Seu guia intermediário para SQL

À medida que você se sentir mais confortável com SQL, você será capaz de executar consultas ainda mais avançadas. Este guia aprofundado dará a você uma introdução mais detalhada para algumas funções de SQL que você

já aprendeu e dará a você algumas ferramentas novas para trabalhar. Certifique-se de salvar este guia, para que você possa consultar facilmente essas dicas úteis no futuro.

# Estrutura SQL para consultas mais complexas

Você aprenderá mais sobre essas cláusulas e expressões nas seções abaixo, mas primeiro, vejamos como você pode estruturar uma consulta SQL mais envolvida:

**SELECT** Coluna que você deseja ver

**FROM** Tabela onde estão os dados

**WHERE** Determinada condição nos dados

**GROUP BY** Coluna pela qual você deseja agregar

**HAVING** Determinada condição na agregação

**ORDER BY** Coluna pela qual você deseja ordenar os resultados e em ordem ASCENDENTE ou DESCENDENTE

**LIMIT** Número máximo de linhas que você deseja que seus resultados contenham

# Diferentes tipos de instrução JOIN

A maioria dos analistas vai usar **INNER JOIN** ou **LEFT JOIN** ao longo de suas carreiras. Ao juntar tabelas, você está combinando dados de uma tabela com dados de outra conectada por um campo comum. Por exemplo, digamos que você tenha as cores favoritas de seus amigos em uma tabela e os filmes favoritos de seus amigos em outra. Você pode ter as cores e filmes favoritos deles em uma tabela juntando as duas tabelas nos nomes de seus amigos, que é o campo que elas têm em comum. Esse é seu campo JOIN. No trabalho, um campo JOIN geralmente é algum tipo de ID, como o ID\_do\_cliente ou o ID\_da\_conta.

Visualização de tabela

|  |  |
| --- | --- |
| **Favorite\_Colors** | **Favorite\_Movies** |
| friend (string) | friend (string) |
| color (string) | movie (string) |

Visualização de dados

|  |  |
| --- | --- |
| **Favorite\_Colors** | |
| **friend** | **color** |
| Rachel DeSantos | blue |
| Sujin Lee | green |
| Najil Okoro | red |
| John Anderson | orange |

|  |  |
| --- | --- |
| **Favorite\_Movies** | |
| **friend** | **movie** |
| Rachel DeSantos | Avengers |
| Sujin Lee | Despicable Me |
| Najil Okoro | Frozen |

Então, nesse exemplo, você deve usar INNER JOIN se quiser ver apenas informações de amigos que têm uma cor favorita e um filme favorito. Isso significa que se John Anderson tiver uma cor favorita, mas nenhum filme favorito, John Anderson não aparecerá em seus resultados. Os amigos devem estar em ambas as tabelas para aparecer em seus resultados. Portanto, a instrução INNER JOIN é útil quando você quer ver dados em que a chave JOIN existe em ambas as tabelas, o que geralmente é o motivo pelo qual você quer juntar os conjuntos de dados inicialmente. Normalmente, analistas usarão INNER JOIN na maioria das vezes.

**SELECT**

**FROM**

**friend, color, movie**

**Favorite\_Colors AS c**

**INNER JOIN**

**Favorite\_Movies AS m ON c.friend = m.friend**

Resultados:

|  |  |  |
| --- | --- | --- |
| **friend** | **color** | **movie** |
| Rachel DeSantos | blue | Avengers |
| Sujin Lee | green | Despicable Me |
| Najil Okoro | red | Frozen |

Como essa consulta usou um INNER JOIN, os resultados têm apenas três dos quatro amigos. Como um lembrete rápido, isso ocorre porque consultas de INNER JOIN só retornam resultados onde o campo JOIN – neste caso, “friend”– existe em ambas as tabelas. Já que John Anderson não existe na tabela de filmes favoritos, ele é excluído dos resultados da consulta.

Agora, digamos que você queira usar um LEFT JOIN para obter informações de todos os seus amigos em uma tabela (por exemplo, tabela de cores favoritas) com dados adicionados de outra tabela (por exemplo, tabela de filmes favoritos), se existir. Assim, se John Anderson tiver uma cor favorita, mas nenhum filme favorito, ele ainda aparecerá em seus resultados. Ele terá apenas um campo vazio (que é nulo) para seu filme favorito. Na maioria das vezes, a instrução LEFT JOIN é usada se os dados que você está tentando extrair de outra tabela são opcionais. Esse é um campo interessante de se ter, mas não é necessário para sua análise, pois você pode obter valores nulos. No trabalho, você descobrirá que analistas usarão LEFT JOIN com menos frequência do que INNER JOIN.

## SELECT

**friend, color, movie**

## FROM

**Favorite\_Colors AS c**

## LEFT JOIN

**Favorite\_Movies AS m ON c.friend = m.friend**

Resultados:

|  |  |  |
| --- | --- | --- |
| **friend** | **color** | **movie** |
| Rachel DeSantos | blue | Avengers |
| Sujin Lee | green | Despicable Me |
| Najil Okoro | red | Frozen |
| John Anderson | orange | null |

Então agora você sabe a diferença entre INNER JOIN e LEFT JOIN. Você sabe que a instrução INNER JOIN será o tipo de JOIN mais usado, porque ele geralmente se alinha com os casos de uso de negócios.

Outro motivo pelo qual a instrução INNER JOIN é usada é porque resultam em menos dados, uma vez que a chave JOIN deve existir em ambas as tabelas. Isso significa que consultas com INNER JOIN tendem a ser executadas mais rapidamente e usam menos recursos

do que consultas com LEFT JOIN. Isso pode não ser um problema para a maioria dos analistas, mas se você estiver trabalhando com tabelas muito grandes com mais de 1 milhão de linhas e/ou a depender do dialeto SQL usado, sua consulta pode demorar muito mais para ser executada se você usar LEFT JOIN ao invés de INNER JOIN.

Basicamente, a lição aqui é usar INNER JOIN o máximo que puder.

# Agregadores tipo SUM() e COUNT()

Os agregadores resumem linhas em um único valor. As funções **SUM()** e **COUNT()** são exemplos de agregadores. Os tipos de agregadores que estarão disponíveis para você dependerão do dialeto SQL que sua empresa usa. Mas os agregadores mais usados como SUM(), COUNT(), MAX() e MIN() estão disponíveis em todos os dialetos SQL, mesmo que haja algumas pequenas diferenças. É fácil verificar como agregadores são formatados para qualquer dialeto com o qual você esteja trabalhando. Há muitos recursos disponíveis online. Basta abrir seu mecanismo de pesquisa favorito e pesquisar pelas funções de agregação que deseja usa e seu dialeto SQL. Por exemplo, procure por “função SUM no SQL Server.” Todos os agregadores funcionam da mesma forma, então vamos analisar SUM() e COUNT(). A função SUM() obtém uma soma de qualquer coluna que você colocar entre parênteses. A função COUNT() conta o número de entradas em qualquer coluna que você colocar entre parênteses. Por exemplo, digamos que você tenha uma tabela de compras com uma lista das pessoas e o número de ingressos de cinema que elas compraram.

A tabela de compras:

|  |  |
| --- | --- |
| **name** | **tickets** |
| Rachel DeSantos | 3 |
| Sujin Lee | 2 |
| Najil Okoro | 2 |
| John Anderson | 1 |

Consulta:

## SELECT

**SUM(tickets) AS total\_tickets,**

**COUNT(tickets) AS number\_of\_purchases**

## FROM

**purchases**

Resultado:

|  |  |
| --- | --- |
| total\_tickets | number\_of\_purchases |
| 8 | 4 |

Você também pode adicionar uma cláusula **DISTINCT** dentro da função. Isso funcionará para a maioria dos dialetos SQL, mas é sempre bom primeiro verificar e confirmar se a função funciona com o dialeto que sua empresa usa. Adicionar uma cláusula DISTINCT na sua função SUM() ou COUNT() permite fazer uma agregação apenas em cada valor distinto do campo. Confira no exemplo abaixo:

## SELECT

**SUM(tickets) AS total\_tickets,**

**SUM(DISTINCT tickets) AS total\_distinct\_tickets,**

**COUNT(tickets) AS number\_of\_purchases,**

**COUNT(DISTINCT tickets) AS**

**number\_of\_distinct\_purchases**

## FROM

**purchases**

Resultado:

|  |  |  |  |
| --- | --- | --- | --- |
| **total\_tickets** | **total\_distinct\_tickets** | **number\_of\_purchases** | **number\_of\_distinct\_ purchases** |
| 8 | 6 | 4 | 3 |

Você pode perceber que os resultados contêm números menores para as colunas com DISTINCT nelas. Isso porque DISTINCT diz ao SQL para agregar apenas valores únicos. Para entender melhor, verifique a segunda coluna para total\_distinct\_tickets, que demonstra como DISTINCT pode ser usado com uma função SUM(). Mas, neste exemplo, não faz sentido fazer uma soma de valores distintos. Você provavelmente nunca usará DISTINCT com funções SUM(). Em vez disso, você provavelmente usará DISTINCT com funções COUNT(), já que é útil para identificar casos únicos.

# Uso de GROUP BY com agregadores

Anteriormente, ao aprender sobre SUM() e COUNT() com compras de ingressos de cinema, você resumiu os dados para obter o número total de ingressos comprados e o número total de compras feitas com SUM() e COUNT(). Quando você usa funções de agregação como SUM() e COUNT() com uma cláusula **GROUP BY**, os grupos são resumidos em fatias especificadas pela cláusula GROUP BY.

Por exemplo, vamos fingir que sua tabela de compras é como a abaixo, onde a transação de cada pessoa foi para uma determinada ocasião. Você deve usar uma cláusula GROUP BY se quiser obter o número total de ingressos vendidos e o número total de compras feitas pelo tipo de ocasião. Você notará que, se quiser agregar por algo (por exemplo, ocasião), poderá usar a cláusula GROUP BY. Desta forma, SQL é bastante intuitivo.

A nova tabela de compras:

|  |  |  |
| --- | --- | --- |
| **occasion** | **name** | **tickets** |
| fun | Rachel DeSantos | 5 |
| date | Sujin Lee | 2 |
| date | Najil Okoro | 2 |
| fun | John Anderson | 3 |

Consulta:

## SELECT

**occasion,**

**SUM(tickets) AS total\_tickets,**

**COUNT(tickets) AS number\_of\_purchases**

## FROM

**purchases**

## GROUP BY

**occasion**

Resultados:

|  |  |  |
| --- | --- | --- |
| **occasion** | **total\_tickets** | **number\_of\_purchases** |
| fun | 8 | 2 |
| date | 4 | 2 |

Ótimo! Agora você sabe como usar a cláusula GROUP BY e quando você deve usá-la. Aqui está outra coisa interessante de se saber: você pode usar o número da coluna na cláusula GROUP BY para especificar o que você deseja agrupar em vez de usar os nomes das colunas. No último exemplo, você queria agrupar por ocasião. Ocasião é a primeira coluna escrita na consulta SQL. Isso significa que é possível escrever GROUP BY 1 ao invés de GROUP BY ocasião. Se ocasião fosse a segunda coluna na cláusula SELECT, então você escreveria GROUP BY 2. Veja abaixo:

Consulta:

## SELECT

**occasion,**

**SUM(tickets) AS total\_tickets,**

**COUNT(tickets) AS number\_of\_purchases**

## FROM

**purchases**

## GROUP BY

**occasion**

É o mesmo que:

## SELECT

**occasion,**

**SUM(tickets) AS total\_tickets,**

**COUNT(tickets) AS number\_of\_purchases**

## FROM

**purchases**

## GROUP BY

**1**

Conhecer esse atalho pode economizar tempo ao escrever suas consultas SQL e ao agrupar por vários campos. Nesse caso, você só precisa separá-los com vírgulas (por exemplo, GROUP BY 1, 2, 3, 4).

# Quando usar HAVING

A cláusula HAVING é semelhante à cláusula WHERE, pois filtra os dados com base em determinadas condições. Mas essas cláusulas são usadas em situações diferentes. A cláusula **WHERE** é usada para fazer filtros em sua tabela, como um filtro para determinados intervalos de datas ou países específicos. A cláusula **HAVING** é usada para fazer filtros em suas agregações e deve ser emparelhada com uma cláusula GROUP BY.

A tabela de compras:

|  |  |  |
| --- | --- | --- |
| **occasion** | **name** | **tickets** |
| fun | Rachel DeSantos | 5 |
| date | Sujin Lee | 2 |
| date | Najil Okoro | 2 |
| fun | John Anderson | 3 |

Neste exemplo, você notará que pode aplicar camadas na cláusula HAVING se quiser definir limites em sua agregação, ou a soma e a contagem neste caso:

Consulta:

## SELECT

**occasion,**

**SUM(tickets) AS total\_tickets,**

**COUNT(tickets) AS number\_of\_purchases**

## FROM

**purchases**

## GROUP BY

**occasion HAVING**

**SUM(tickets) > 5**

Resultados:

|  |  |  |
| --- | --- | --- |
| **occasion** | **total\_tickets** | **number\_of\_purchases** |
| fun | 8 | 2 |

É importante observar que seus resultados não contêm mais a ocasião ‘data’. Isso ocorre porque sua cláusula HAVING filtra por somas maiores que 5. A ocasião ‘data’ teve apenas 4 ingressos totais, o que é menos que 5, logo, a ocasião ‘data’ não aparece em seus resultados.

Ótimo trabalho! Agora você sabe como e quando usar a cláusula HAVING. Como analista de dados, você usará muitas cláusulas WHERE e apenas algumas cláusulas HAVING. Isso ocorre por causa do caso de uso de negócios, mas também por causa de recursos, assim como INNER JOIN vs. LEFT JOIN. Se sua consulta contiver uma cláusula HAVING, levará mais tempo para ser executada e consumirá mais recursos, porque o SQL precisa filtrar depois de executar os cálculos SUM() e COUNT(). Então, é uma boa ideia tentar minimizar o uso da cláusula HAVING sempre que possível. Porém, se você precisar usar HAVING, tente usar tabelas temporárias.

# Uso de ORDER BY para organizar seus resultados

A cláusula **ORDER BY** ajuda a organizar seus resultados. Ela vai no final da consulta SQL e é a última cláusula a ser usada, a menos que você tenha uma cláusula LIMIT.

Uma versão ligeiramente modificada da tabela de compras:

|  |  |
| --- | --- |
| **name** | **tickets** |
| Rachel DeSantos | 3 |
| Sujin Lee | 5 |
| Najil Okoro | 2 |
| John Anderson | 4 |

Digamos que queremos que todos nesta tabela sejam organizados pelo número de ingressos que compraram, do maior para o menor, ou em ordem decrescente.

## SELECT

**name,**

**tickets**

## FROM

**purchases**

## ORDER BY

**tickets DESC**

Resultado:

|  |  |
| --- | --- |
| **name** | **tickets** |
| Sujin Lee | 5 |
| John Anderson | 4 |
| Rachel DeSantos | 3 |
| Najil Okoro | 2 |

Se você quiser mostrar primeiro a pessoa com a menor quantidade de ingressos, você deve ordenar seus resultados em ordem CRESCENTE. Para fazer isso no SQL, você pode usar ASC ou deixá-lo em branco, pois SQL ordena as colunas em ordem CRESCENTE por padrão. Mas, a melhor prática é escrever ASC ou DESC para que essa cláusula fique clara para todos que leiam sua consulta.

# Quando usar LIMIT

A cláusula **LIMIT** é útil quando você quer trabalhar apenas com um número selecionado de linhas. Isso geralmente é usado em duas situações.

Na primeira situação, digamos que você queira os X maiores números de casos. No exemplo do ingresso de cinema, digamos que você queira apenas as 3 maiores compras. Você poderia usar uma cláusula LIMIT como abaixo.

Consulta:

## SELECT

**name,**

**tickets**

## FROM

**purchases**

## ORDER BY

**tickets DESC**

**LIMIT 3 --apenas os 3 maiores resultados**

Resultado:

|  |  |
| --- | --- |
| **name** | **tickets** |
| Sujin Lee | 5 |
| John Anderson | 4 |
| Rachel DeSantos | 3 |

Na segunda situação, digamos que você queira trabalhar com todo o conjunto de dados antes de escrever sua consulta. Nesse caso, você usaria uma cláusula LIMIT para não desperdiçar recursos extraindo cada linha.

Consulta:

## SELECT

**name,**

**tickets**

## FROM

**purchases**

## ORDER BY

**tickets DESC**

**LIMIT 20 --apenas os 20 maiores resultados**

Resultado:

|  |  |
| --- | --- |
| **name** | **tickets** |
| Rachel DeSantos | 3 |
| Sujin Lee | 5 |
| Najil Okoro | 2 |
| John Anderson | 4 |

Você pode ter notado que tem apenas quatro linhas de dados em seus resultados, embora você tenha definido um limite de

1. Isso porque a tabela de compras contém apenas quatro linhas de dados. A cláusula LIMIT é um número máximo de linhas a serem exibidas. E se a tabela de compras contiver um milhão de linhas, apenas 20 linhas serão exibidas. Mas, como a tabela de compras contém menos de 20 linhas, todos os dados são exibidos.

# Expressões condicionais como CASE, IF e COALESCE()

**Expressões condicionais** como instruções CASE e a função COALESCE() são usadas quando você deseja mudar como seus resultados são apresentados. Embora você esteja definindo condições em expressões condicionais, as expressões condicionais diferem do tipo de condições que você coloca em uma cláusula WHERE. As condições postas em cláusulas WHERE se aplicam a toda a consulta, enquanto expressões condicionais se aplicam apenas

a esse campo específico. Além disso, você pode alterar como seus resultados são apresentados com expressões condicionais, o que é algo que você não pode fazer com instruções WHERE. Vejamos as três expressões condicionais mais comuns: CASE, IF e COALESCE().

# Instruções CASE:

As instruções CASE são as mais usadas como rótulos em seu conjunto de dados. Você pode usar instruções CASE para rotular linhas que atendam a determinada condição como X e linhas que atendam a outra condição como Y. É por isso que você as encontrará comumente usadas com funções de agregação quando quiser agrupar coisas por categorias. Aqui está um exemplo de uso de uma tabela de filmes que estão em cartaz no cinema local:

A tabela MovieTheater:

|  |  |
| --- | --- |
| **genre** | **movie\_title** |
| horror | Silence of the Lambs |
| comedy | Jumanji |
| family | Frozen 2 |
| documentary | O 13 |

Digamos que você deseja agrupar esses filmes em filmes que você assistirá e filmes que você não assistirá e contar o número de filmes que se enquadram em cada categoria. Sua consulta seria:

## SELECT CASE

**WHEN genre = ‘horror’ THEN ‘will not watch’**

**ELSE ‘will watch’**

**END AS watch\_category, --criar sua própria categoria COUNT(movie\_title) AS number\_of\_movies**

## FROM

**MovieTheater GROUP BY**

**1 --ao agrupar por CASE, use números de posição ou digite a instrução CASE inteira aqui**

Resultados:

|  |  |
| --- | --- |
| **watch\_category** | **number\_of\_movies** |
| Will not watch (Não assistirei) | 1 |
| Will watch (Assistirei) | 3 |

Você pode notar que adicionou seus próprios rótulos ao conjunto de dados, o que pode ser feito com instruções CASE. Mas lembre-se de que esse recurso não está em todos os dialetos SQL, incluindo BigQuery. Se você quiser

saber mais, revise a documentação de COUNT() ou SUM() para seu dialeto SQL específico e verifique como instruções CASE podem ser usadas.

Há também outra maneira de usar instruções CASE no BigQuery (novamente, isso pode não se aplicar a todos os dialetos SQL). Se suas condições forem correspondentes, como no exemplo acima, então você poderá escrever sua instrução CASE como (compare lines 2-3):

## SELECT

**CASE genre**

**WHEN ‘horror’ THEN ‘will not watch’ ELSE ‘will watch’**

**END AS watch\_category**

**COUNT(movie\_title) AS number\_of\_movies**

## FROM

**MovieTheater**

## GROUP BY

**1**

Isso produzirá os mesmos resultados, mas não é recomendado, porque se limita a apenas condições de correspondência (por exemplo, genre = ‘horror’). Em comparação, a versão anterior com WHEN genre = ‘horror’ é flexível e pode aceitar outros tipos de condições, como maior que (>), menor que (<), diferente de (<> or !=), etc.

# Instruções IF

A seguir, são as instruções IF. As **instruções IF** são semelhantes às instruções CASE, porém, há uma diferença fundamental: As instruções CASE podem aceitar várias condições, enquanto as instruções IF não podem. No exemplo acima, você só tinha uma condição (por exemplo, WHEN genre = ‘horror’), então também poderia ter usado uma instrução IF como:

## SELECT

**IF(genre=’horror’, ‘will not watch’, ‘will watch’) AS watch\_category,**

**COUNT(movie\_title) AS number\_of\_movies**

## FROM

**MovieTheater**

## GROUP BY

**1**

Mas, se você tiver várias condições, convém usar uma instrução CASE como:

## SELECT

**CASE**

**WHEN genre = ‘horror’ THEN ‘will not watch’**

**WHEN genre = ‘documentary’ THEN ‘will watch alone’**

**ELSE ‘watch with others’ END AS watch\_category,**

## FROM

**COUNT (movie\_title) AS number\_of\_movies**

**MovieTheater**

## GROUP BY 1

Resultados:

|  |  |
| --- | --- |
| **watch\_category** | **number\_of\_movies** |
| Will not watch (Não assistirei) | 1 |
| Will watch alone (Assistirei só) | 1 |
| Watch with others (Assistirei com outras pessoas) | 2 |

# Função COALESCE()

Finalmente, existe a função **COALESCE()**. Esta função é usada para retornar a primeira expressão não nula na ordem especificada na função. Ela é útil quando seus dados estão espalhados em várias colunas. Por exemplo, digamos que você tenha uma tabela de filmes como linhas, colunas como meses e valores como 1 se o filme foi lançado naquele mês ou nulo se não. Veja a tabela MovieLaunches abaixo:

|  |  |  |  |
| --- | --- | --- | --- |
| **movie\_title** | **Jan\_2030** | **Feb\_2030** | **Mar\_2030** |
| Avengers X | 1 | *null* | *null* |
| Frozen V | *null* | 1 | *null* |
| Lion King IV | *null* | *null* | *null* |

Se você quiser saber se esses filmes foram lançados entre janeiro e março de 2030, use COALESCE. Por exemplo:

## SELECT

**movie\_title,**

**COALESCE(Jan\_2030, Feb\_2030, Mar\_2030) AS**

**launched\_indicator**

## FROM

**MovieLaunches**

Resultados:

|  |  |
| --- | --- |
| **movie\_title** | **launched\_indicator** |
| Avengers X | 1 |
| Frozen V | 1 |
| Lion King IV | *null* |

Você pode notar que dois dos três filmes tinham valores não nulos em campos especificados (Jan\_2030, Feb\_2030, Mar\_2030). Este exemplo demonstra como a função COALESCE funciona. Ela pesquisará em cada coluna que você especificar dentro da função e tentará retornar a um valor não nulo, se encontrar um.

No trabalho, COALESCE é frequentemente usada para garantir que os campos não contenham nulos. Assim a instrução COALESCE poderia ser: COALESCE(try\_this\_field, then\_this\_field, 0) para informar ao SQL para verificar os dois primeiros campos por um valor não nulo. Se nenhum existir nesses campos, atribua um zero no lugar de um nulo. No BigQuery, este é o mesmo que usar a função IFNULL() (mais sobre isso [aqui](https://cloud.google.com/bigquery/docs/reference/standard-sql/conditional_expressions#ifnull)). Outros dialetos SQL podem não ter a função IFNULL() e, nesse caso, COALESCE() é usada ao invés disso.

# Criação e exclusão de tabelas

Dados no SQL são armazenados em tabelas. Este guia tem feito referência a tabelas pequenas como as tabelas de compras, MovieLaunches e MovieTheater. Essas são tabelas que já existem, porque você não as criou.

# Criação de tabelas

A situação ideal para criar uma tabela é se as três seguintes condições forem atendidas:

* 1. Consulta complexa contendo várias instruções JOIN
  2. A saída do resultado é uma tabela
  3. Você precisa executar a consulta com frequência ou regularmente

Se essas condições forem atendidas, é uma ótima ideia criar uma tabela de relatórios. Mas é prática recomendada verificar com seu gerente ou colegas de equipe antes disso, caso você precise acessar permissões para criar uma tabela de relatórios.

A sintaxe para a criação de tabelas mudará a depender do dialeto SQL e da plataforma SQL que você usa. Você verá como criar tabelas em BigQuery aqui, porém, se sua empresa usa um dialeto SQL diferente, é uma boa ideia pesquisar online como criar tabelas nesse dialeto SQL (por exemplo, “Criar tabelas em PostgreSQL”). Ou melhor ainda, peça ajuda a seu gerente ou colega de equipe.

De modo geral, quando você cria tabelas, você quer se certificar de que a mesma tabela ainda não existe. Isso porque se você tentar criar uma tabela que já existe, a consulta retornará um erro.

A verificação de tabelas existentes será diferente entre dialetos SQL, mas sempre é bom verificar para

evitar erros desnecessários no seu trabalho.

A maneira de verificar tabelas pré-existentes em BigQuery é:

**CREATE TABLE IF NOT EXISTS mydataset.FavoriteColorAndMovie**

**AS**

## SELECT

**friend,**

**color,**

**movie**

## FROM

**Favorite\_Colors AS c**

## INNER JOIN

**Favorite\_Movies AS m ON c.friend = m.friend**

Você já criou uma tabela FavoriteColorAndMovie! Você pode fazer referência a esta tabela única para encontrar a cor favorita e/ou filme favorito de cada amigo sem ter que juntar as duas tabelas separadas, Favorite\_Colors e Favorite\_Movies, a cada vez.

O método CREATE TABLE IF NOT EXISTS é melhor se as tabelas em sua consulta (por exemplo, Favorite\_Colors e Favorite\_Movies) não estiverem sendo atualizadas continuamente. O CREATE TABLE IF NOT EXISTS não fará nada se a tabela já existir porque ela não será atualizada. Portanto, se as tabelas de fonte forem continuamente atualizadas (por exemplo, novos amigos adicionados continuamente com as cores e filmes favoritos deles), é melhor usar um método diferente de criação de tabelas.

Esse outro método de criar tabelas é o método CREATE OR REPLACE TABLE. Aqui está como isso funciona:

**CREATE TABLE OR REPLACE TABLE mydataset.FavoriteColorAndMovie**

**AS**

## SELECT

**friend,**

**color,**

**movie**

## FROM

**Favorite\_Colors AS c**

## INNER JOIN

**Favorite\_Movies AS m ON c.friend = m.friend**

Você pode notar que a única diferença entre as duas maneiras de criar tabelas é a primeira linha da consulta. Isso informa ao SQL o que fazer se a tabela já existir. CREATE TABLE IF NOT EXISTS só criará uma tabela se ela ainda não existir. Se existir, a consulta será executada, mas não fará nada. Isso é uma segurança para que você não sobrescreva acidentalmente uma tabela potencialmente importante. Alternativamente, se precisar sobrescrever uma tabela, você pode usar CREATE OR REPLACE TABLE.

Em resumo, você usaria CREATE TABLE IF NOT EXISTS se estiver criando uma tabela estática que não precisa ser atualizada. Se sua tabela precisar ser atualizada continuamente, você usaria CREATE OR REPLACE TABLE em vez disso.

# Exclusão de tabelas

Agora, vamos falar sobre a exclusão de tabelas. Isso raramente acontece, mas é importante aprender sobre porque, uma vez que as tabelas são excluídas, os dados contidos nelas podem ser perdidos. E, já que os dados são de propriedade da empresa para a qual você trabalha, excluir uma tabela pode significar se livrar de algo que não é seu.

Pense em como você trataria uma conta de mídia social que não lhe pertence. Por exemplo, se sua empresa lhe deu acesso à conta dela. Você pode ver postagens e talvez (com permissão) criar sua própria postagem para ela, mas não excluiria uma postagem de mídia social porque essa é a conta do proprietário, não sua.

Se você pensar em apertar o botão delete para uma tabela que você não criou, garanta que verificou os motivos pelos quais você está pressionando delete e confira novamente com seu gerente apenas por precaução.

Mas, se você precisar excluir uma tabela, especialmente se for uma que você criou e não precisa mais, você pode excluir uma tabela em BigQuery usando esta consulta:

**DROP TABLE IF EXISTS mydataset.FavoriteColorAndMovie**

**DROP TABLE** informa ao SQL para excluir a tabela e a parte IF EXISTS garante que você não receba um erro se a tabela não existir. Essa é uma segurança, de modo que, se a tabela existir, ela será descartada. Se a tabela não existir e você executar essa consulta, nada acontecerá. Então, de qualquer forma, a segurança funciona a seu favor. A melhor prática é adicionar o IF EXISTS.

# Tabelas temporárias

Até aqui, você aprendeu como criar tabelas e quando deseja criá-las. Sinta-se à vontade para revisar isso na seção acima se precisar de um lembrete. As tabelas que você criar com o método CREATE TABLE IF NOT EXISTS ou o método CREATE OR REPLACE TABLE são tabelas permanentes. Elas podem ser compartilhadas e vistas por outras pessoas e podem ser acessadas posteriormente.

Mas, pode haver situações em que você não precise criar tabelas permanentes. Lembre-se: armazenar dados no SQL custa dinheiro e recursos para a empresa. Se não precisa de uma tabela permanente, você pode criar tabelas temporárias em vez disso. As tabelas temporárias existem apenas na sua sessão (ou por até 24 horas, dependendo da sua plataforma SQL) e não podem ser compartilhadas ou visualizadas por outras pessoas. As tabelas temporárias existem apenas para você durante sua sessão. Pense em tabelas temporárias como um bloco de rascunhos onde você pode rabiscar seus cálculos antes de colocar sua resposta final.

Comecemos descrevendo quando você deve querer criar uma tabela permanente versus uma tabela temporária.

Aqui estão as três condições para quando você deseja criar uma tabela permanente. Todas as três condições devem ser atendidas.

1. Consulta complexa contendo várias instruções JOIN
2. A saída do resultado é uma tabela
3. Você precisa executar a consulta com frequência ou regularmente

Alternativamente, as **tabelas temporárias** são usadas para dividir consultas complexas em incrementos menores. Essas consultas complexas podem conter várias instruções JOIN, mas não necessariamente. Você pode quer usar tabelas temporárias se uma ou mais das seguintes condições se aplicarem:

* Consulta de execução lenta com várias instruções JOIN e instruções WHERE
* Consulta de execução lenta contendo GROUP BY e HAVING
* Consultas aninhadas (ou seja, consulta dentro de uma consulta)
* Se você precisar fazer um cálculo em cima de um cálculo (por exemplo, obter a soma por dia, em seguida, a média das somas do dia)

Se alguma das condições acima for atendida, o uso de uma tabela temporária pode acelerar sua consulta, o que facilitará a elaboração da consulta, assim como facilitará a solução de problemas de sua consulta se algo der errado.

Veja como criar uma tabela temporária:

**CREATE TEMP TABLE ExampleTable**

**AS**

## SELECT

**colors**

## FROM

**Favorite\_Colors**

Agora, voltemos para a tabela permanente que você criou anteriormente:

**CREATE TABLE IF NOT EXISTS mydataset.FavoriteColorAndMovie AS**

## SELECT

**friend,**

**color,**

**movie**

## FROM

**Favorite\_Colors AS c**

## INNER JOIN

**Favorite\_Movies AS m ON c.friend = m.friend**

Essa consulta funciona para uma tabela permanente porque todas as três condições mencionadas anteriormente foram atendidas. Mas, para tabelas temporárias, essa consulta não é uma boa ideia. Você está trabalhando com várias instruções JOIN, mas não há instruções WHERE, e a consulta é executada muito rapidamente, já que as tabelas Favorite\_Colors e Favorite\_Movies são relativamente pequenas (<100k linhas).

Consideremos um cenário diferente quando você quiser usar tabelas temporárias. Anteriormente, você aprendeu sobre GROUP BY e HAVING. Se sua consulta contiver ambas as cláusulas, como a abaixo, convém usar tabelas temporárias se sua consulta estiver sendo executada lentamente.

Consulta de tabela temporária (se sua consulta estiver sendo executada lentamente):

## SELECT

**occasion,**

**SUM(tickets) AS total\_tickets,**

**COUNT(tickets) AS number\_of\_purchases**

## FROM

**purchases**

## GROUP BY

**occasion**

**HAVING**

**SUM(tickets) > 5**

Na consulta acima, o SQL deve executar três ações. Primeiro, agrupar sua tabela por ocasião. Segundo, obter SUM() e COUNT() da coluna de ingressos. Terceiro, mostrar apenas ocasiões com um SUM() de ingressos maior que cinco. Se a tabela de compras fosse muito maior (mais de 1 milhão de linhas) e você também tivesse várias instruções JOIN nessa tabela, sua consulta provavelmente seria executada lentamente. Mas você pode evitar isso usando a cláusula HAVING e acelerar sua consulta dividindo a consulta em duas etapas usando as tabelas temporárias.

Primeiro, você pode fazer as agregações GROUP BY:

**CREATE TEMP TABLE TicketsByOccasion AS**

## SELECT

**occasion,**

**SUM(tickets) AS total\_tickets,**

**COUNT(tickets) AS number\_of\_purchases**

## FROM

**purchases**

## GROUP BY

**occasion;**

Então, você pode fazer a limitação HAVING como uma condição WHERE:

## SELECT

**occasion,**

**total\_tickets,**

**number\_of\_purchases**

## FROM WHERE

**TicketsByOccasion total\_tickets > 5**

Há três coisas importantes a serem observadas aqui:

1. Se você estiver executando duas consultas ao mesmo tempo, o que deve ser feito com tabelas temporárias (apenas disponíveis durante aquela sessão), precisará de um ponto e vírgula separando cada consulta.
2. A primeira consulta é onde você criou a tabela temporária. A segunda consulta faz referência a essa

tabela temporária e aos campos dela. É por isso que você pode acessar a soma de ingressos como total\_tickets na segunda consulta.

1. Ao criar nomes de tabelas, não use espaços ou a consulta retornará um erro. A melhor prática é usar a capitalização CamelCase ao nomear a tabela que você está construindo. A **capitalização CamelCase** significa que o início de cada palavra será maiúscula sem espaço entre elas, assim como a corcova de um camelo. Por exemplo, a tabela TicketsByOccasion utiliza o padrão CamelCase.

Em resumo, você não precisa usar tabelas temporárias, mas elas podem ser uma ferramenta muito útil para ajudar a dividir consultas complexas ou complicadas em etapas menores e mais gerenciáveis.

# Conclusão

Este guia abrange muitos conceitos, mas você pode voltar a ele várias vezes enquanto continua a escrever consultas SQL por conta própria. Como você aprendeu ao longo deste curso, a prática é uma parte importante do processo de aprendizagem e, quanto mais você praticar o trabalho no SQL, mais descobertas novas você fará. Você pode salvar este guia para que possa revisar e fazer referência a essas funções e conceitos conforme for preciso.