EBOOK DE FÁCIL APRENDIZADO

DOMINANDO SQL – DO BÁSICO AO AVANÇADO

Coffee and Tips

Tech Tutorials

Como surgiu a linguagem SQL

A linguagem SQL (Structured Query Language) surgiu da necessidade de uma maneira mais simples e eficiente de interagir com grandes volumes de dados em bancos de dados relacionais. A história do SQL começa na década de 1970, quando o aumento do uso de computadores gerou a necessidade de gerenciar informações de forma estruturada.

Tudo começou no Centro de Pesquisa da IBM, onde os cientistas de computação Edgar F. Codd e sua equipe estavam desenvolvendo o conceito de bancos de dados relacionais. Em 1970, Codd publicou um artigo revolucionário chamado "A Relational Model of Data for Large Shared Data Banks", que descrevia o modelo relacional de bancos de dados. Sua proposta era organizar os dados em tabelas (ou "relações"), permitindo que diferentes conjuntos de dados fossem associados por meio de chaves e sem a necessidade de duplicar informações.

Com o modelo relacional definido, os pesquisadores da IBM precisavam de uma maneira prática de consultar e manipular esses dados. Foi aí que **Donald D. Chamberlin** e **Raymond F. Boyce** desenvolveram uma nova linguagem, chamada SEQUEL (Structured English Query Language), que permitia que os usuários fizessem perguntas ("queries") ao banco de dados de forma semelhante à linguagem humana.

O nome foi posteriormente alterado para SQL, e, em 1979, a empresa Relational Software Inc. (que mais tarde se tornaria a Oracle Corporation) lançou a primeira versão comercial do SQL em seu sistema de gerenciamento de banco de dados relacional.

A linguagem SQL rapidamente se popularizou, sendo adotada por diversos sistemas de bancos de dados e, em 1986, foi padronizada pela American National Standards Institute (ANSI), tornando-se uma linguagem universal para gerenciamento de dados relacionais.

Desde então, o SQL evoluiu e passou a incluir novos recursos, como suporte a funções de agregação, manipulação de grandes volumes de dados, e integração com outras tecnologias. Hoje, SQL é amplamente utilizado em bancos de dados como MySQL, PostgreSQL, Microsoft SQL Server, e muitos outros, sendo uma das linguagens mais importantes no campo da análise de dados e desenvolvimento de software.

Essa jornada de mais de 50 anos desde seu nascimento mostra como o SQL é essencial para lidar com a explosão de dados que vivemos hoje, e por que é uma habilidade fundamental para profissionais de tecnologia e dados.

Introdução ao SQL

Antes de nos aprofundarmos em SQL avançado, é importante recapitular os conceitos básicos que formam a base para consultas mais complexas. Como vimos anteriormente, SQL é uma linguagem utilizada para gerenciar dados em bancos de dados relacionais. Os principais comandos como SELECT, INSERT, UPDATE, e DELETE formam a base de qualquer operação em SQL.

Neste eBook, vamos nos concentrar em tópicos **avançados** que expandem essas operações básicas e tornam as consultas mais poderosas e eficientes. Se você já domina o básico, está pronto para avançar.

Uso de Joins no SQL

Um dos conceitos mais poderosos do SQL é a capacidade de combinar dados de várias tabelas usando joins. Existem diferentes tipos de joins que permitem reunir informações dispersas em várias tabelas, proporcionando uma visão mais completa dos dados.

INNER JOIN

O INNER JOIN retorna apenas os registros que têm correspondências em ambas as tabelas. Suponha que você tenha uma tabela *pedidos* além da tabela *usuarios*:

Tabela: pedidos

id_pedido	id_usuario	valor
1	1	150.00
2	3	200.00
3	1	99.99

Tabela: usuarios

id	nome	idade	cidade
1	Ana	25	São Paulo
2	João	30	Rio de Janeiro
3	Maria	22	Belo Horizonte
4	Carlos	35	Brasília
5	Bianca	28	Curitiba

Agora, queremos combinar as informações de *usuarios* com seus respectivos pedidos:

```
SELECT usuarios.nome, pedidos.valor
FROM usuarios
INNER JOIN pedidos
ON usuarios.id = pedidos.id_usuario;
```

Resultado:

nome	valor
Ana	150.00
Ana	99.99
Maria	200.00

LEFT JOIN

O LEFT JOIN retorna todos os registros da tabela à esquerda (neste caso, *usuarios*), mesmo que não haja correspondência na tabela da direita (*pedidos*):

```
SELECT usuarios.nome, pedidos.valor
FROM usuarios
LEFT JOIN pedidos ON usuarios.id =
pedidos.id_usuario;
```

Resultado:

nome	valor
Ana	150.00
Ana	99.99
João	NULL
Maria	200.00
Carlos	NULL
Bianca	NULL

Este exemplo mostra todos os usuários, mesmo os que não têm pedidos.

RIGHT JOIN

O RIGHT JOIN funciona de maneira oposta ao LEFT JOIN: ele retorna todos os registros da tabela à direita (neste caso, *pedidos*), mesmo que não haja correspondência na tabela da esquerda (*usuarios*).

```
SELECT usuarios.nome, pedidos.valor F
ROM usuarios
RIGHT JOIN pedidos
ON usuarios.id = pedidos.id_usuario;
```

Resultado:

nome	valor
Ana	150.00
Ana	99.99
Maria	200.00

Neste caso, ele exibe todos os registros da tabela *pedidos*, independentemente se houver correspondência na tabela *usuarios* através dos campos chaves *usuarios.id* e *pedidos.id_usuario.*

FULL OUTER JOIN

O FULL OUTER JOIN combina o comportamento de ambos os LEFT JOIN e RIGHT JOIN. Ele retorna todos os registros de ambas as tabelas, exibindo NULL onde não houver correspondência. Essa técnica é útil quando você deseja ver todos os dados de ambas as tabelas, incluindo os que não têm correspondência.

Exemplo: Queremos listar todos os usuários e seus pedidos, mas também mostrar usuários que não fizeram pedidos e pedidos sem usuários (caso algum registro seja excluído ou adicionado erroneamente):

```
SELECT usuarios.nome, pedidos.valor
FROM usuarios
FULL OUTER JOIN pedidos
ON usuarios.id = pedidos.id_usuario;
```

Resultado:

nome	valor
Ana	150.00
Ana	99.99
João	NULL
Maria	200.00
Carlos	NULL
Bianca	NULL

Aqui, o FULL OUTER JOIN retorna todos os usuários e todos os pedidos. Os usuários que não têm pedidos aparecem com NULL na coluna valor, e, se houvesse pedidos sem usuários, o nome apareceria como NULL.

Agora que você conhece os principais tipos de joins em SQL, está pronto para combinar dados de diferentes tabelas de maneira eficiente e extrair informações valiosas. Vamos avançar para outro tópico poderoso: subconsultas e consultas aninhadas!

Subconsultas e Consultas Aninhadas

As subconsultas permitem que você realize consultas dentro de outras consultas, tornando-as muito poderosas para consultas complexas. Elas são úteis quando você precisa fazer uma segunda consulta com base nos resultados da primeira.

Exemplo:

Encontrando o maior pedido de cada usuário
Para encontrar o maior valor de pedido feito por cada usuário, podemos usar uma subconsulta:

```
SELECT nome,
(SELECT MAX(valor) FROM pedidos WHERE
pedidos.id_usuario = usuarios.id)
AS maior_pedido
FROM usuarios;
```

Resultado:

nome	maior_pedido
Ana	150.00
João	NULL
Maria	200.00
Carlos	NULL
Bianca	NULL

Otimização de Consultas SQL

À medida que os bancos de dados crescem, otimizar consultas se torna fundamental para garantir um desempenho rápido e eficiente. Aqui estão algumas técnicas comuns:

Índices

Criar índices nas colunas que são frequentemente usadas em WHERE e JOIN pode acelerar significativamente as consultas.

```
CREATE INDEX idx_usuario_id ON usuarios(id);
```

Uso do EXPLAIN

O comando EXPLAIN permite visualizar como uma consulta será executada pelo banco de dados, mostrando detalhes sobre o plano de execução.

```
EXPLAIN SELECT * FROM pedidos
WHERE valor > 100;
```

Isso ajuda a identificar possíveis gargalos, como falta de índices ou tabelas escaneadas inteiras.

Funções Agregadas e Funções de Janela

Funções Agregadas

As funções agregadas realizam cálculos em um conjunto de valores e retornam um valor único. Exemplos incluem COUNT(), SUM(), AVG(), entre outros.

Exemplo: Contagem de usuários por cidade

```
SELECT cidade, COUNT(*) AS total_usuarios
FROM usuarios
GROUP BY cidade;
```

Resultado:

cidade	total_usuarios
São Paulo	1
Rio de Janeiro	1
Belo Horizonte	1
Brasília	1
Curitiba	1

Funções de Janela

As funções de janela são uma das ferramentas mais poderosas do SQL para executar cálculos sobre um conjunto de linhas que estão relacionadas à linha atual, sem precisar agrupar os resultados como fazemos com o GROUP BY. Essas funções permitem que você mantenha os detalhes das linhas enquanto realiza cálculos em um conjunto de dados.

RANK e PARTITION BY

A função RANK() é uma das funções de janela mais comuns e é usada para atribuir uma classificação (ou "rank") a cada linha de um conjunto de resultados com base em uma determinada ordem. Junto com RANK(), a cláusula PARTITION BY permite que você divida os dados em grupos (partições) para que a classificação seja reiniciada dentro de cada grupo. Isso é extremamente útil quando você precisa ordenar ou ranquear dados dentro de subconjuntos específicos de um conjunto maior.

Como o RANK() Funciona

O RANK() atribui uma posição a cada linha com base na ordenação que você especificar. No caso de valores iguais, ele atribui a mesma classificação, mas pula posições. Vamos ver um exemplo para entender melhor.

Exemplo: Imagine que temos a seguinte tabela de pedidos (pedidos):

id_pedido	id_usuario	valor
1	1	150.00
2	3	200.00
3	1	99.99
4	2	250.00
5	3	175.00

Agora, queremos classificar esses pedidos de acordo com o valor, com base em cada usuário. Vamos usar o RANK() para ver a posição de cada pedido dentro do grupo de cada usuário.

```
SELECT id_usuario, valor,
RANK() OVER
(PARTITION BY id_usuario ORDER BY valor DESC)
AS rank_pedido FROM pedidos;
```

Resultado:

id_usuario	valor	rank_pedido
1	150.00	1
1	99.99	2
2	250.00	1
3	200.00	1
3	175.00	2

Neste exemplo:

- Para o usuário 1 (id_usuario = 1), o pedido de valor 150.00 é classificado em primeiro lugar e o de 99.99 em segundo.
- Para o usuário 3, o pedido de 200.00 é o primeiro, e o de 175.00 é o segundo.
- O usuário 2 tem apenas um pedido, então ele ocupa o primeiro lugar automaticamente.

O PARTITION BY divide os dados com base na coluna id_usuario, de modo que a classificação (RANK()) seja reiniciada para cada grupo de usuários. O ORDER BY dentro da função define que estamos classificando com base no valor do pedido, em ordem decrescente.

Diferença Entre RANK(), DENSE_RANK() e ROW_NUMBER()

Além de RANK(), o SQL oferece outras funções de janela similares que também podem ser úteis dependendo do seu caso de uso:

- **RANK()**: Atribui o mesmo rank a linhas com valores iguais, mas pula posições quando houver empates. Por exemplo, se dois valores estão empatados na primeira posição, o próximo valor será classificado como 3º lugar.
- **DENSE_RANK()**: Semelhante ao RANK(), mas não pula posições. Se dois valores estão empatados em 1º, o próximo valor será o 2º.
- **ROW_NUMBER()**: Atribui um número único e sequencial a cada linha, independentemente de valores repetidos.

Exemplo Comparativo:

Para ver a diferença entre essas funções, podemos usá-las no mesmo conjunto de dados:

```
SELECT id_usuario, valor,
RANK() OVER (ORDER BY valor DESC) AS rank_valor,
DENSE_RANK() OVER (ORDER BY valor DESC) AS dense_rank_valor,
ROW_NUMBER() OVER (ORDER BY valor DESC) AS row_num_valor
FROM pedidos;
```

Resultado:

id_usuario	valor	rank_valor	dense_rank_valor	row_num_valor
2	250.00	1	1	1
3	200.00	2	2	2
3	175.00	3	3	3
1	150.00	4	4	4
1	99.99	5	5	5

- RANK(): Atribui classificações baseadas na ordenação, mas pula classificações quando há empates (se houver).
- DENSE_RANK(): Não pula classificações, mesmo com valores iguais.
- ROW_NUMBER(): Dá um número sequencial a cada linha, sem considerar valores repetidos.

Conclusão

Neste eBook, cobrimos desde o básico até os conceitos avançados de SQL. Agora, você está pronto para utilizar joins, subconsultas e otimizações para construir consultas eficientes e poderosas. Lembre-se de que SQL é uma linguagem prática — quanto mais você pratica, mais domínio você terá.

Se você tiver dúvidas ou quiser aprender mais, continue praticando, consultando documentação e experimentando com diferentes bancos de dados. SQL é uma habilidade valiosa que irá acelerar sua carreira no mundo dos dados e da tecnologia.

CURTIU? ACESSE MAIS TUTORIAIS COMO ESSE EM COFFEEANDTIPS.COM

E NOS SIGA NO INSTAGRAM

Coffee and Tips

Tech Tutorials