング部門に対しては、より効果を得るための材料を提示したり、出稿している広告ごとの成果・獲得効率の可視化、予算確保のための材料を示すと効果的です。本書記載の内容では下記のレポートが該当します。

関連項目	5-3-3 登録からの売上を経過日数別に集計する 6-2-3 成果に結び付くページを把握する 6-2-4 ページの価値を算出する など

レポートを受け取る人は、何を知りたいのか

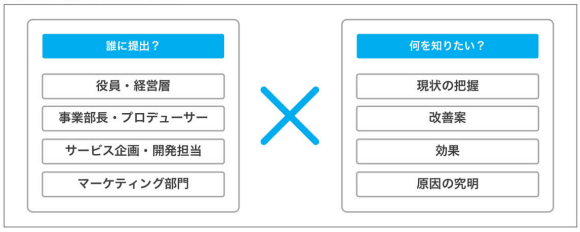
職種や役職によって知りたい指標と粒度が異なります。さらには、その知りたい指標と粒度に対して、何を求めているかで提出すべき内容が異なります。

- 指標の推移・バランスを知りたい（現状の把握）
- 問題点、改善すべきポイントを知りたい（改善案）
- 投下したコスト、施策に対しての効果を知りたい（効果）
- なぜ指標が増加、減少したのか原因を知りたい（原因の究明）

端的には、PDCAサイクル（Plan、Do、Check、Act）を進めていく段階で、どのフェーズにいるかで求められるレポートが異なります。現状の把握ばかりでは、当初は話を聞いてくれるかもしれませんが、「それで？どうしたら良いの？」と返されるだけになります。

例えば、施策の実施直後は当然、全体像の把握や問題点・改善ポイントを示すレポートではなく、施策を打った箇所や影響する指標の変化を可視化することが求められるレポートです。

図 9.1.7.1：誰に何を提出するのか



9-1-8 さらなるデータ活用のスキルを磨く

本書では、ビッグデータ分析・活用術のアウトプットとして、さまざまなレポーティング手法や分析指標を紹介し、それらを実現するためのSQLの紹介に特化してきました。そのため、基本的なSQLの構文説明やSQLプログラミングの背景知識、各ミドルウェアの特性に応じたパフォーマンス・チューニングなどは割愛しています。しかし、実際にビッグデータ分析の現場で活躍するためには、本書の扱っている範囲を越えて、自力で課題を解決していく必要に迫られることでしょう。

本項では、本書を執筆する際に参考とした書籍を紹介し、本書では扱いきれなかった分野のスキルを磨いていくための指針を示します。SQLプログラミングを学ぶための関連書籍、各ミドルウェアの知識を身に付けるための関連書籍、検索・レコメンドなど応用的なデータ活用の知識を学ぶための関連書籍を紹介します。

SQLプログラミング関連書籍

SQLは1980年代に登場した歴史の長い言語であり、関連書籍も数多く存在します。しかし、SQLの基本的な構文はその長い歴史に反して普遍的であり、SQLの本質的な考え方を習得してしまえば、覚えることはそう多くはありません。ここではSQLを学ぶ上で執筆陣が座右の書としている書籍を厳選し、次の3冊を紹介します。

『プログラマのためのSQL 第4版』

Joe Celko（著）、ミック（訳）
（翔泳社）

SQLに関する書籍を1冊だけ選ぶのであれば、まず『プログラマのためのSQL』が候補に挙がるでしょう。SQLの基本的な概念や理論的枠組みから、SQLの多彩な機能までを網羅した書籍で、ウィンドウ関数など、新しい構文の解説も詳しく記載されています。合計800ページ弱のボリュームは手に取ることを躊躇させてしまうかもしれませんが、SQLを学ぶ上で必要なことはすべてこの書籍に書かれているといっても過言ではありません。注意点としては、書籍で紹介されているSQLのコードは基本的に標準SQLの構文で記述されているため、必ずしも各ベンダーの製品で動作するとは限らない点です。

『SQLパズル 第2版 プログラミングが変わる書き方/考え方』

Joe Celko（著）、ミック（訳）
（翔泳社）

『SQLパズル』は、前述の『プログラマのためのSQL』と対になる「問題集」的な立ち位置となる書籍です。題名の「パズル」からある種の趣味的な意味合いを感じるかもしれませんが、この書籍で扱われている問題は、実際にWebのフォーラムで投稿された業務上の課題や、筆者がエンジニアたちに相談された課題を元に作成されており、極めて実践的な課題ばかりです。ただし、出版時期が比較的古いため、今日では当たり前となっている共通表式（WITH句）やウィンドウ関数は、一部のクエリで追記されているに留まります。

『SQL Hacks ―データベースを自由自在に操るテクニック』

Andrew Cumming（著）、Gordon Russell（著）、西沢 直木（訳）
（オライリー・ジャパン）

上記のジョー・セルコ本が標準SQLに準拠したSQL本だとすれば、『SQL Hacks』は各ベンダーの実装に即したSQLテクニック集といえます。対応するデータベースシステムはMicrosoft SQL Server、Oracle、MySQL、Access、PostgreSQL、DB2です。「SQLでここまでできるのか！？」という驚きから、「SQLでここまでやるのか…」というマニアックなテクニックまでを提供してくれる良書です。

なお、著者の運営している「sqlzoo」（http://sqlzoo.net/）は、オンラインでSQLを実践しながら学べるチュートリアルを提供しており、SQL初学者にはおすすめのサイトです。

ミドルウェア関連書籍

本書では、各ミドルウェアの特性や分散処理の原理、環境構築、パフォーマンス・チューニングなどの項目には触れていません。しかし、ビッグデータを扱う上で、パフォーマンス・チューニングやその基礎となる分散処理に関する知識は不可欠です。ここでは、本書で扱っているミドルウェアの分散処理の原理が学べる書籍を紹介します。

『Hadoop 第3版』

Tom White（著）、玉川 竜司（訳）、兼田 聖士（訳）
（オライリー・ジャパン）

Hiveの分散処理の原理であるMapReduceを学ぶには、Hadoopを理解することが近道です。『Hadoop』は、Hadoopと呼ばれる分散処理フレームワークが必要となった時代的背景から、HadoopやMapReduceの仕組み、Hiveの実行、Sqoopを用いたデータ転送などを体系的に学ぶことができます。Hadoop運用者でもなければすべての内容を理解する必要はないかもしれませんが、代表的な分散処理システムの例として、Hadoopの仕組みを理解しておいて損ではないでしょう。

『データベース徹底攻略 (WEB+DB PRESS plus)』

松信 嘉範（著）、羽生 章洋（著）、ミック（著）、奥野 幹也（著）、松下 雅和（著）、桑野 章弘（著）、
青木 峰郎（著）、ひろせ まさあき（著）、小林 篤（著）、島田 康樹（著）、
WEB+DB PRESS編集部（編集）
（技術評論社）

雑誌『WEB+DB PRESS』に掲載されたデータベース関連記事を再編集した特集号です。DB設計やSQLの基本的な考え方に加え、RDB（MySQL）とRedshiftのパフォーマンス・チューニングにも言及されており、幅広いミドルウェアのチューニングを学ぶ入門書として最適です。

『詳解 Apache Spark』

下田 倫大（著）、師岡 一成（著）、今井 雄太（著）、石川 有（著）、
田中 裕一（著）、小宮 篤史（著）、加茂 長門（著）
（技術評論社）

『詳解Apache Spark』は、現時点で刊行されている日本語のSpark関連本では最も新しい書籍です（2017年3月執筆時）。「SparkSQL」と呼ばれるSparkの1つのインターフェースに関してしか紹介していませんが、Sparkエコシステムのその他の機能や、Sparkにおける分散処理の概念を学ぶ入門書としておすすめです。

検索・レコメンド関連書籍

「Chapter 8 データを武器にするための分析術」(P.368) では、検索エンジン改善のための基本的な評価指標や、簡単なレコメンドの実装方法を紹介しました。ここでは、より発展的な評価指標やレコメンドアルゴリズムの実装について学ぶための書籍を紹介します。

『情報推薦システム入門 -理論と実践-』

Dietmar Jannach (著)、Markus Zanker (著)、Alexander Felfernig (著)、Gerhard Friedrich (著)、田中 克己 (訳)、角谷 和俊 (訳)
(共立出版)

『情報推薦システム入門』は、情報推薦 (レコメンド) に関わるさまざまな技術を体系的にまとめた書籍です。代表的なレコメンドアルゴリズムの分類や詳細なアルゴリズムの解説、現実のレコメンド機能を実装する上での課題と一般的な解決策を 1 冊で学ぶことができます。

レコメンドの技術には、機械学習や統計処理、データマイニングなどのさまざまな要素技術が組み合わせられており、その応用分野も幅広いため、体系的な知識を身に付けることはビッグデータ分析・活用でも大きな助けとなります。

Modern Information Retrieval: The Concepts and Technology behind Search (2nd Edition)

Ricardo Baeza-Yates (著)、Berthier Ribeiro-Neto (著)
(Addison-Wesley Professional)

『Modern Information Retrieval』は、情報検索に関連する理論的背景や技術を体系的にまとめた書籍で、検索エンジンの歴史からモデリング、実装、評価、今後の技術的な課題までが詳細に解説されています。

英語の書籍である点と900ページを越えるボリュームには、少し敷居が高く感じるかもしれませんが、一部の章は公式ページ (http://www.mir2ed.org/) からPDFをダウンロードできるので、気になった章から試し読みするのも良いでしょう。特に第 4 章のRetrieval Evaluationの章は、検索エンジンの評価に留まらず、さまざまな応用分野で独自の指標を考え出すためのヒントが得られるはずです。

9-1-9 ビッグデータ時代のデータ分析者に向けて

「1-2-3 職種を越える横断的な分析力を身に付ける」(P.6) では、下記のことを述べました。

> 本書は、分析担当者とエンジニア双方に新しい知識を提供し、相互間の通訳的な役割となり、双方がビッグデータを扱う上で必要な知識を習得し、業務を遂行できるスキルを身に付けることを目的としています。最終的には、分析担当者とエンジニアの枠を越えて、自らデータを取り扱って施策を考案し、事業を良い方向に導く専門的な立場の人材となることを望んでいます。
>
> 過去によく耳にしたWebアナリストでもなく、昨今話題のデータサイエンティストともまた違う、オールマイティかつ必要不可欠な人材として、新たな道を切り開いていく手助けとなることを願います。

上記を実現するために、あなたは「9-1-2 データに関わる登場人物を知る」で紹介した、さまざまな役割、さまざまな技能を持つ人と協力して、データ分析基盤を構築、運用していくことでしょう。所属するチームを超えて、協力を願うこともあるでしょう。

そして、「9-1-7 相手の職種・役職に応じたレポートを作成する」で紹介したさまざまな職種、さまざまな役職とやり取りして、「9-1-1 データの活用方法を考える」で紹介した、データに関するさまざまな提案、施策の立案を行います。データ分析の担当者であるあなたは、領域を超えて各々とやり取りできることが求められます。また、それぞれが何を求めているのかを判断して、最適なソリューションを提案することになります。

また、そのソリューションは単に組織に対するメリットだけでなく、一般ユーザーに対して、適切な商品をオススメしたり、最適な情報を適切なタイミングで届けることで、ユーザーの生活をより豊かにする手助けができるかもしれません。

データ分析者はさまざまな人と関わり、互いに支え合い協力して、サービスや組織の価値を最大化していくことができる、極めて重要かつ刺激的なポジションであると感じます。

関わる登場人物が多いゆえに、大変なこともあるでしょう。しかしながら、自分の領域を狭めず、関心と関与、影響の輪を広げることで、大変ながらも楽しさが生まれてくるはずです。

本書が、皆様の業務で何らかのきっかけ、もしくは手助けになることを祈りつつ、ここに筆を置きます。

加嵜長門・田宮直人

数字

0除算 ····· 43
2ノルム正規化 ····· 410, 417

記号

|| 演算子 ····· 37
<->演算子 ····· 45
<<演算子 ····· 53

A

ABC分析 ····· 115
abs関数 ····· 45
add_months関数 ····· 48
age関数 ····· 50
Amazon Redshift ····· 12
Apache Hive ····· 10
ARPPU ····· 237
ARPU ····· 237
array型 ····· 90
Average Revenue Per Paid User ····· 241
Average Revenue Per User ····· 241
AVG(CASE～)構文 ····· 337
AVG関数 ····· 58

B

Binary Preferece ····· 394
BPREF ····· 394

C

CASE式 ····· 27, 71, 83, 139
CAST関数 ····· 32
Click Through Rate ····· 36
COALESCE関数 ····· 35
collect_list関数 ····· 68
CONCAT関数 ····· 37
Conversion Rate ····· 36
Cost Per Acquisition ····· 242
COUNT(DISTINCT ～)構文 ····· 133, 245
COUNT関数 ····· 58
CPA ····· 242
CROSS JOIN unnest構文 ····· 152
CROSS JOIN ····· 70
CTE ····· 85
CTR ····· 279
CUBE句 ····· 148
CVR ····· 251, 269, 279

D

date_add関数 ····· 48
date_diff関数 ····· 49
date_part関数 ····· 257
date_sub関数 ····· 48
dateadd関数 ····· 48
datediff関数 ····· 49
date型 ····· 31
DCG ····· 394
DENSE_RANK関数 ····· 62
Discounted Cumulated Gain ····· 394
DISTINCT句 ····· 58

E

E-Measure ····· 397
EFO ····· 304
Entry Form Optimization ····· 304
explode関数 ····· 70
EXTRACT関数 ····· 33, 50, 257

F

F-Measure ····· 398
FIRST_VALUE関数 ····· 63, 119, 262, 285
FLOOR関数 ····· 125
from_timestamp関数 ····· 48
FULL OUTER JOIN ····· 358, 384, 390

G

generate_series関数 ····· 88
Google BigQuery ····· 13
greatest関数 ····· 38
GROUP BY句 ····· 58, 95

H

HAVING句 ····· 347
host関数 ····· 29

I
inet型 ····· 53, 316, 339
interval型 ····· 47
IPアドレス ····· 53
IPアドレスの除外 ····· 339
IS DISTINCT FROM演算子 ····· 358
Item to Item ····· 407, 410

K
Key Goal Indicator ····· 470
Key Performance Indicators ····· 470
KGI ····· 470
KPI ····· 470

L
LAG関数 ····· 62, 171, 285, 291
LAST_VALUE関数 ····· 63, 262
LATERAL VIEW ····· 70
LATERAL VIEW explode構文 ····· 152
LEAD関数 ····· 62, 291
least関数 ····· 38
LEFT JOIN ····· 80
Life Time Value ····· 242
LIKE演算子 ····· 333, 377
listagg関数 ····· 68
log関数 ····· 431, 442
lpad関数 ····· 53
LTV ····· 237

M
MAP ····· 393
MapReduce ····· 11
map型 ····· 91
MAU ····· 207
MAX(CASE～)構文 ····· 68
MAX関数 ····· 58
Mean Average Precision ····· 393
Mean Reciprocal Rank ····· 394
Min-Max正規化 ····· 434, 449
MIN関数 ····· 58
Monthly Active Users ····· 207
MRR ····· 394

N
NDCG ····· 394
NoMatch率 ····· 370
NoMatchワード ····· 370
Normalized DCG ····· 394
NTILEウィンドウ関数 ····· 155
NULLIF関数 ····· 42

O
OVER(...ORDER BY～) ····· 61
OVER(...PARTITION BY～) ····· 58
OVER(...ROW～) ····· 61
OVER(ORDER BY～) ····· 96
OVER(PARTITION BY～ORDER BY～) ····· 98

P
P@n ····· 391
parse_url関数 ····· 29
PERCENT_RANK関数 ····· 331
point型 ····· 45
PostgreSQL ····· 10
POWER関数 ····· 45, 430
PV ····· 245

R
RANK関数 ····· 62
regexp_replace関数 ····· 29
regexp_split_to_table関数 ····· 70
RFM分析 ····· 159
RIGHT OUTER JOIN ····· 358
RMS ····· 45
ROLLUP句 ····· 113, 133
ROW_NUMBER関数 ····· 62, 273
ROWS BETWEEN start AND end構文 ····· 64

S
SIGN関数 ····· 38, 83, 148, 269
SparkSQL ····· 13
split_part関数 ····· 30
split関数 ····· 30
sqrt関数 ····· 45
stddev_pop関数 ····· 438

string_agg関数……68
substring関数……28, 34
SUM(CASE～END)構文……102
SUMウィンドウ関数……108, 269
SUM関数……58

T
timestamp_add関数……48
timestamp_sub関数……48
timestamp型……31

U
UNION ALL句……79, 88
unix_timestamp関数……48
unnest関数……70
URL関数……28, 247, 251
URL生成ツール……251
User to Item……407, 417
UU……133

V
VALUE句……88

W
width_bucket関数……122
WITH句……85

Z
Zチャート……104

ア行
アソシエーション分析……400
移動平均……96
入口ページ……262
ウィンドウ関数……58, 61
エラー率……305
エントリーフォーム最適化……304

カ行
カート放棄率……233
下位n%……331
階級……122
確信度……400
確定率……307
カゴ落ち率……233
加重平均……427, 447
カムバック……207
基準値……438
業務データ……15, 17
国判定……316
クローラーの除外……333
継続率……174, 189
合計……95
構成比累計……115
顧客生涯価値……237
コサイン類似度……410, 417

サ行
再検索率……373
再検索ワード……373, 377
再現率……384
最頻値……122
昨対比……102, 170
残存率……203
時間帯……256
シグモイド関数……434
支持度……400
四則演算子……38
上位n%……331
小計……113
スピアマンの順位相関係数……362
正規表現……28, 247, 251
成長指数……218
絶対値……45
相加平均……427
相関サブクエリ……80
相関ルール抽出……400
相乗平均……427

タ行
対数……442
地域判定……316
調和平均……427
直帰率……266

定着率 …… 174, 189, 193, 199
データの検証 …… 337
データマイニング …… 399
テーブル関数 …… 72
適合率 …… 390
出口ページ …… 262
デシル分析 …… 155
登録数 …… 170
読了率 …… 298
度数 …… 122
度数分布表 …… 122
土日・祝日 …… 325
トランザクションデータ …… 15

ナ行
二乗平均平方根 …… 45
年齢 …… 49
年齢別区分 …… 139, 143

ハ行
ヒストグラム …… 122
ビッグデータ …… 3
日付/時刻関数 …… 31, 228, 256, 328
一人あたり …… 133
日跨ぎ考慮 …… 328
表関数 …… 72
標準偏差 …… 438
ファンチャート …… 119
フォーム直帰率 …… 310
フォールアウト …… 285
俯瞰図 …… 302
フレーム指定 …… 63
平均 …… 95
ページ価値 …… 273
ページビュー …… 245
ベクトルの内積 …… 410, 417
偏差値 …… 438
ベン図 …… 148
訪問回数 …… 245
訪問者数 …… 245

マ行
マスタデータ …… 16
文字列関数 …… 31, 256

ヤ行
ユークリッド距離 …… 46
ユーザーフロー …… 291
ユニークユーザー …… 133
曜日 …… 256

ラ行
ランディングページ …… 262
リードタイム …… 228
離脱ページ …… 262
離脱率 …… 266, 307
リピート …… 207
リフト …… 400
流入元 …… 251
利用率 …… 133
累計 …… 98
レコメンドシステム …… 407
ログデータ …… 16, 19
ログフォーマット …… 459

謝辞

本書を執筆するにあたり、多くの方々のご協力をいただきました。編集の丸山弘詩氏には企画段階から常に適切なアドバイスと多大なサポートをしていただきました。川畑雄補氏からは本書執筆のきっかけをいただきました。この場をお借りて深くお礼申し上げます。
深澤充子氏には素敵な表紙カバーと本文デザインを用意していただき、荻野博章氏には、すばらしいイラストで理解しやすい紙面にしていただきました。また、西山悠太朗氏には査読ならびに様々なご意見を頂戴し、中野裕貴氏ならびに領家飛鳥氏にはテストデータの作成や査読、藤岡華子氏には各種ミドルウェアの検証にご協力いただきました。その他、本書を執筆するにあたって、数多くの方々のご協力をいただきました。
最後に、執筆に集中できる環境を用意してくれ、常に支えてくれた家族・同僚に感謝します。家族・同僚の理解があってこそ、本書の執筆に専念できました。皆様には、この場をお借りして感謝の気持ちを伝えたいと思います。

加嵜長門、田宮直人

著者プロフィール

加嵜 長門（カサキ ナガト）
株式会社DMM.comラボ所属。慶應義塾大学大学院 政策・メディア研究科修士課程修了。大学院や学生ベンチャーにて、マルチメディアデータベースを対象とした検索やレコメンドアルゴリズムの研究およびサービス開発に従事し、現在DMM.comラボではビッグデータ活用基盤の構築に携わり、SparkやSQL on Hadoopを用いたレコメンド機能、ビッグデータ活用の研究開発を担当。共著に『詳解Apache Spark』（技術評論社刊）。

田宮 直人（タミヤ ナオト）
データコンサルタント。エンジニアとして大手新聞社の関連サービス、求人サービス、コミュニティサービスの開発に携わり、株式会社サイバーエージェント在籍時にデータアナリストへ転身、株式会社DMM.comラボではマーケティング開発部マネージャーとしてビッグデータ部を立ち上げる。現在はフリーランスとして、データの解析のみならず、データ解析環境の設計・構築、ログの設計、レコメンドAPIの作成など、データに関連する業務全般を担当している。

編集者プロフィール

丸山 弘詩（マルヤマ ヒロシ）
書籍編集者。早稲田大学政治経済学部経済学科中退。国立大学大学院博士後期課程編入（システム生産科学専攻）、単位取得の上で満期退学。大手広告代理店勤務を経て、現在は書籍編集に加え、さまざまな分野のコンサルティング、プロダクトディレクション、開発マネージメントなどを手掛ける。著書は『スマートフォンアプリマーケティング 現場の教科書』（マイナビ出版刊）など多数。

STAFF

編集：丸山 弘詩（Hecula, Inc.）
ブックデザイン：Concent,Inc.（深澤 充子）
本文イラスト：荻野 博章
DTP：Hecula, Inc.
編集部担当：角竹 輝紀

ビッグデータ分析・活用のためのSQLレシピ

2017年3月31日　初版第1刷発行・電子版ver1.00

著　　者　加嵜長門、田宮直人
発 行 者　滝口 直樹
発 行 所　株式会社マイナビ出版
〒101-0003　東京都千代田区一ツ橋2-6-3 一ツ橋ビル 2F
☎ 03-3556-2736（編集）
✉pc-books@mynavi.jp

URL：http://book.mynavi.jp

©2017 Nagato Kasaki , Naoto Tamiya , Hecula.inc

● 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、著者、発行者の許諾を得ずに、無断で複写、複製することは禁じられています。

マイナビBOOKSのWebサイト
プレゼントが当たるアンケートも実施中!

株式会社マイナビ出版の書籍・雑誌の
詳しい情報や購入はこちら。
書籍のサポート情報も公開中。
https://book.mynavi.jp

本書に対するご意見・ご感想をお聞かせ下さい。
アンケートにお答え頂いた方の中から
抽選でプレゼントを差し上げます。
内容の詳細や回答は、下記ページをご覧下さい。
https://book.mynavi.jp/quest/all/

コンピュータ関連の電子書籍が揃うストア

Tech Book Zone Manatee
ITを学ぶ電子書籍ストア［マナティ］
https://book.mynavi.jp/manatee

CONTENTS

Chapter 1　ビッグデータ時代に求められる分析力とは
Chapter 2　本書で扱うツールとデータ群
Chapter 3　データ加工のためのSQL
Chapter 4　売上を把握するためのデータ抽出
Chapter 5　ユーザーを把握するためのデータ抽出
Chapter 6　Webサイトでの行動を把握するためのデータ抽出
Chapter 7　データ活用の精度を高めるための分析術
Chapter 8　データを武器にするための分析術
Chapter 9　知識に留めず行動を起こす

データ加工から売上・ユーザー把握、
レポーティング等々の各種データ分析まで
ビッグデータ
分析・活用のための
SQLレシピ
加嵜長門、田宮直人［著］
「ビッグデータ時代」の
SQL活用術・レシピ集！
PostgreSQL、Apache Hive、Amazon Redshift、
Google BigQuery、SparkSQL対応。
マイナビ