Aula 3

Preparação e Limpeza de Dados

1. Perfilamento de dados (Data Profiling)

Raramente os dados que coletamos e armazenamos são perfeitos. No mundo real, eles podem ser incompletos, inconsistentes ou simplesmente incorretos. Antes de extrair informações confiáveis ou construir relatórios precisos, precisamos passar por uma etapa crucial conhecida como limpeza ou tratamento de dados.

O primeiro passo nesse processo é o **Perfilamento de Dados** (*Data Profiling*). Esta é uma fase de investigação inicial onde buscamos entender a estrutura, o conteúdo e a qualidade dos nossos dados. As técnicas a seguir nos ajudam a ter uma visão geral e a identificar potenciais problemas.

1.1. Contagem de registros (COUNT(*))

O nosso primeiro passo será saber o "tamanho" da nossa tabela. No caso, queremos contar quantas linhas uma certa coluna tem. Para isso, usamos o comando **COUNT()**, que recebe como parametro o nome de uma coluna, sendo * representado todas as colunas.

```
-- Retorna o número total de clientes na tabela
SELECT COUNT(*) FROM clientes;
```

Note que linhas nulas não serão contadas.

1.2. Valores distintos (COUNT(DISTINCT col))

Outra contagem de dados muito útil é encontrar a quantidade de *linhas distintas*. Por exemplo, vamos supor que você tenha uma tabela de clientes, onde uma das colunas é o país de origem desta pessoa. Se você quiser quantos países diferentes seu serviço atingiu, você pode executar:

```
-- Retorna a quantidade de estados únicos na tabela de clientes
SELECT COUNT(DISTINCT estado) FROM clientes;
```

1.3. Distribuições e frequências (GROUP BY)

O comando COUNT pode ser pareado com o GROUP BY. Como o nome diz, ele agrupa dados de uma tabela, por exemplo, suponhamos uma tabela de clientes onde uma das colunas é o estado, então colocar "GROUP BY estado" vai agrupar as linhas de clientes por estado. Assim, quando você pedir "SELECT COUNT(*)", ele não vai pegar o número total de linhas da tabela, mas sim o número de linhas de cada grupo.

Isso permite que você faça uma averiguação da distribuição do seus dados. Por exemplo: contar a quantidade de clientes em cada estado, ou, usando comando que veremos mais a frente, encontrar a média de idade de seus clientes em cada estado.

```
-- A primeira coluna é o estado, e a outra é o COUNT(*) de clientes nesse estado
SELECT estado, COUNT(*) AS total_clientes
FROM clientes
GROUP BY estado;
```

1.4. Detecção de outliers e ranges (MIN, MAX, AVG)

Outras operações que podem ser feitas para sumarizar seus dados, além do COUNT, são esses comandos estatísticos. Todavia, esses comandos só se aplicam a colunas com tipo de dado numérico.

Os nomes são autoexplicativos: MIN retorna a linha com o menor valor no campo, MAX o de maior valor, AVG é a média (*average*, em inglês). Esses comandos são especialmente úteis para encontrar outliers.

Note que esses comandos, assim como o COUNT, podem ser usados junto com o GROUP BY, permitindo refinar sua análise.

```
-- Mostra a idade mínima, máxima e a média de idade dos clientes
SELECT
    MIN(idade) AS idade_minima,
    MAX(idade) AS idade_maxima,
    AVG(idade) AS idade_media
FROM clientes;
```

1.5. Detecção de duplicatas (GROUP BY ... HAVING COUNT(*) > 1)

Para podermos limpar nossos dados, precisamos ser capazes de analisar linhas que não nos interessa. Um exemplo disso são dados duplicados. Para isso, podemos combinar o GROUP BY e COUNT, visto que dados duplicados serão agrupados juntos, e logo sua contagem será maior que 1:

```
-- Encontra emails duplicados e mostra quantas vezes eles aparecem
SELECT email, COUNT(*)
FROM clientes
GROUP BY email
HAVING COUNT(*) > 1;
```

1.6. Identificação de Valores Nulos

Outro dado indesejado para nós limparmos serão os valores nulos. Para quantificar os campos nulos, podemos ver a diferença entre COUNT(*) e COUNT(coluna). Por exemplo, se nossa tabela houver 100 clientes, então COUNT(*) será 100. Porém se um dos clientes tiver um campo vazio (por exemplo, o campo de telefone), então COUNT(telefone) será 99.

```
-- Retorna a quantidade de valores nulos na coluna 'telefone'
SELECT
    COUNT(*) - COUNT(telefone) AS total_nulos_telefone
FROM clientes;
```

2. Tratamento de nulos e valores inválidos

Raramente os dados que coletamos e armazenamos são perfeitos. No mundo real, os dados podem ser incompletos, inconsistentes ou simplesmente incorretos. Dentre os motivos para isso, estão falhas humanas (como erros de preenchimento de formulários ou de digitação), falhas de sistema e integração de sistemas com diferentes padronizações (um sistema que utiliza a cidade de um cliente como "São Paulo"e outro que utiliza como "SP", por exemplo). Antes de podermos extrair informações confiáveis ou construir relatórios precisos, precisamos passar por uma etapa crucial conhecida como limpeza ou tratamento de dados.

Neste capítulo, vamos realizar o tratamento de valores nulos e de valores inválidos, visando desenvolver uma base confiável para análises futuras.

2.1. Substituição de Valores Nulos

Valores nulos (NULL) caracterizam, em relações, a ausência de informação. Valores desse tipo podem ser um problema, já que operações aritméticas e de agrupamento tendem a retornar NULL, o que pode prejudicar análises futuras. Uma das maneiras de se lidar com valores nulos é realizando uma substituição por valores válidos. Para tanto, temos uma série de comandos que podem ser utilizados, como CASE, COALESCE e NULLIF. No caso do comando CASE, temos a situação semelhante ao if else, onde, se o dado for nulo, realizamos a troca por outro valor, senão mantemos o valor original. Suponha que temos a seguinte relação de um sistema de produtos de um comércio, cada qual com nome, preço base e preço com desconto (sendo esse, se não houver desconto, nulo).

id	nome	preco_padrao	preco_desconto
1	Notebook	4500.00	4250.00
2	Mouse	150.00	NULL
3	Teclado	250.00	NULL
4	Monitor	1200.00	1100.00

Para uma consulta que retorna o preço final de um produto, poderíamos fazer

```
SELECT
    nome AS Nome,
    CASE
        WHEN preco_desconto IS NULL THEN preco_padrao
        ELSE preco_desconto
    END AS Preco_final
FROM
    Produtos;
```

o que retornaria a tabela a seguir:

Nome	Preco_final
Notebook	4250.00
Mouse	150.00
Teclado	250.00
Monitor	1100.00

Analogamente, o comando COALESCE pode ser utilizado para obter os mesmos resultados, mas com escrita mais compacta. Nesse caso, toma-se um conjunto de valores e retorna-se, da esquerda para a direita, o primeiro valor não-nulo. Com isso, o exemplo anterior poderia ser reescrito da seguinte forma:

```
SELECT
    nome,
    COALESCE(preco_desconto, preco_padrao) AS preco_final
FROM
    produtos;
```

Além desses, podemos também utilizar o comando NULLIF. Ele recebe dois valores (VAL1 e VAL2) e realiza a seguinte operação: se VAL1 = VAL2, então retorna NULL; caso contrário, retorna VAL1. Em situações em que, no estágio de captação de dados, um valor NULL é trocado por um valor padrão, podemos utilizar NULLIF para converter o valor padrão para NULL. Por exemplo, se uma coluna de data é foi convertida para um valor padrão '1980-01-01', podemos fazer:

```
NULLIF(data, '1980-01-01')
```

2.2. Eliminação de Registros com Nulos Críticos

Em alguns cenários, podemos ter valores nulos que não podem ser facilmente substituidos e que são fundamentais para a análise dos dados. Nesses casos, uma abordagem de tratamento de NULLs é a eliminação das linhas que possuem NULL. Suponha, por exemplo, que temos uma relação com dados sobre a venda de produtos, um cliente com id nulo não pode ser substituído. Logo, para uma análise sobre os clientes que consomem tais produtos, teríamos que fazer:

```
SELECT *
FROM Vendas
WHERE Id_Cliente IS NOT NULL;
```

2.3. Padronização de Valores Faltantes

Além de valores nulos, a ausência de informação pode ser representada de outras maneiras, conforme a estruturação dos dados. Podemos ter strings vazias (”) ou com valores como ’Desconhecido’ e ’Não Informado’, por exemplo. Apesar de, dos olhos de um analista, tais dados serem iguais, o sistema de banco de dados toma eles como distintos, o que leva à necessidade de padronizá-los. Suponha que tenhamos a seguinte relação de usuários cadastrados num sistema:

Id_Usuario	Nome	Idade	Estado_Civil
1	Gabriela	28	’Solteira’
2	Fernando	34	”
3	Letícia	45	’Casada’
4	Roberto	52	’N/A’
5	Sandra	21	”

Para padronizar os valores nulos, poderíamos fazer:

```
SELECT
    id_usuario,
    CASE
        WHEN estado_civil = '' OR estado_civil = 'N/A' THEN 'Não Informado'
        ELSE estado_civil
    END
FROM
    Usuarios;
```

2.4. Valores fora de Faixa com CASE

Valores fora de faixa são dados que estão fora do domínio ou intervalo esperado para um determinado atributo. Uma desconto percentual de mais de 100% ou uma idade irreal, por exemplo, são casos em que, apesar de o dado existir, ele está incorreto e precisa ser tratado. De maneira geral, podemos tratá-lo utilizando o comando CASE da seguinte forma:

```
CASE
    WHEN atributo COMPARAÇÃO Limite1 THEN Valor1
    WHEN atributo COMPARAÇÃO Limite2 THEN Valor2
    ...
    ELSE atributo
END
```

Por exemplo, se temos uma relação com as notas que os alunos de uma matéria obtiveram (de 0 a 10) e, por descuido de um professor, notas acima de 10 foram postas, podemos visualizar as notas dos alunos da seguinte forma:

```
SELECT
    id_aluno,
    CASE
        WHEN nota > 10 THEN 10
        ELSE nota
    END AS nota_final
FROM
    Alunos;
```

2.5. Normalização de Categorias

Outro problema comum durante o processo de limpeza e tratamento de dados é aquele em que um mesmo valor de um atributo está armazenado em diferentes formas. Para o campo de cidade de um formulário, um usuário poderia inserir "São Paulo", "SP", "sp", "são paulo", etc. Ao realizar um agrupamento sobre as cidades, teremos inúmeros grupos distintos para uma mesma cidade, o que pode levar a problemas de análise. Por conseguinte, há a necessidade de padronização de tais valores. Para tanto, podemo utilizar CASE:

```
CASE
    WHEN cidade IN ("São Paulo", "SP", "sp", "são paulo") THEN "São Paulo"
    ELSE cidade
END AS Cidade
```

Observe que poderiamos utilizar as funções de conversão de strings para caixa alta (UPPER()) ou caixa baixa (LOWER()), o que evita a repetição de strings. O mesmo código acima pode ser reescrito da seguinte forma:

```
CASE
    WHEN LOWER(cidade) IN ("sp", "são paulo") THEN "São Paulo"
    ELSE cidade
END AS Cidade
```

3. Conversão e Padronização de Tipos

A etapa de conversão e padronização de tipos de dados é crucial para garantir que as informações estejam no formato correto para análise e processamento. Dados brutos frequentemente vêm com tipos incorretos (por exemplo, um número armazenado como texto) ou com inconsistências que precisam ser corrigidas.

3.1. Uso de CAST e CONVERT

As funções **CAST()** e **CONVERT()** são as principais ferramentas para alterar explicitamente o tipo de uma coluna. Isso é necessário quando se precisa realizar operações matemáticas em colunas que foram importadas como strings, ou para formatar datas e horas.

- **CAST(expressão AS tipo)**: A sintaxe CAST é padrão em SQL e geralmente é a mais recomendada por ser mais portável entre diferentes bancos de dados.
- **CONVERT(tipo, expressão)**: A sintaxe CONVERT é específica para alguns sistemas de banco de dados, como o SQL Server.

Exemplo Prático:

Imagine uma tabela de vendas onde o preço foi importado como texto.

```
SELECT
'150.75' AS PrecoTexto,
CAST('150.75' AS DECIMAL(10, 2)) AS PrecoNumerico,
CAST('2023-01-15' AS DATE) AS DataConvertida
FROM
MinhaTabela;
```

No exemplo acima, a coluna PrecoTexto é convertida para um tipo numérico que pode ser usado em cálculos. Da mesma forma, a string '2023-01-15' é convertida em um tipo de dado DATE, permitindo ordenação cronológica e operações de data.

3.2. Normalização de Strings

A padronização de strings é essencial para evitar duplicatas causadas por diferenças de caixa (maiúsculas/minúsculas) ou espaços extras. Uma cidade como "são paulo", "São Paulo"e "SÃO PAULO "deve ser tratada como um único valor.

As funções mais comuns para isso são:

- **LOWER()**: Converte a string para minúsculas.
- **UPPER()**: Converte a string para maiúsculas.
- **TRIM()**: Remove espaços em branco do início e do fim da string.

Exemplo Prático:

Para padronizar a coluna de Cidade de uma tabela de clientes:

```
SELECT
TRIM(UPPER(Cidade)) AS CidadePadronizada
FROM
Clientes;
```

Esse comando garante que todos os registros da cidade de São Paulo, por exemplo, serão tratados como 'SÃO PAULO', facilitando a contagem de registros e agrupamentos.

3.3. Conversão de Datas e Textos em PostgreSQL

Dados de data e hora, quando armazenados como texto, precisam ser convertidos para tipos nativos como `DATE` ou `TIMESTAMP` para que o PostgreSQL possa realizar cálculos e ordenações corretamente. O PostgreSQL é particularmente rigoroso com os formatos de data que aceita em conversões diretas.

O Caso Padrão: Conversão Direta com `CAST` Se o texto já estiver no formato padrão ISO 8601 (`'YYYY-MM-DD'`), a conversão é trivial. Pode-se usar a função `CAST` ou, de forma mais idiomática em PostgreSQL, o operador de atalho `::`.

```
-- Ambas as linhas funcionam perfeitamente para o formato padrão
SELECT CAST('2023-01-15' AS DATE);
SELECT '2023-01-15'::DATE;
```

No entanto, tentar usar essa conversão direta em um texto como `'15/01/2023'` resultará em um erro, pois o formato é ambíguo para o interpretador padrão.

O Caso Realista: Usando `TO_DATE` para Formatos Específicos Na prática, os dados de texto raramente estão em um formato ideal. Para lidar com formatos variados e não padronizados, como `'DD/MM/YYYY'` ou `'Janeiro 15, 2023'`, a função `TO_DATE` é a ferramenta correta e segura, pois permite que você especifique exatamente como a string deve ser lida.

Exemplo Prático: Vamos calcular há quantos dias um evento ocorreu, sabendo que sua data está em uma coluna de texto no formato `'DD/MM/YYYY'`.

```
-- Exemplo em PostgreSQL
SELECT
    DataEventoTexto,
    -- Etapa 1: Usar TO_DATE para interpretar o formato específico 'DD/MM/YYYY'
    TO_DATE(DataEventoTexto, 'DD/MM/YYYY') AS DataEventoFormatada,

    -- Etapa 2: Calcular a diferença em dias usando o operador de subtração
    (CURRENT_DATE - TO_DATE(DataEventoTexto, 'DD/MM/YYYY')) AS DiasDesdeOEvento
FROM
    Eventos;
```

Análise do Código de Exemplo O comando SQL acima utiliza as seguintes funções e operadores:

- **`TO_DATE(texto, formato)`:** Converte uma string para o tipo `DATE`. É essencial quando o texto não segue o padrão ISO. Ela recebe dois argumentos: o texto a ser convertido (`DataEventoTexto`) e um molde que descreve o formato de entrada (`'DD/MM/YYYY'`).
- **`CURRENT_DATE`:** Função padrão SQL que retorna a data atual do servidor, sem o componente de hora.

- **O Operador de Subtração (-):** No PostgreSQL, subtrair uma data de outra (`data_final - data_inicial`) retorna um número inteiro que representa a diferença exata em dias, simplificando o cálculo.

A correta conversão e padronização dos tipos de dados é o alicerce para uma análise de dados robusta, prevenindo erros e garantindo que filtros e cálculos cronológicos funcionem com precisão.

4. Binning e discretização

O processo de *binning* em SQL consiste em transformar variáveis contínuas em categorias (ou faixas). Isso pode ser feito manualmente (definindo intervalos específicos) ou automaticamente (utilizando funções ou quantis). Essa técnica é útil para simplificar análises, criar relatórios e preparar dados para modelos preditivos.

4.1 Binning Manual com CASE WHEN

Podemos criar faixas de valores explicitamente utilizando a cláusula `CASE WHEN`.

ID_Cliente	Idade
1	17
2	22
3	35
4	47
5	63

Exemplo em SQL:

```
SELECT
    customer_id,
    CASE
        WHEN age < 18 THEN 'Menor de idade'
        WHEN age BETWEEN 18 AND 29 THEN 'Jovem'
        WHEN age BETWEEN 30 AND 59 THEN 'Adulto'
        ELSE 'Idoso'
    END AS faixa_etaria
FROM clientes;
```

Resultado esperado:

ID_Cliente	Faixa_Etária
1	Menor de idade
2	Jovem
3	Adulto
4	Adulto
5	Idoso

4.2 Binning Automático com Funções

Também podemos usar funções como `FLOOR`, `ROUND` ou quantis via funções de janela.

ID_Produto	Preço
201	12.50
202	23.40
203	48.90
204	52.10
205	97.80

Exemplo em SQL (binning em intervalos de 20):

```
SELECT
    product_id,
    price,
    FLOOR(price / 20) * 20 AS faixa_preco
FROM produtos;
```

Resultado esperado:

ID_Produto	Preço	Faixa_Preço
201	12.50	0
202	23.40	20
203	48.90	40
204	52.10	40
205	97.80	80

4.3 Binning com Quantis (Funções de Janela)

Outra forma é dividir os dados em quantis usando `NTILE`.

```
SELECT
    customer_id,
    amount,
    NTILE(4) OVER (ORDER BY amount) AS quartil
FROM payment;
```

Esse exemplo divide os pagamentos em 4 grupos de valores (quartis). Assim, podemos rapidamente identificar quem está no grupo dos menores ou maiores gastos.

4.4 Aplicações

- Categorizar idades em faixas (ex.: jovem, adulto, idoso).
- Agrupar preços em intervalos de valores.
- Transformar métricas contínuas em classes para análise estatística.
- Preparar variáveis para modelos de classificação.

5. Modelagem e Reformatação de Dados

A modelagem e reformatação de dados são etapas importantes para organizar informações de forma que fiquem mais fáceis de analisar. Ela envolve a transformação da estrutura das tabelas ou a criação de tabelas derivadas para análises específicas.

5.1 Pivot (linhas → colunas)

O **pivot** transforma registros de linhas em colunas. Isso é útil quando queremos visualizar medidas agregadas de forma cruzada ou resumida.

Exemplo: temos uma tabela de vendas por mês:

Produto	Mês	Vendas
A	Jan	100
A	Fev	120
B	Jan	200
B	Fev	180

Para pivotar a tabela de modo que cada mês seja uma coluna:

```
SELECT
    Produto,
    SUM(CASE WHEN Mes = 'Jan' THEN Vendas ELSE 0 END) AS Jan,
    SUM(CASE WHEN Mes = 'Fev' THEN Vendas ELSE 0 END) AS Fev
FROM Vendas
GROUP BY Produto;
```

Resultado esperado:

Produto	Jan	Fev
A	100	120
B	200	180

5.2 Unpivot (colunas → linhas)

O **unpivot** é o processo inverso do pivot: transforma colunas em linhas. Isso é útil para análises que exigem um formato "long" em vez de "wide".

Exemplo: usando a tabela pivotada acima:

```
SELECT
    Produto,
    Mes,
    Vendas
FROM
    (SELECT Produto, Jan, Fev FROM VendasPivot) vp
UNPIVOT
    (Vendas FOR Mes IN (Jan, Fev)) AS unpvt;
```

Resultado esperado:

Produto	Mes	Vendas
A	Jan	100
A	Fev	120
B	Jan	200
B	Fev	180

5.3 Criação de Tabelas Derivadas (Subqueries e CTEs)

Tabelas derivadas são consultas que criam conjuntos temporários de dados que podem ser reutilizados em outras consultas. Existem duas abordagens principais:

- **Subqueries na cláusula FROM:** cria uma "tabela temporária" dentro de uma consulta.
- **CTEs (Common Table Expressions):** permitem nomear a tabela derivada e usá-la em múltiplas etapas da consulta, aumentando legibilidade.

Exemplo 1: Subquery

```
SELECT Produto, MediaVendas
FROM (
    SELECT Produto, AVG(Vendas) AS MediaVendas
    FROM Vendas
    GROUP BY Produto
) AS VendasMedia
WHERE MediaVendas > 150;
```

Exemplo 2: CTE

```
WITH VendasMedia AS (
    SELECT Produto, AVG(Vendas) AS MediaVendas
    FROM Vendas
    GROUP BY Produto
)
SELECT *
FROM VendasMedia
WHERE MediaVendas > 150;
```

Resultado esperado:

Produto	MediaVendas
B	190

5.4 Boas Práticas

- Sempre use `GROUP BY` com funções de agregação ao criar pivots.
- Prefira CTEs para consultas complexas, pois aumentam legibilidade e facilitam manutenção.
- Evite criar pivots muito largos (muitas colunas), prefira manter dados "long" se houver muitas categorias.
- Documente as transformações de dados, especialmente quando usar subqueries e CTEs, para rastreabilidade.