Aula 6

1. Fundamentos da Análise de Texto em SQL

A análise de texto em SQL é uma prática que combina o poder das consultas relacionais com a exploração de informações textuais. Embora o SQL seja amplamente reconhecido por suas capacidades de manipulação e agregação de dados estruturados, ele também oferece ferramentas úteis para examinar, transformar e extrair informações de campos textuais. Neste módulo introdutório, vamos entender em quais situações o SQL pode ser eficaz para lidar com dados de texto e quando é melhor recorrer a outras abordagens.

- **Por que análise de texto com SQL?** O SQL é uma das linguagens mais acessíveis e difundidas no ambiente de análise de dados. Sua grande vantagem é permitir que análises textuais simples sejam feitas diretamente no banco de dados, sem a necessidade de exportar os dados para outras ferramentas. É possível, por exemplo, identificar a frequência de palavras, buscar ocorrências específicas em descrições ou títulos e realizar filtros baseados em padrões textuais (como nomes, expressões ou prefixos). Além disso, o SQL se integra facilmente com grandes volumes de dados, tornando-o ideal para tarefas iniciais de exploração textual em bancos relacionais.

- **O que é análise de texto?** A análise de texto é o processo de examinar dados textuais com o objetivo de extrair significado, padrões ou informações úteis. No contexto do SQL, essa análise pode envolver:

  - contagem de ocorrências de palavras ou expressões;
  - busca de palavras-chave em campos textuais usando operadores como `LIKE`, `ILIKE` ou expressões regulares;
  - limpeza e padronização de textos com funções como `LOWER()`, `TRIM()` e `REPLACE()`;
  - extração de informações específicas, como domínios de e-mails, códigos, ou trechos de texto relevantes.

  Essas operações são muito úteis para criar uma base sólida antes de aplicar técnicas mais avançadas de análise textual.

- **Quando SQL não é a melhor opção?** Embora o SQL ofereça bons recursos para buscas e transformações básicas, ele não é ideal para tarefas mais complexas de Processamento de Linguagem Natural (PLN). Por exemplo, análises de sentimento, classificação de textos, resumo

automático, identificação de tópicos ou compreensão semântica exigem ferramentas mais especializadas — como bibliotecas em Python (`spaCy`, `NLTK`, `transformers`) ou plataformas de aprendizado de máquina. Assim, o SQL é mais apropriado para a etapa de exploração, limpeza e extração inicial dos dados textuais, servindo como um ponto de partida eficiente para análises mais avançadas.

2. Manipulação e Transformação de Texto

Neste módulo, aprenderemos a realizar operações fundamentais de manipulação e transformação de textos no SQL. Essas técnicas são essenciais para limpar, padronizar e extrair informações úteis de campos textuais antes de análises mais complexas.

2.1 Funções de Transformação de Texto

Em muitos bancos de dados, especialmente no **PostgreSQL**, existem diversas funções que permitem alterar o conteúdo de textos. Essas funções são amplamente utilizadas em processos de *data cleaning* e padronização.

- `UPPER(text)` – converte todo o texto para letras maiúsculas.
- `LOWER(text)` – converte todo o texto para letras minúsculas.
- `INITCAP(text)` – coloca a primeira letra de cada palavra em maiúscula.
- `REPLACE(text, from, to)` – substitui ocorrências de uma substring.
- `TRIM(text)` – remove espaços em branco no início e no fim da string.
- `LTRIM(text)`, `RTRIM(text)` – removem espaços apenas à esquerda ou à direita.

Exemplo: Suponha a tabela `clientes` com uma coluna de nome despadronizada:

nome
’ ana SILVA ’
’joão ’
’CARLOS souza’

Podemos padronizar os nomes com:

```
SELECT
    INITCAP(TRIM(nome)) AS nome_padronizado
FROM clientes;
```

Resultado:

nome_padronizado
Ana Silva
João
Carlos Souza

Essas transformações tornam os dados mais consistentes e facilitam buscas e comparações posteriores.

—

2.2 Características do Texto: Comprimento e Posição

Outra etapa importante da análise de texto é entender sua estrutura — quantos caracteres tem, onde certos elementos começam ou terminam, etc.

- `LENGTH(text)` – retorna o número de caracteres de uma string.
- `POSITION(substring IN text)` – retorna a posição da primeira ocorrência de uma substring.
- `LEFT(text, n)` / `RIGHT(text, n)` – retornam os primeiros ou últimos `n` caracteres.
- `SUBSTRING(text FROM start FOR length)` – extrai parte específica do texto.

Exemplo: Identificar padrões em descrições de produtos.

```
SELECT
    produto,
    LENGTH(descricao) AS tamanho_texto,
    POSITION('USB' IN descricao) AS pos_usb,
    LEFT(descricao, 10) AS inicio,
    RIGHT(descricao, 10) AS fim
FROM produtos;
```

Resultado ilustrativo:

produto	tamanho	pos_usb	inicio	fim
Mouse óptico USB	17	14	Mouse ópt	óptico USB
Teclado mecânico	17	NULL	Teclado m	mecânico

Essas informações ajudam a compreender a estrutura textual dos dados e identificar padrões relevantes.

—

2.3 Encontrando Elementos em Blocos de Texto

Com dados textuais mais limpos e estruturados, podemos localizar palavras ou expressões específicas. As funções de substring e busca são úteis para filtrar registros com base em conteúdo textual.

- `SUBSTRING(text, pattern)` – extrai parte de um texto com base em um padrão (regex simples).
- `STRPOS(text, substring)` – retorna a posição da substring, semelhante a `POSITION`.
- `SPLIT_PART(text, delimiter, index)` – divide o texto em partes com base em um delimitador e retorna a parte desejada.

Exemplo 1: Encontrar o domínio de e-mails.

```
SELECT
    email,
    SPLIT_PART(email, '@', 2) AS dominio
FROM usuarios;
```

Resultado:

email	dominio
ana@gmail.com	gmail.com
joao@empresa.com	empresa.com

Exemplo 2: Encontrar registros que contenham a palavra "error" em uma tabela de logs.

```
SELECT *
FROM logs
WHERE STRPOS(LOWER(mensagem), 'error') > 0;
```

Exemplo 3: Extrair o primeiro termo de uma frase.

```
SELECT
    SPLIT_PART(descricao, ' ', 1) AS primeira_palavra,
    descricao
FROM eventos;
```

Essas técnicas são a base da **análise lexical em SQL**, permitindo categorizar, filtrar e resumir informações textuais.

—

2.4 Boas Práticas

- Sempre normalize o texto (`LOWER()` ou `UPPER()`) antes de comparações para evitar problemas de *case-sensitive*.
- Use `TRIM()` para remover espaços invisíveis que afetam igualdade e busca.
- Prefira funções nativas do banco (como `SPLIT_PART` ou `POSITION`) a expressões complexas com `LIKE`, sempre que possível.
- Quando trabalhar com textos grandes, avalie índices e funções específicas de busca textual (`to_tsvector`, `to_tsquery`) para melhor desempenho.

—

2.5 Exercícios de Fixação

1. Padronize uma coluna de nomes aplicando `TRIM()`, `INITCAP()` e `REPLACE()`.
2. Extraia o domínio de e-mails de uma tabela de usuários e agrupe por domínio.
3. Conte quantos registros de uma tabela contêm a palavra “alert” em um campo de descrição.
4. Liste os três primeiros caracteres de cada valor em uma coluna textual.
5. Localize a posição da palavra “SQL” em uma frase e extraia o trecho completo após ela.

3. Técnicas de Busca e Casamento de Padrões

Neste capítulo, vamos explorar técnicas de localização de strings em blocos de texto. Esse tipo de operação pode ser interessante para filtrar resultados (registros com uma dada palavra ou expressão), categorizar dados com base na presença ou ausência de expressões e substituir determinadas strings por outros valores.

Em primeiro lugar, vamos falar da utilização dos comandos LIKE e ILIKE. De modo geral, ambos podem ser utilizados para procurar por uma string em um texto. Se queremos obter todas as ocorrências da palavra "man"dos registros do dataset, podemos fazer (lembre-se que o curinga "%"substitui uma sequência de caracteres):

```
SELECT COUNT(*)
FROM ufo
WHERE description LIKE/ILIKE '%man%';
```

A diferença entre os comandos está em como eles realizam os parseamentos entre a palavra requerida pelo comando e a palavra presente no texto. No caso do exemplo anterior, a utilização de LIKE resultaria na busca de expressões com a presença de palavras exatamente da forma "man"(case-sensitive). Por outro lado, ao utilizar ILIKE, todas as diferentes variações de "man"seriam procuradas, como "man", "Man", "MAN", etc (case-insensitive). O comando ILIKE pode ser construido a partir do LIKE, como abaixo:

```
SELECT COUNT(*)
FROM ufo
WHERE LOWER(description) LIKE '%man%';
```

Nas aulas introdutórias, apresentamos alguns curingas, como "_"(substitui um caractere qualquer) e "%"(substitui uma quantia arbitrária de caracteres). A seguir, temos um exemplo de utilização desses elementos, onde buscamos todos os dados que possuem as expressões "man"e "men".

```
SELECT *
FROM ufo
WHERE description ILIKE '%m_n%';
```

No caso de contagem do número de diferentes strings nos textos do dataset, podemos utilizar o camando case juntamente ao count. No código abaixo, contamos o número de registros com as palavras "south", "north", "west"e "east".

```
SELECT
    COUNT(CASE WHEN description ILIKE '%south%' THEN 1 END) AS South,
    COUNT(CASE WHEN description ILIKE '%north%' THEN 1 END) AS North,
    COUNT(CASE WHEN description ILIKE '%east%' THEN 1 END) AS East,
    COUNT(CASE WHEN description ILIKE '%west%' THEN 1 END) AS West
FROM ufo;
```

Além de LIKE e ILIKE, podemos também utilizar outros operadores, como IN e NOT IN. Esses comandos são úteis para manter o código mais compacto e com menor chance de erros decorrentes de códigos grandes e confusos. Suponha que, por exemplo, queremos observar todos os registros em que a descrição se iniciava por uma cor tradicional, como amarelo, branco, azul, etc. Podemos fazer isso utilizando apenas uma sequência de WHERE. Observe que o comando "split_part(texto, delimitador, índice)"divide um texto com base no caractere de delimitação e obtém a partição de índice fornecido.

```
SELECT first_word, description
FROM
    (
        SELECT split_part(description,' ', 1) as first_word
        ,description
        FROM ufo
    ) a
WHERE first_word = 'Red'
OR first_word = 'Orange'
OR first_word = 'Yellow'
OR first_word = 'Green'
OR first_word = 'Blue'
OR first_word = 'Purple'
OR first_word = 'White';
```

Podemos, contudo, agrupar todos os diferentes valores de comparação do WHERE em apenas uma linha, como abaixo.

```
SELECT first_word, description
FROM
    (
        SELECT split_part(description,' ', 1) as first_word
        ,description
        FROM ufo
    ) a
WHERE first_word IN ('Red','Orange','Yellow','Green','Blue','Purple','White')
```

Por fim, segue um exemplo de categorização da primeira palavra de cada registro com base em questões de cor, moviemento e formato. Observe que o código é extremamente legível e compacto, o que não aconteceria com o uso de diversos case statements.

```
SELECT
    CASE WHEN LOWER(first_word) IN ('red','orange','yellow','green',
    'blue','purple','white') THEN 'Color'
    WHEN LOWER(first_word) IN ('round','circular','oval','cigar')
    THEN 'Shape'
    WHEN first_word ILIKE 'triang%' THEN 'Shape'
    WHEN first_word ILIKE 'flash%' THEN 'Motion'
    WHEN first_word ILIKE 'hover%' THEN 'Motion'
    WHEN first_word ILIKE 'pulsat%' THEN 'Motion'
    ELSE 'Other'
    END AS first_word_type,
    count(*)
FROM
    (
        SELECT split_part(description,' ',1) as first_word
        ,description
        FROM ufo
    ) a
GROUP BY 1
ORDER BY 2 desc;
```

4. Expressões Regulares (RegEx) para Buscas Complexas

As expressões regulares, ou **RegEx** (Regular Expressions), são padrões usados para encontrar, validar e manipular textos de maneira flexível. Em SQL, elas nos permitem realizar buscas muito mais precisas do que o operador tradicional `LIKE`.

Por que usar RegEx em SQL?

Quando usamos `LIKE`, conseguimos identificar padrões simples, como:

```
SELECT nome FROM clientes WHERE nome LIKE 'Jo%';
```

Esse comando retorna todos os nomes que **começam com "Jo"**, como "João" ou "Josefa". Mas e se quisermos:

- Encontrar palavras que **terminam com "son"**, como "Anderson" ou "Robson";
- Buscar nomes que **tenham um número no meio**;
- Identificar e-mails válidos, CEPs, telefones ou códigos específicos?

O `LIKE` não é suficiente para isso. É aí que entram as expressões regulares.

Operadores RegEx no SQL (PostgreSQL)

O PostgreSQL oferece operadores específicos para trabalhar com expressões regulares:

- `~` → corresponde a um padrão (case-sensitive);
- `~*` → corresponde a um padrão (sem diferenciar maiúsculas e minúsculas);
- `!~` → nega o padrão (não corresponde);
- `!~*` → nega o padrão (sem diferenciar maiúsculas e minúsculas).

Por exemplo:

```
SELECT nome FROM clientes WHERE nome ~ '^Jo';
```

Esse comando retorna todos os nomes que **começam com "Jo"**.

Construindo Padrões Simples

Antes de vermos exemplos complexos, vamos entender os principais símbolos das expressões regulares:

- `.` → representa **qualquer caractere**;
- `*` → zero ou mais ocorrências do elemento anterior;
- `+` → uma ou mais ocorrências;
- `?` → zero ou uma ocorrência;
- `^` → início da string;
- `$` → fim da string;

- `[ ]` → conjunto de caracteres permitidos;
- `|` → operador "ou";
- `()` → agrupa partes do padrão.

Exemplos Práticos e Progressivos

1. Palavras que começam com uma letra específica

```
SELECT nome FROM clientes WHERE nome ~ '^A';
```

Seleciona nomes que **começam com A**, como "Ana" e "Amanda".

2. Palavras que terminam com determinada sequência

```
SELECT nome FROM clientes WHERE nome ~ 'son$';
```

Seleciona nomes que **terminam com "son"**, como "Robson" e "Anderson".

3. Palavras que contêm um número

```
SELECT usuario FROM logins WHERE usuario ~ '[0-9]';
```

Retorna todos os usuários que possuem **pelo menos um número** no nome, como "juan123" ou "ana2".

4. E-mails válidos (formato básico)

```
SELECT email
FROM usuarios
WHERE email ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$';
```

Verifica se os e-mails seguem o formato padrão: `usuario@dominio.com`.

5. Telefone no formato (XX)XXXXX-XXXX

```
SELECT telefone
FROM contatos
WHERE telefone ~ '^\([0-9]{2}\)[0-9]{5}-[0-9]{4}$';
```

Seleciona telefones como `(11)98765-4321`.

6. Palavras que contenham "data" ou "info"

```
SELECT descricao
FROM produtos
WHERE descricao ~ 'data|info';
```

Retorna textos que tenham as palavras "data" ou "info".

7. Textos com letras maiúsculas e minúsculas misturadas

```
SELECT texto
FROM mensagens
WHERE texto ~* 'sql';
```

A busca é feita sem considerar maiúsculas ou minúsculas — encontra “SQL”, “Sql” ou “sql”.

Comparando RegEx e LIKE

- **LIKE** → busca simples, adequada para padrões fixos (ex: “Jo
- **RegEx** → busca complexa, ideal para padrões variáveis e validações.

Por exemplo:

```
-- LIKE
SELECT nome FROM clientes WHERE nome LIKE 'Jo%';


-- RegEx
SELECT nome FROM clientes WHERE nome ~ '^Jo';
```

Ambos encontram nomes que começam com “Jo”, mas o RegEx permite evoluir o padrão facilmente — por exemplo, buscar “Jo” seguido de **qualquer vogal**:

```
SELECT nome FROM clientes WHERE nome ~ '^Jo[aeiou]';
```

Resumo Intuitivo

As expressões regulares funcionam como uma “linguagem de padrões” dentro do SQL. Com elas, é possível:

- Realizar buscas mais inteligentes em textos;
- Validar formatos (como e-mails e telefones);
- Extrair informações específicas de colunas textuais;
- Criar filtros que se adaptam a dados não padronizados.

Uma boa forma de aprender é começar testando pequenos padrões, entendendo como cada símbolo afeta o resultado. Com prática, RegEx se torna uma ferramenta poderosa para quem trabalha com SQL e análise de dados textuais.