Preditiva.ai

Bancos de Dados Relacionais

Parte IV – Performance e outras operações avançadas no SQL

Particionamento

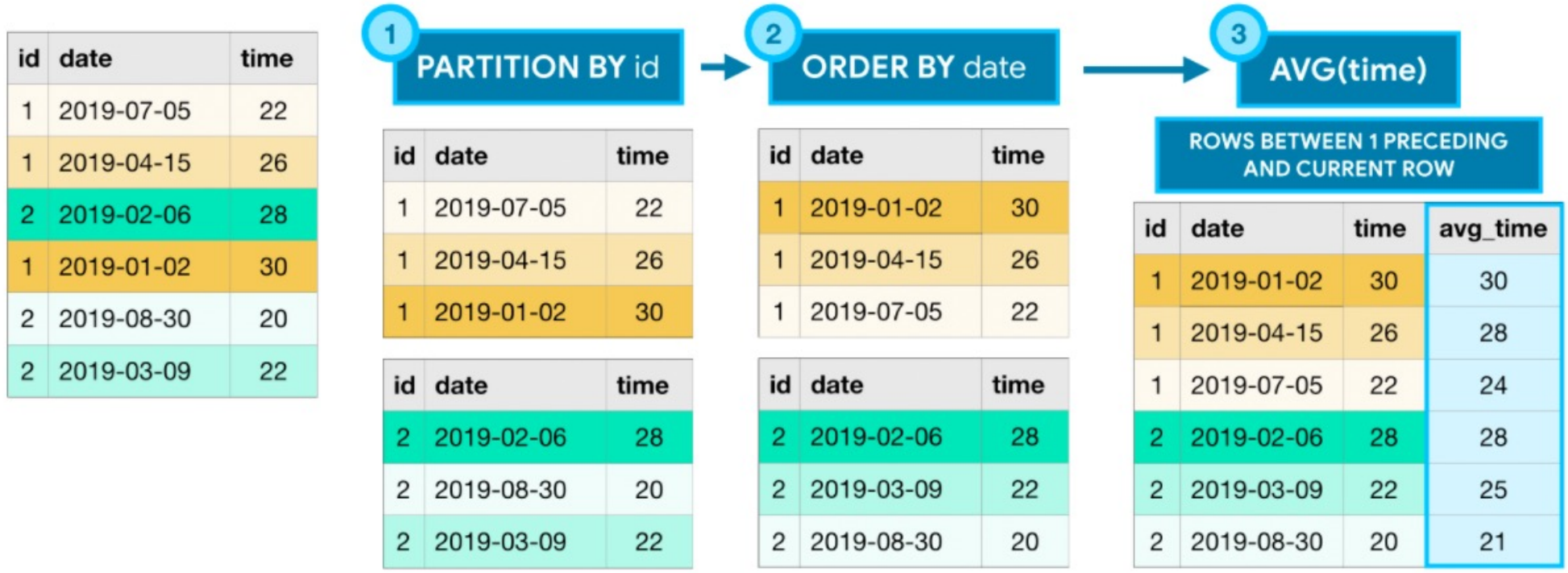
Manipulação de Dados

Particionamento

Preditiva.ai

Um tipo de operação muito comum para análise de dados é o uso de particionamento. Esse tipo de operação é realizada com a função **OVER().** Vejamos o conceito geral:

```
SELECT
    id,date,time,
    AVG(time) OVER(PARTITION BY id ORDER BY date ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS avg_time
FROM
    table
```



id	date	time
1	2019-07-05	22
1	2019-04-15	26
2	2019-02-06	28
1	2019-01-02	30
2	2019-08-30	20
2	2019-03-09	22

id	date	time
1	2019-07-05	22
1	2019-04-15	26
1	2019-01-02	30

id	date	time
2	2019-02-06	28
2	2019-08-30	20
2	2019-03-09	22

id	date	time
1	2019-01-02	30
1	2019-04-15	26
1	2019-07-05	22

id	date	time
2	2019-02-06	28
2	2019-03-09	22
2	2019-08-30	20

id	date	time	avg_time
1	2019-01-02	30	30
1	2019-04-15	26	28
1	2019-07-05	22	24
2	2019-02-06	28	28
2	2019-03-09	22	25
2	2019-08-30	20	21

Manipulação de Dados

Particionamento

Preditiva.ai

Com o particionamento, podemos fazer uma infinidade de operações, entre elas, aplicando funções de agregação, navegação e funções de numeração. Veja um resumo:

1) Funções de agregação

Como você deve se lembrar, AVG() (do exemplo acima) é uma função agregada. A cláusula OVER é o que garante que ela seja tratada como uma função analítica (agregada). As funções agregadas recebem todos os valores da janela como entrada e retornam um único valor.

- **MIN() (ou MAX())** - retorna o mínimo (ou máximo) dos valores do particionamento
- **AVG() (ou SUM())** - retorna a média (ou soma) dos valores do particionamento
- **COUNT()** - Retorna o número de linhas do particionamento

2) Funções de Navegação entre linhas

As funções de navegação atribuem um valor com base no valor em uma linha (geralmente) diferente da linha atual.

- **FIRST_VALUE() (ou LAST_VALUE())** - retorna o primeiro (ou último) valor do particionamento
- **LEAD() (e LAG())** - retorna o valor em uma linha subsequente (ou anterior)

3) Funções de numeração

As funções de numeração atribuem valores inteiros a cada linha com base na ordenação.

- **ROW_NUMBER()** - Retorna a ordem em que as linhas aparecem do particionamento (começando com 1)
- **DENSE_RANK()** - Todas as linhas com o mesmo valor na coluna de ordenação recebem o mesmo valor de classificação, onde a próxima linha recebe um valor de classificação que é incrementado.

Demonstração de consultas SQL com particionamento

Arquivo: Particionamento.sql

Manipulação de Dados

Particionamento

Preditiva.ai

Hands on

Tendo acesso ao banco de dados da empresa **Parch and Posey**, crie uma query que produza o relatório abaixo:

ano_venda	mes_venda	Soma_Venda_USD	Acum_Soma_Venda_USD
2.013	12	377.331	377.331
2.014	1	286.141	663.472
2.014	2	349.722	1.013.194
2.014	3	341.513	1.354.707
2.014	4	344.894	1.699.601
2.014	5	319.211	2.018.812
2.014	6	300.359	2.319.171
2.014	7	294.583	2.613.754
2.014	8	358.529	2.972.283
2.014	9	299.969	3.272.252

Dicas:

1. Utilize a tabela "**db_parchnposey.orders**" e os campos "**occurred_at**" (data de venda) e **total_amt_usd** (Vendas em USD).

2. Utilize funções de data para extrar o ano e mês. Dica: DATEPART();

3. Utilize Particionamento. Dica: Não é necessário aplicar o parâmetro PARTITION BY.

Performance

Performance no SQL

Complexidade de algum algoritmo

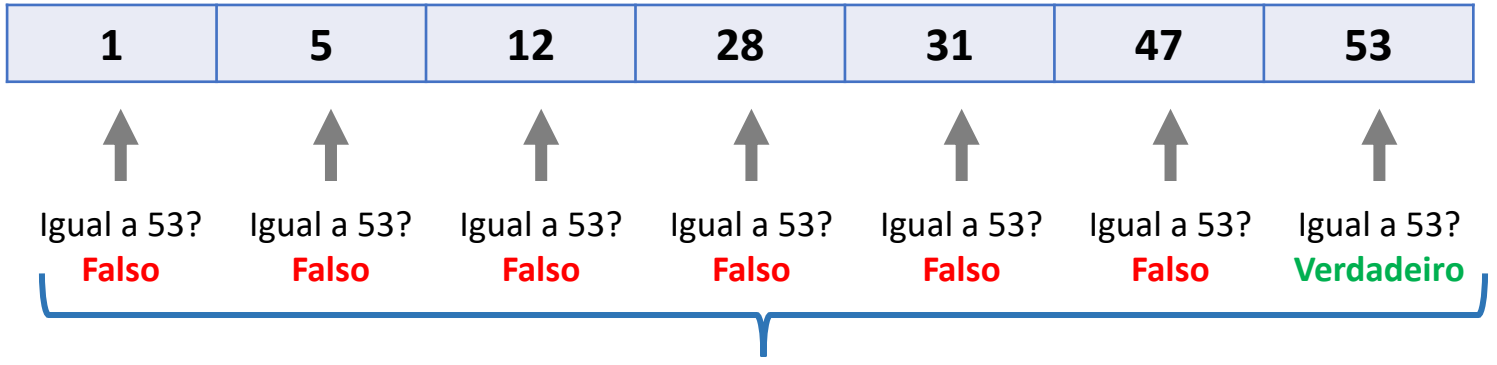
Preditiva.ai

Para falar de performance em queries precisamos primeiramente entender como um algoritmo computacional trabalha. Vejamos um exemplo de um algoritmo de busca numérica:

Queremos saber se o número 53 está na lista abaixo, qual algoritmo resolveria esse problema?

n = 7

Um algoritmo possível é comparar cada número da lista com 53, e verificar se há algum caso verdadeiro:



7 iterações

Performance no SQL

Complexidade de algum algoritmo

Preditiva.ai

O processo anterior é chamado de **busca linear ou sequencial**. Um algoritmo alternativo é a **busca binária**.

1º) Localizar o elemento central:

1	5	12	28	31	47	53

2º) Comparar o elemento central com o número que queremos localizar (53):

- *Se o número que procuramos for igual ao central, encontramos!*
- *Se for menor, procuramos apenas na metade menor;*
- *Se for maior, procurmos apenas na metade maior.*

1	5	12	28	31	47	53

1ª iteração

53 > 28

Então procuraremos na metade maior

Performance no SQL

Complexidade de algum algoritmo

Preditiva.ai

3º) Repetimos o passo de comparar o elemento central com o número 53 até localizarmos o número ou acabar as opções:

Repare que, para uma lista de tamanho n=7, o processo de busca linear teve 7 iterações, enquanto o de busca binária teve 3 iterações. Mas isso depende do número que procuramos e onde ele está localizado. Como podemos analisar cada algoritmo?

Performance no SQL

Complexidade de algum algoritmo

Preditiva.ai

A eficiência de um algoritmo é medido pela quantidade de operações feitas pelo algoritmo. Essa função, que depende de n (tamanho de entrada), é chamada de **Complexidade**.

No caso dos algoritmos de busca, podemos mostrar que a complexidade da busca linear é de ordem n e a da busca binária é $log_2 n$.

Vamos comparar o que acontece a medida que aumentamos o valor de n:

n	Pior Caso		Variação %
	Linear	Binária	
10	10	3	-66,8%
20	20	4	-78,4%
30	30	5	-83,6%
40	40	5	-86,7%
50	50	6	-88,7%
100	100	7	-93,4%
200	200	8	-96,2%
300	300	8	-97,3%
400	400	9	-97,8%
500	500	9	-98,2%
1.000	1.000	10	-99,0%
10.000	10.000	13	-99,9%
100.000	100.000	17	-100,0%
1.000.000	1.000.000	20	-100,0%
10.000.000	10.000.000	23	-100,0%

Essa forma de comparar a eficiência de algoritmos é chamada de **Notação Big O.** Ao desenvolver algoritmos, programadores deveriam buscar aqueles com a menor complexidade possível, para que o computador execute menos operações.

Performance no SQL

Complexidade de algum algoritmo

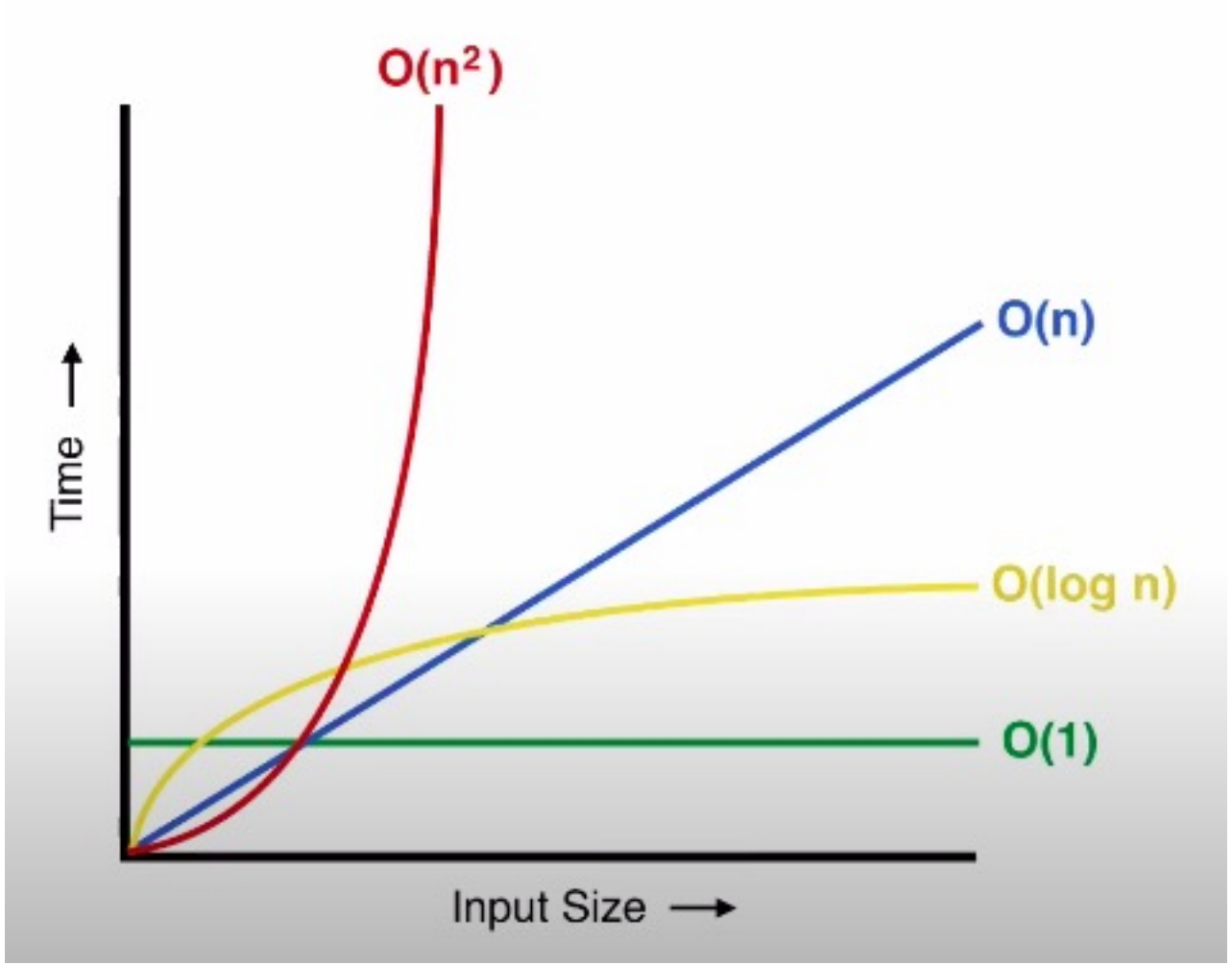
Preditiva.ai

A eficiência de um algoritmo é medido pela quantidade de operações feitas pelo algoritmo. Essa função, que depende de n (tamanho de entrada), é chamada de **Complexidade**.

No caso dos algoritmos de busca, podemos mostrar que a complexidade da busca linear é de ordem n e a da busca binária é $log_2 n$.

Vamos comparar o que acontece a medida que aumentamos o valor de n:

Essa forma de comparar a eficiência de algoritmos é chamada de **Notação Big O.** Ao desenvolver algoritmos, programadores deveriam buscar aqueles com a menor complexidade possível, para que o computador execute menos operações.



Performance no SQL

Ordem de execução

Preditiva.ai

Na mesma linha de eficiência, também podemos buscar **melhores práticas no código SQL**. Otimizar o servidor de banco de dados é muito importante, mas melhorar o **desempenho de consultas individuais pode ser ainda mais**. Existem várias formas de otimizar o banco e as consultas. Separamos aqui os principais:

1) Entenda a ordem de execução do SQL. Veja:

i. FROM (e JOIN) referencia as tabelas a serem usadas;
ii. WHERE filtra os dados;
iii. GROUP BY agrega os dados;
iv. HAVING filtra a agregação realizada;
v. SELECT seleciona as colunas (e depois remove o duplicados se DISTINCT for usado);
vi. ORDER BY ordena os resultados.

Como regra geral das queries, quanto menos dados forem passando entre as etapas, melhor.

Ou seja, use aquilo que realmente precisa.

Performance no SQL

Ordem de execução

Preditiva.ai

2) Só utilize o **ORDER BY** e **DISTINCT** quando realmente necessário. Esses comandos ocupam muitos recursos.

3) Não coloque muitos JOINS simultâneos na mesma Query. Use as **WITHs** sempre que possível como etapas anteriores. Além disso, busque filtrar as tabelas com WHERE antes dos JOINs, assim a busca entre as tabelas é reduzida.

4) Utilizar **índices** nas tabelas pode ser uma boa alternativa. O time de banco de dados / engenharia de dados pode ajudar caso não consiga criar os índices por falta de permissão de escrita no banco de dados.

Performance no SQL

Índices em tabelas

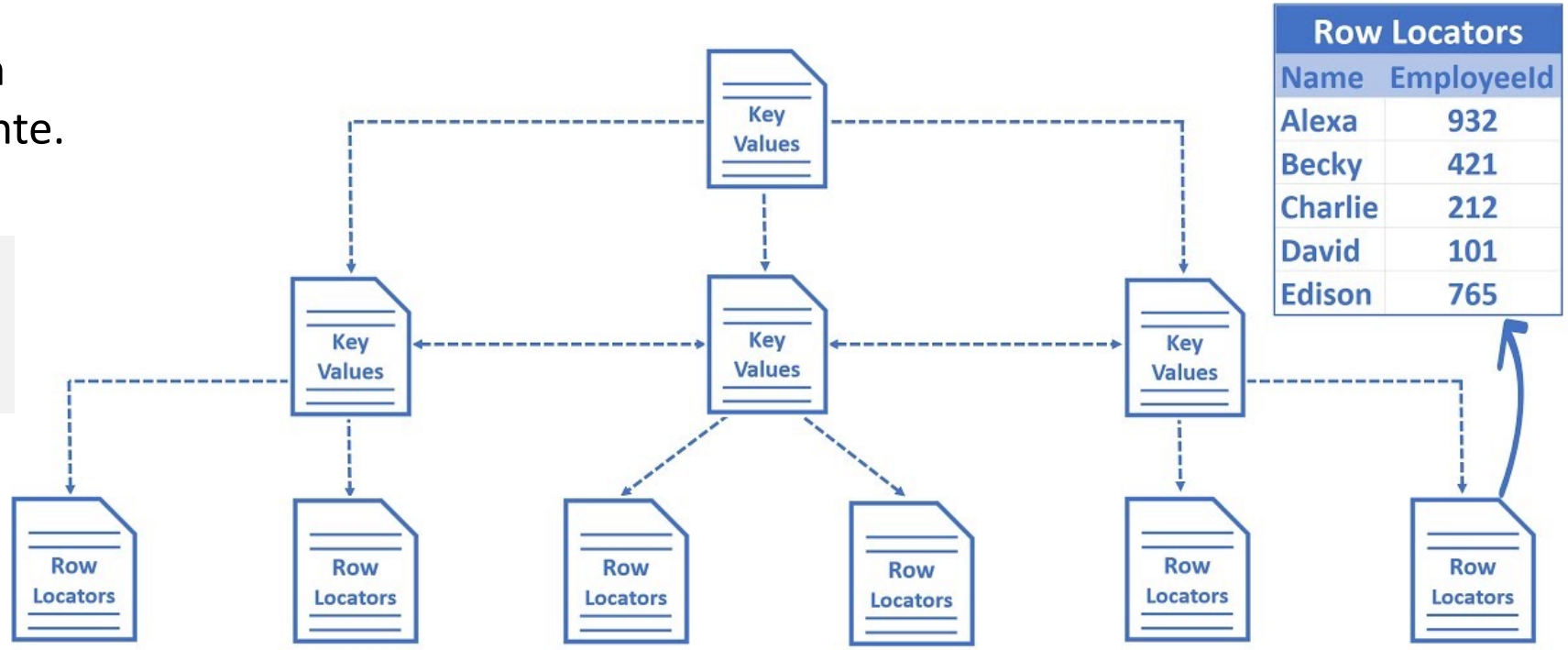
Preditiva.ai

Os **Índices** nos bancos de dados são utilizados para facilitar a busca de informações em uma tabela com o menor número possível de operações de leituras, tornado assim a busca mais rápida e eficiente.

Veja como funciona:

Obs: Colunas do tipo PK já têm índices criados automaticamente.

Esse tipo de índice mostrado ao lado é chamado de **NON CLUSTERED INDEX**.



Performance no SQL

Índices em tabelas

Preditiva.ai

Os índices são bons muitas vezes, mas **não** salvam a performance em todos os momentos. Veja algumas dicas e considerações ao optar pelo seu uso:

1) É indicado colocar índices apenas nas colunas mais usadas quem passam por filtros.

2) Um dos problemas dos índices é que eles também criam novos dados a serem armazenados, fazendo com que mais dados também tenham que ser lidos e a performance pode ficar ainda pior.

3) Quando uma tabela passa por muitos **updates** ou **deletes**, veja se vale a pena criar índices. Os índices funcionam melhor em tabelas que são muito utilizadas mas que não são muito atualizadas.

4) Não crie índices como primeira forma de otimização da Query. Primeiro foque nas outras dicas e, se não adiantar, teste os índices e veja se há melhora.

Demonstração de Índices

Preditiva.ai