

## *Construção de Comandos SQL com boa performance*

Em **bancos de dados relacionais** as informações são guardadas em **tabelas**. Para recuperar uma informação necessária ao usuário, deve-se buscá-la em várias tabelas diferentes, estabelecendo-se um **relacionamento** entre elas. Esta é a origem do nome deste paradigma de banco de dados.

Tabelas são na verdade conjuntos. Por exemplo, quando em um sistema existe uma tabela de vendas, esta tabela corresponde ao conjunto de todas as vendas feitas por uma empresa. A tabela de vendedores corresponde ao conjunto de vendedores que trabalham em uma empresa. **Cada linha ou registro da tabela corresponde a um elemento do conjunto.** Consultas e alterações na base de dados correspondem a operações realizadas sobre conjuntos. Estas operações são definidas pela álgebra relacional.

Por exemplo, **determinar quais os vendedores com tempo de casa maior que um determinado patamar**, significa determinar um subconjunto do conjunto de vendedores, onde todos os elementos possuam uma determinada propriedade. Para determinar as vendas destes vendedores, é necessário realizar a operação já citada e depois, para cada elemento do subconjunto "vendedores veteranos" é necessário determinar o subconjunto do conjunto de vendas, contendo a propriedade de ter sido realizada pelo vendedor em questão. O resultado final da consulta será a união de todos estes subconjuntos.

O problema apresentado possui também outra forma de solução. Podemos, em um primeiro momento, determinar para cada vendedor, quais as suas vendas. Teríamos então vários subconjuntos, cada um contendo as vendas de um vendedor. Feito isto, podemos verificar quais os vendedores são veteranos, formando o resultado final a partir da união dos subconjuntos associados a cada um.

**Consultas em banco de dados não passam de problemas de álgebra relacional.** Assim como acontece com a álgebra "tradicional", os operadores possuem algumas propriedades. Sabemos que  $2 \times 3 = 3 \times 2$ . Isto significa que, quando precisamos contar uma expressão de álgebra relacional para chegar a um determinado resultado, podemos fazê-lo de mais de uma forma, pois várias expressões levam ao mesmo resultado. Em outras palavras, quando o banco de dados precisa montar uma expressão algébrica para encontrar um resultado, ele deve escolher uma entre várias. Apesar de apresentarem o mesmo resultado, as expressões são diferentes, **e a diferença fará com que o banco de dados adote um diferente caminho para resolver cada uma. Escolher o caminho mais curto é uma das grandes atribuições do banco de dados.** Está é a missão do **otimizador**, um subsistema do banco de dados, responsável por determinar o **plano de execução** para uma consulta.

**A linguagem SQL (Search and Query Language) é um grande padrão de banco de dados.** Isto decorre da sua simplicidade e facilidade de uso. Ela se opõe a outras linguagens no sentido em que uma consulta SQL especifica a forma do resultado e não o caminho para chegar a ele. Ela é uma linguagem **declarativa** em oposição a outras linguagens **procedurais**. Isto reduz o ciclo de aprendizado daqueles que se iniciam na linguagem.

**Vamos comparar:**

**descrição declarativa:**

"quero saber todas as vendas feitas por vendedores com mais de 10 anos de casa."

### **descrição procedural:**

"Para cada um dos vendedores, da tabela *vendedores*, com mais de 10 anos de casa, determine na tabela de vendas todas as *v @* destes vendedores. A união de todas estas vendas será o resultado final do problema."

Na verdade, fui bem pouco honesto na comparação. Minha declaração procedural tem muito de declarativa, o que a torna mais simples. Se ela fosse realmente procedural, seria ainda mais complicada.

O fato é que, ter que informar o como fazer, torna as coisas mais difíceis. Neste sentido, SQL facilitou muito as coisas.

Porém, a partir do nosso "o que queremos", o banco de dados vai determinar o "como fazer". No problema das "vendas dos veteranos", descrevi duas formas de solucionar o problema, a primeira certamente melhor que a segunda. O objetivo deste texto é apresentar formas de dizer para o banco de dados o que queremos, ajudando-o a determinar uma forma de fazer que tenha esforço mínimo. Para chegarmos a este objetivo, lamentavelmente, teremos que nos preocupar com o "como fazer". É fato que parte da necessidade da nossa preocupação com o "como fazer" é decorrência do estágio atual da tecnologia, que ainda pode evoluir. Porém, por melhor que seja o otimizador de um banco de dados, ele poderá trocar a consulta fornecida pelo usuário por outra equivalente segundo a álgebra relacional. Em algumas situações uma consulta é equivalente à outra apenas considerando-se a semântica dos dados. Neste caso, se nós não nos preocuparmos com o "como fazer", teremos uma performance pior.

### **Uso de índices**

Quando fazemos consultas em uma tabela estamos selecionando registros com determinadas propriedades. Dentro do conceito de álgebra relacional, estamos fazendo uma simples operação de determinar um subconjunto de um conjunto. A forma trivial de realizar esta operação é avaliar cada um dos elementos do conjunto para determinar se ele possui ou não as propriedades desejadas. Ou seja, avaliar, um a um, todos os seus registros.

Em tabelas grandes, a operação descrita acima pode ser muito custosa. Imaginemos que se deseje relacionar todas as apólices vendidas para um determinado cliente, para saber seu histórico. Se for necessário varrer toda a tabela de apólices para responder esta questão o processo certamente levará muito tempo.

**A forma de resolver este problema é o uso de índices.** Índices possibilitam ao banco de dados o acesso direto às informações desejadas.

Fisicamente, a tabela não está organizada em nenhuma ordem. Os registros são colocados na tabela na ordem cronológica de inserção. Deleções ainda causam mudanças nesta ordem. **Um índice é uma estrutura onde todos os elementos de uma tabela estão organizados, em uma estrutura de dados eficiente, ordenados segundo algum critério.** Um registro no índice é composto pelo conjunto de valores dos campos que compõem o índice e pelo endereço físico do registro na tabela. Ao escrever uma consulta SQL, não informamos especificamente qual índice será usado pela consulta. **Esta decisão é tomada pelo banco de dados.** Cabe a nós escrever a

consulta de forma que o uso do índice seja possível. É preciso que nossa consulta *disponibilize* o índice.

### Possibilitando uso de colunas para acesso indexado

Na verdade, a consulta disponibiliza colunas que podem ser usadas em acesso à índices. O banco de dados, a partir das colunas disponíveis e dos índices existentes, determina a conveniência de se usar determinado índice.

Para permitir que uma coluna seja usada em acesso à índice, **ela deve aparecer na cláusula *where* de um *select***.

Por exemplo, a consulta:

```
select campol
from tabela
where campol = 3
    and campo2 > 4
    and campo3 <> 7
    and campo4 between 10 and 20
    and campo5 + 10 = 15
    and to - number (campo6) = 0
    and nvl (campo7, 2) = 2
    and campo8 like 'GENERAL%'
    and campo9 like '%ACCIDENT'
```

Disponibiliza o uso de índices nos campos campo 1, campo2 e campo4. Nos casos dos campos campo2 e campo4, o acesso a índice será para buscar registros que possuam valor numa determinada faixa.

A condição *campo3 <> 7* não disponibiliza índice por ser uma relação de desigualdade.

De fato, se a tabela possuir *n* registros, um dos quais com *campo3 = 7*, não parece razoável um índice para recuperar *N - 1* elementos.

A condição *campo.5 + 10 = 15* não permite uso de índice pela coluna *campo5* por igualar ao valor 15 **uma expressão envolvendo a coluna, e não a coluna isolada**. De fato, uma técnica para se inibir o uso de índice em uma coluna, quando desejado, é usar expressões tais como:

- nome-coluna + 0 = 15, para campos *number* ou *date*, ou
- nome-coluna || ' = 'ABCD', para campos *char*.

As expressões envolvendo as colunas *campo6* e *campo 7* também não disponibilizam índice pelo mesmo motivo. Foram incluídas aqui por se tratarem de casos em que, freqüentemente, as pessoas acham que o uso de índice seria possível.

A expressão envolvendo *campo8* disponibiliza índice, pois ela é na verdade como se escrevêssemos: *campo8 >= 'GE cccc... 'and campo8 <= 'GEddd...'* onde **c é o menor caracter** na ordem da tabela ASCII, dentro do domínio possível de valores para o campo **é d o maior**. Já a expressão envolvendo *campo9* não permite uso de índice.

Se houver um índice único na tabela pelo campo 1, o acesso disponibilizado por este índice é um **unique scan**: o banco de dados faz um acesso ao índice para buscar um único registro.

Mesmo que haja um índice único pelo *campo2*, será feito um **range scan** na tabela, ou seja, uma busca para recuperar vários registros. O mesmo ocorre para o *campo4*.

### Escolhendo um índice

Dadas as colunas que podem ser usadas para acesso indexado, o banco de dados passa a decisão sobre qual índice será usado. Para isto, ele determinará os índices disponíveis para então escolher um.

Um índice estará disponível se um **prefixo** dos campos que o compõem estiver disponível. Por exemplo, se o índice for composto pelas colunas *campo1*, *campo2*, *campo3* e *campo4*, o índice estará disponível se estiverem disponíveis as colunas:

*campo1* ou

*campo1* e *campo2* ou

*campo1*, *campo2*, e *campo3* ou

*campo1*, *campo2*, *campo3* e *campo4*.

Neste último caso, o uso do índice será completo, se ele for usado.

A seleção entre os índices para determinar qual será realmente usado é feita a partir de heurísticas, como por exemplo, **é melhor usar um índice único que um índice não único**. Esta seleção pode considerar também informações sobre os dados armazenadas na tabela, dependendo da configuração do servidor de banco de dados.

### Qual o melhor índice?

O critério básico para escolha de índices é a **seletividade**. Quando o banco de dados resolve uma consulta, freqüentemente, ele precisa percorrer mais registros do que aqueles realmente retomados pela consulta. Os registros percorridos que forem rejeitados representam o trabalho perdido.

Quanto menor for o trabalho perdido, mais perto estaremos da performance ótima para resolver a consulta. Portanto, o melhor índice para uma consulta é aquele que apresenta a maior seletividade. Vejamos a consulta abaixo:

```
select campo1
from tabela
where campo2 = 2 and campo3 = 1 and campo4 = 3;
```

tabela possui os índices:

índice 1:	campo2, campo5
índice 2:	campo1
índice 3:	campo3, campo1
índice 4:	campo4
índice 5:	campo5, campo4

Neste caso, estão disponíveis para consultas indexadas os campos *campo2*, *campo3* e *campo4* o que permite o uso dos índices 1, 3 e 4. O índice mais seletivo será aquele que recuperar o mínimo número de registros.

Se houver 10 registros com *campo2* = 2, 2000 registros com *campo3* = 1 e 50 registros com *campo4* = 3, o índice 1 será o mais seletivo. Nossa melhor alternativa é portanto um **range scan** no índice 1. Vale a pena ressaltar que o fato do índice 1 possuir também a coluna *campo5* prejudica um pouco a consulta. Ou seja, seria melhor, para esta consulta, que o índice 1 possuísse apenas o *campo2*.

Para resolver a consulta, o banco de dados fará o acesso ao índice, onde irá recuperar o endereço físico, na tabela, dos registros candidatos a compor o resultado. Com este endereço, **ele verifica cada** registro quanto **às outras condições**. Os que satisfizerem as outras condições comporão o resultado.

Em alguns casos, este cantinho pode ser mais simples. Veja o exemplo abaixo:

```
Select campo1 from tabela where campo2 = 2;
```

Tabela possui os índices:

índice 1: campo1 , campo3

índice2: campo2, campo1, campo3

índice3: campo1, campo2

Neste caso, o banco de dados apenas pode usar o índice 2. A consulta pode ser resolvida sem acesso à tabela, usando apenas o índice. Uma vez que o índice também possui os valores para *campo1* de cada registro, não há necessidade de se recuperar este valor da tabela.

### **Campos nulos não entram**

Toda vez que um registro possuir valores nulos para todos os campos que compõem um índice, este registro não será colocado no índice.

Isto causa problemas de performance em sistemas mal projetados. Suponha que a modelagem de dados de um sistema de contas a pagar tenha resultado em um projeto onde existe uma tabela de contas (ou compromissos, ou títulos) a pagar, contendo, entre outros, dois campos: *data de vencimento* e *data de pagamento*. A primeira seria a data em que o título deve ser pago. A segunda a data em que o título foi efetivamente pago. Suponha ainda que a dinâmica do sistema determine que todo título, ao ser inserido no sistema, tenha valor nulo para o campo *data de vencimento*. Quando o pagamento vier a ser efetuado, o campo será atualizado. É bastante possível que seja construída uma consulta para recuperar todos os títulos não pagos. Neste caso, não será possível o uso de índices, pois estamos procurando campos com valor nulo. Se tivermos, nesta tabela, 200000 títulos, dos quais 500 não pagos, esta consulta terá desempenho bastante aquém do possível. Uma solução melhor, seria inicializar o campo *data de vencimento* com o valor 01/01/1998, significando conta não paga.

## Tabelas pequenas

Como última consideração sobre consultas em uma tabela, vale lembrar que quando fazemos uma consulta a uma tabela bastante pequena, não compensa usar índices. O trabalho envolvido em acessar o índice para pegar o endereço e, depois, acessar a tabela é maior que o esforço de ler a tabela inteira.

## Consultas em várias Tabelas

Consultas envolvendo várias tabelas são feitas através de *joins*. Na teoria da álgebra relacional, estas consultas são produtos cartesianos entre os conjuntos (tabelas) envolvidos. Para cada elemento do conjunto resultado do produto cartesiano, será verificado se ele possui ou não um determinado conjunto de condições, imposto ao resultado da consulta.

O banco de dados irá tirar proveito de propriedades matemáticas para otimizar estas consultas. Imagine que tenhamos dois conjuntos, um de retângulos e um de círculos. Tome como objetivo obter o subconjunto do produto cartesiano destes dois conjuntos que apenas contenham círculos amarelos e retângulos azuis. Há duas formas de resolver isto. Uma é fazer o produto cartesiano, e, a partir do resultado, excluir os elementos que não atendem a premissa. Outra é determinar o subconjunto dos círculos amarelos e o subconjunto dos retângulos azuis, fazendo um produto cartesiano dos dois subconjuntos. Apesar de equivalentes quanto ao resultado, o segundo método é bastante mais eficiente em termos de performance.

Normalmente, quando se faz uma consulta em duas ou mais tabelas, existe alguma informação que as une. Por exemplo, você quer relacionar registros de um determinado empregado apenas ao registro do departamento onde este empregado trabalha. Esta condição é chamada de *condição de join*. Normalmente, esta condição evita que seja necessária a realização de um produto cartesiano de fato. Vejamos o exemplo:

```
select depto.nome, emp.nome
from    empregados emp, departamentos depto
where   emp.data_admissao < '01-jan-92'
and depto.id - departamento = emp.departamento and depto.area-de~negocio = 'FAST~FOOD';
```

Neste caso, estamos trabalhando sobre dois conjuntos: **empregados e departamentos**. Possuímos restrições sobre estes dois conjuntos e uma restrição que serve para *juntá-/os*.

O banco de dados pode resolver a consulta acima de duas formas. A primeira envolve determinar o subconjunto dos empregados e departamentos que obedecem as restrições nestes conjuntos. Posteriormente, geramos o resultado a partir dos dois subconjuntos, respeitando a **condição de join**. A segunda forma envolve escolher um conjunto para iniciar a solução, por exemplo o de departamentos. Determinamos o subconjunto deste que possua a propriedade apresentada (determinada área de negócio). Para cada elemento deste subconjunto, faremos sua associação ao elemento correspondente no outro conjunto, segundo a condição de join. Finalmente, verificamos se o registro formado possui a restrição no outro conjunto. A primeira forma de solução é chamada de **sort-merge join**. A segunda é chamada de **nested-loops**. O banco de dados sempre usa uma destas duas técnicas para resolver consultas em múltiplas tabelas. Ele pode mesmo, combinar as duas.

Na maioria dos casos, o método de **nested-loops** apresenta melhores resultados. Em alguns casos complexos, o **sort-merge** é indicado.

## Outer join

**O.outer join(ou junção extema) é um caso particular do join.** Quando se faz um join simples, a impossibilidade de se encontrar em alguma tabela um registro associado ao resultado intermediário anterior, determina que o join não retomará nenhuma informação.

Quando uma tabela entra no **join** através de um **outer join**, quando não houver registros na tabela associados ao resultado intermediário, uma linha imaginária será adicionada à tabela em questão, contendo valor nulo para todos os seus campos. Esta linha será juntada aos registros de resultado intermediário, compondo o novo resultado intermediário.

Um Outer join não causa uma lentidão de processamento muito maior que a de um join.

## Sub Select

Outro mito que ronda a construção de SQL diz que é melhor usar Sub Select do que usar um join. Dependendo de quem reproduz o mito, a situação se inverte, e o join passa a ser melhor que o Sub Select. Na verdade, deve-se fazer uma análise caso a caso. As duas técnicas podem inclusive, serem equivalentes.

Vejamos o exemplo:

```
select campo1
```

```

from tabela1
  where campo2 in (Select campo2
                   from tabela2 where campo4 = constante)

```

e

```

select  campo1
from    tabela1, tabela2
where   campo4 = constante and
tabela1.campo2 = tabela2.campo2

```

Neste caso, primeiro devemos notar que as duas consultas só são equivalentes se houver unicidade nos valores do campo3. Caso contrário, pode haver diferença de resultados, quanto ao número de registros retornados. Imagine que as duas tabelas tenham três registros cada uma, todos com o mesmo valor para campo2 (na tabela1) e o mesmo valor para campo3 (na tabela2). Imagine também que todas as linhas da tabela1 tenham o mesmo valor para campo 1. No caso da sub select, a consulta retorna três linhas como resultado. No caso do join, retorna 9 linhas. Portanto, para serem equivalentes de fato, é necessário que exista unicidade nos valores de campo3 ).

Se houver a unicidade, o uso do sub select . ou join (com relação à performance) é absolutamente equivalente. De fato, para resolver esta consulta, em qualquer dos dois casos, o banco de dados irá determinar os registros da tabela2 que possuem "campo4 = constante". Irá recuperar então o valor de campo3 para estes registros. Para cada valor, obterá os registros da tabela1 com campo2 igual a este valor. Os valores de campo1 nestes registros comporão os conjuntos resultados.

Vejamos unia situação onde precisamos saber todas as notas fiscais que contenham itens fabricados por um determinado fornecedor, em um determinado dia.

Sub Select:

```

select num nota
from   notas Fiscais
where  data-emissao = constante and exists (select 'x'
                                           from produtos , itens-nota
                                           where produtos.num-nota = itens-nota.num- nota and
                                           produtos.produto = itens-nota.produto and
                                           fornecedor = constante)

```

Join:

```

select distinct num - nota

```

```
from produtos , itens - nota , notas-fiscais
where notas-fiscais.data-emissao = constante and
itens-notas.num-nota = notas-fiscais.num-nota and
produtos.produto = itens-nota.produto and
produto.fornecedor = constante
```

Existe uma diferença apreciável de performance entre as duas consultas. Note que, o importante é saber se existe pelo menos um item de uma nota fiscal associado a um determinado fornecedor. Não é necessário determinar **todos** os itens deste fornecedor. Este é o ponto onde a consulta com join torna-se pior, em termos de performance que a consulta com sub select. Note que a data da nota foi incluída para ser um fato seletivo e justificar o exemplo. Se não houvesse tal restrição, o jeito certo de construir a consulta seria determinar todos os produtos do fornecedor, depois os itens de nota com este produto e finalmente as notas correspondentes. Desde, é claro, que um fornecedor seja razoavelmente seletivo. Neste caso, teríamos um exemplo onde join é melhor do que sub select. Apenas por uma diferença sutil de existência de um critério de data.