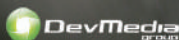


Feita para Desenvolvedores de Software e DBAs

SQL

magazine

Edição 66 :: Ano 5 : R\$ 11,90



Desafio SQL: Coloque à prova seus conhecimentos em banco de dados

Projeto

Aprenda a elaborar Diagramas de Atividades para projetos de software orientados a objetos

PostgreSQL

Saiba como aplicar técnicas de tuning para melhoria de desempenho

ORACLE

Aprenda a Estruturar seus Scripts utilizando um Help que pode ser Acessado Remotamente

Veja como Configurar um Repositório Centralizado de Autenticação utilizando o Oracle Internet Directory

Guia para Instalação e Configuração do Oracle Data Integrator

Brinde!

Compre esta edição e ganhe **2 vídeo-aulas**

Conceitos de OO - Parte 1 (Conceitos, Classes, Objetos e Atributos)

Conceitos de OO - Parte 2 (Encapsulamento, Métodos Getters e Setters)

SQL Server

Entenda e veja como trabalhar com o SQL Server Agent

SQL Server

Veja como administrar o SQL Server 2008 através de políticas

ISSN 1677918-5



Nesta seção você encontra artigos sobre banco de dados, SQL ou persistência



Aplicando Técnicas de Tuning para Melhoria de Desempenho em Banco de Dados PostgreSQL

De que se trata o artigo?

O Artigo trata de técnicas para melhoria de desempenho em bancos de dados PostgreSQL.

Para que serve?

A aplicação de tais técnicas é algo vital para se obter um melhor desempenho de bancos com grande acesso a determinadas tabelas que precisam de uma resposta rápida.

Em que situação o tema é útil?

A utilização do tuning é útil para maioria de bancos com problemas de tempo de resposta.

Com o tempo, as consultas, inserções, modificações e exclusões em um SGBD começam a ficar lentas, as tabelas começam a ficar grandes demais fazendo com que o SGBD perca desempenho. Quando isso acontece, a única solução é a aplicação de técnicas de *tuning* de desempenho no SGBD.

Este artigo aborda a aplicação de tais técnicas de *tuning* para melhoria do desempenho de bancos de dados

PostgreSQL. São apresentados os principais problemas encontrados em algumas consultas ao banco de dados nessa etapa e, em seguida, as respectivas soluções sugeridas. Durante o artigo são apresentadas situações sobre como a aplicação de técnicas de *tuning* pode melhorar o desempenho de um banco de dados. O objetivo do artigo é apresentar possíveis soluções de melhorias em um banco de dado PostgreSQL com ênfase em melhoria no seu desempenho.



Vinicius Aquino do Vale

viniaquino.vale@hotmail.com

Graduado em Ciência da Computação. Analista de TI, possui experiência de 4 anos na área de Ciência da Computação, com ênfase em Redes e Engenharia de Software, especialista em PostgreSQL e linguagem PG/SQL.

Tuning em Banco de Dados

O processo de *tuning* é algo que só deve ser realizado quando houver uma real necessidade, pois envolve uma grande quantidade de processos tanto do SGBD como do sistema operacional. *Tuning* não é só configuração do SGBD, algumas vezes requer configuração no sistema operacional ou otimizar as consultas realizadas nos bancos, e algumas configurações podem acarretar em problemas futuros ou até mesmo mau funcionamento do SGBD.

A técnica de *tuning* de desempenho é uma prática feita em todos os SGBDs importantes com técnicas diferentes. É uma necessidade de toda grande empresa, e requer um grande nível de conhecimento e experiência sobre os processos e suas técnicas, pois nem sempre o problema será resolvido com uma ou duas alterações, às vezes é necessário uma mudança nas queries.

Prática do Tuning no PostgreSQL

Uma forma de melhorar o desempenho das tabelas com grandes quantidades de registros e especialmente com muitos acessos através do PostgreSQL é a inclusão de índices estratégicos. Além da chave primária, é importante inserir índices em campos que compõem a cláusula WHERE ou que fazem parte de cláusulas ORDER BY, GROUP BY. Na criação do banco de dados e especialmente na criação das consultas, é muito importante atentar para um bom planejamento, normalização e consultas otimizadas tendo em vista o planejador de consultas do PostgreSQL através do uso dos comandos EXPLAIN e ANALYZE.

A administração do PostgreSQL também é muito importante para tornar o SGBD mais eficiente e rápido. Desde a instalação e configuração temos cuidados que ajudam a otimizar o PostgreSQL.

Aplicando Tuning no PostgreSQL

A prática de *tuning* não é algo que se faz simplesmente por fazer. É necessário ter conhecimento e prática sobre o assunto. Algumas técnicas de tuning são simples e de fácil compreensão, mas algumas requerem conhecimento sobre Sistema Operacional ou Hardware.

A aplicação de técnicas de *tuning* eventualmente requer muita análise, devendo sempre trabalhar cada caso de uma forma diferente, ou seja, não existe uma “receita de bolo” pré-definida para a aplicação de *tuning* em um banco de dados. As técnicas que usaremos neste artigo são técnicas simples de uso do próprio banco. Possíveis técnicas a serem aplicadas no Sistema Operacional ou no hardware serão somente citadas, sem maiores detalhes.

No nosso estudo de caso estamos trabalhando com a versão mais atual do PostgreSQL, a versão 8.3RC2, e com o sistema operacional Windows Vista.

Estudo de Caso

No nosso estudo de caso, teremos a situação de uma tabela de *clientes* com grande quantidade de registros, e usaremos técnicas de *tuning* para melhorar a velocidade de resposta da consulta.

A partir de agora iremos inicialmente construir nosso cenário de aplicação de *tuning*, para depois aplicar as técnicas. Sendo assim, inicialmente vamos criar um BD com o nome *SQLMagazine*. A codificação WIN1252 será usada por ser padrão do PostgreSQL 8.3RC2. O código de criação está apresentado na **Listagem 1**.

Listagem 1. Criando o banco de dados de nosso estudo de caso

```
CREATE DATABASE "SqlMagazine" WITH OWNER = postgres ENCODING = 'WIN1252';
```

Depois de criado o banco de dados, devemos criar uma tabela com o nome *clientes*, cujos campos sejam *codcli* (chave primária – integer), *nome* (varchar(30)), *idade* (integer) e *ativo* (boolean). Estes campos serão usados para nosso exemplo de criação de índices. O código de criação da tabela *clientes* está apresentado na **Listagem 2**.

Listagem 2. Criando a tabela clientes

```
CREATE TABLE clientes (
    codcli
    serial NOT NULL,
    nome
    character varying(30) NOT NULL,
    idade
    integer NOT NULL,
    ativo
    boolean NOT NULL DEFAULT false,
    CONSTRAINT pk_codcli PRIMARY KEY (codcli)
) WITH (OIDS=FALSE);
ALTER TABLE clientes OWNER TO postgres;
```

Agora que nossa tabela *clientes* foi criada, devemos preenchê-la com dados válidos. Como nosso objetivo é aplicar técnicas *tuning* para prover melhorias de desempenho, devemos preencher nosso banco com algo em torno de 100.000 (cem mil) registros para que possamos ter uma noção de tempo de desempenho e a influência da técnica de *tuning* no banco de dados. Sendo assim, para facilitar a criação de registro foram criadas 2 (duas) funções para inserção de dados na nossa tabela. O código das funções está descrito na **Listagem 3**.

A função *ins_nome* foi criada para gerar caracteres aleatórios para preenchimento do campo *nome* da tabela *clientes*. A função recebe um valor inteiro que pode variar de 1 a 30, e a função então gera um loop e começa a criar caracteres para nosso campo *nome*. A função *ins_cli* insere o nome dos clientes através da função *ins_nome*, e também gera o registro dos campos *idade* e *ativo* de forma aleatória, uma forma de preencher a tabela *clientes* com vários registros válidos.

As funções devem ser criadas nessa ordem. Depois de criadas as funções, devemos executar a função *ins_cli*, como mostra a **Figura 1**.

Com este comando o banco vai inserir registros (nome, idade e ativo) aleatórios, e dependendo da configuração da sua máquina pode demorar alguns minutos para inserção de todos os registros. No ambiente onde esse teste foi realizado para a escrita do artigo, demorou-se 40747 milissegundos (cerca de 40 segundos). Depois de inseridos os registros, podemos começar a fazer o *tuning* do banco de dados.

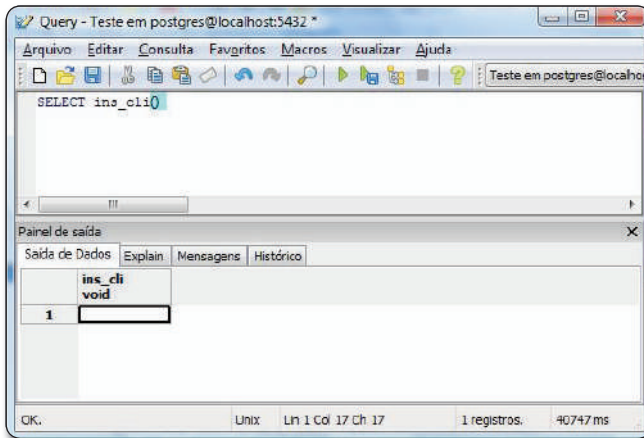


Figura 1. Gerando registros para tabela *clientes*.

Listagem 3. Funções para Povoamento da tabela clientes

```
CREATE OR REPLACE FUNCTION ins_nome(qtde integer)
RETURNS character varying AS $BODY$
declare
    nome varchar;
    num int;
    i int;
begin
    nome = '';
    FOR i IN 1.. $1 LOOP
        num = (select (random()*125)::int);
        if (num <= 64) or (num >= 122) then
            num = (select (random()*10)::int)+65;
        end if;
        nome = nome || (select chr(num));
    end loop;
    return nome;
end
$BODY$
LANGUAGE 'plpgsql' VOLATILE
COST 100;
ALTER FUNCTION ins_nome(integer) OWNER TO postgres;

CREATE OR REPLACE FUNCTION ins_cli()
RETURNS void AS $BODY$
declare
    i int;
    inicial int;
    final int;
begin
    inicial = (select count(*)+1 from clientes);
    final = (select count(*)+1 from clientes)+100000;
    for i in inicial..final loop
        insert into clientes values(i, (select ins_nome((select
        (random()*20+1)::int))), (select (random()*100+1)::int), (select
        (random())::boolean));
    end loop;
end
$BODY$
LANGUAGE 'plpgsql' VOLATILE
COST 100;
ALTER FUNCTION ins_cli() OWNER TO postgres;
```

Aplicando Técnicas de Tuning

Agora que nossa tabela já está com muitos registros, vamos fazer uma consulta simples sem o uso de índices, como mostra a Figura 2.

Usando uma simples consulta buscando todos os clientes com idade igual a 23 (vinte e três) anos, foram retornados 1.002 registros com o tempo de 32 milissegundos, lembrando que a função cria idades aleatórias na nossa tabela, ou seja, quando vocês forem testar em sua máquina a quantidade pode ser superior ou inferior àquela obtida em nossos testes e apresentada na Figura 2.

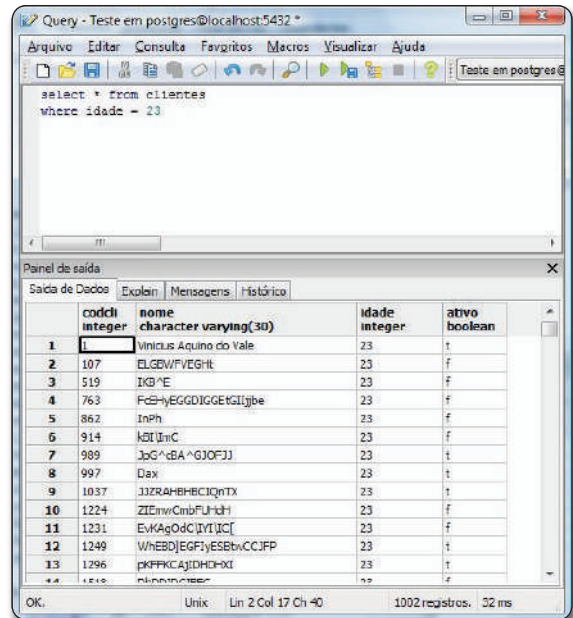


Figura 2. Consulta simples sem usar índice.

Podemos usar o comando *Explain* para vermos como o banco analisou a tabela e a forma que adotou para retornar os registros. Sendo assim, observamos que o banco faz uma varredura seqüencial na tabela, ou seja, ele faz a leitura de todos os registros um a um, como mostra Figura 3.

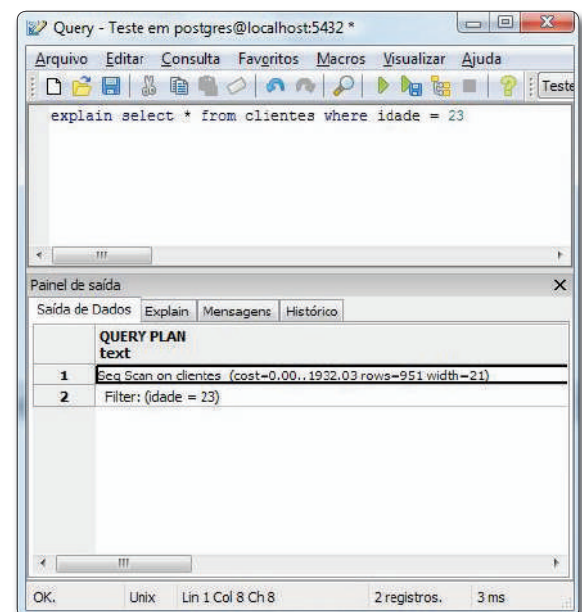


Figura 3. Explicação de como foi feita a busca na tabela.

Técnica de Tuning: Criação de Índices

Agora iremos aplicar uma estratégia de *tuning*: a criação de um índice. Criando um índice para a tabela *clientes* no campo *idade*, podemos diminuir o tempo de reposta do servidor para consultas. Como dica, um índice somente deve ser criado para tabelas cuja quantidade mínima de registros seja 1.000, pois só assim o índice terá grandes benefícios.

Vamos criar um índice para o campo *idade* da tabela *clientes*. O código para criação do índice está apresentado na **Listagem 4**.

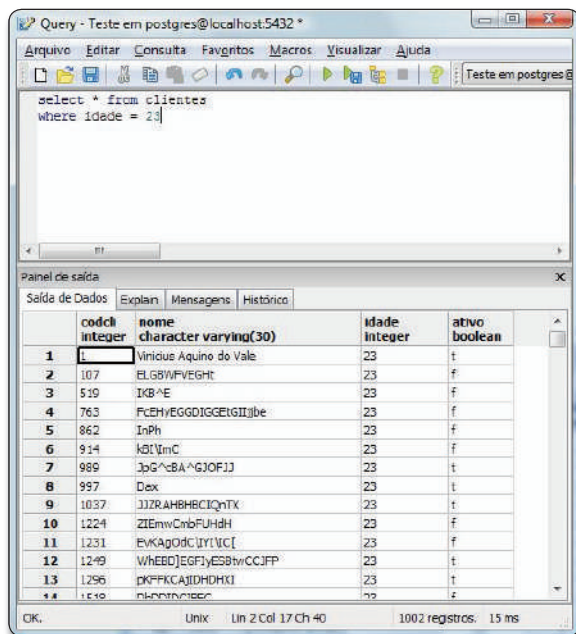
Listagem 4. Criação de um índice para a coluna *idade*

```
CREATE INDEX cl_idade ON clientes USING btree (idade);
```

Além do método B-tree, existem outros três métodos: *Hash*, *R-Tree* e o *GIST*.

- **B-Tree:** Implementa árvores B e é o método padrão. É usado nas seguintes operações: <, <=, =, >=, >, LIKE, BETWEEN.
- **Hash:** Cria uma tabela de *hashing* para indexação. Somente usado para operações de igualdade (=). Seu uso não é recomendado.
- **R-Tree:** Deve ser usado somente para indexar tipo de dados geométricos (*box*, *line*, *circle*, etc). Será descontinuado em favor de índices **GIST**.
- **GIST:** Fornece uma estrutura de árvore padrão para ser estendida, e pode ser usada com novos tipos de dados.

Depois de criado o índice, vamos refazer a mesma consulta de antes e veremos qual o resultado, como mostra a **Figura 4**.



The screenshot shows the PostgreSQL Query Editor with the query `select * from clientes where idade = 23`. The results panel displays a table with 13 rows and 4 columns: *codcli* (integer), *nome* (character varying(30)), *idade* (integer), and *ativo* (boolean). The first row is highlighted.

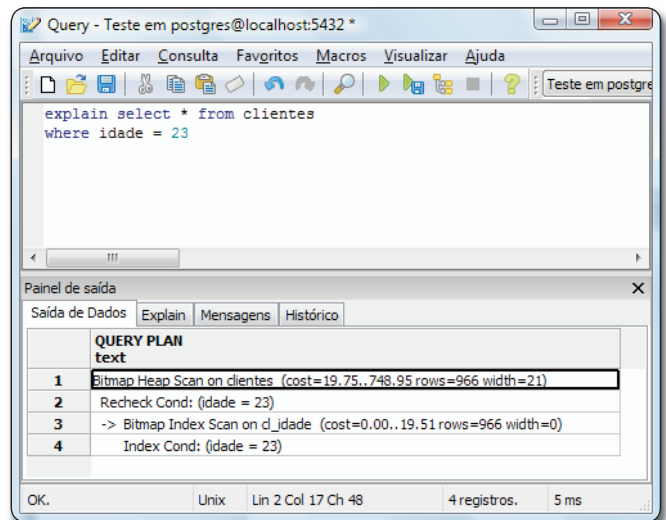
	codcli integer	nome character varying(30)	idade integer	ativo boolean
1	1	Vinicius Aquino do Vale	23	t
2	107	ELGBWVVEGHT	23	f
3	519	IKB^E	23	f
4	763	FcEHYEGGDIIGGEGIIjbe	23	f
5	862	InPh	23	f
6	914	k8VimC	23	f
7	989	3pG^cBA^G3OFJJ	23	t
8	997	Dex	23	t
9	1037	JJZRAHBHBCIQnTX	23	f
10	1224	ZIEmwCmbFUHdH	23	f
11	1231	EyKagOdCjYVVC[23	f
12	1249	WHEBD]EGfjyES9twCCJFP	23	t
13	1296	pK^FKCAJIDHDHXI	23	t

Figura 4. Consulta simples usando índice.

Com o índice criado, a consulta demorou apenas 15 milissegundos, ou seja, a metade do tempo da consulta sem o índice. Para confirmarmos a diferença de desempenho, devemos parar o serviço do SGBD ou reiniciar a máquina antes de fazer a segunda consulta, pois alguma informação pode já estar carregada em memória, o que tornaria a segunda tentativa mais rápida que a primeira.

Em um ambiente de cliente/servidor, poderia aumentar ainda mais já que o índice não faz a leitura total da tabela, como poderemos observar na **Figura 5**.

Para criação de índice, tem-se a opção de qual método utilizar para o índice, lembrando que o método B-tree é o mais recomendado para a maior parte das aplicações. Usando o comando *Explain* depois da criação do índice, podemos observar que o processo de busca é diferente da varredura sequencial, como mostra a **Figura 5**.



The screenshot shows the PostgreSQL Query Editor with the query `explain select * from clientes where idade = 23`. The results panel displays the query plan, showing a Bitmap Heap Scan on *clientes* (cost=19.75..748.95 rows=966 width=21) and a Bitmap Index Scan on *cl_idade* (cost=0.00..19.51 rows=966 width=0).

	QUERY PLAN
1	text
2	Bitmap Heap Scan on clientes (cost=19.75..748.95 rows=966 width=21)
3	Recheck Cond: (idade = 23)
4	-> Bitmap Index Scan on cl_idade (cost=0.00..19.51 rows=966 width=0)

Figura 5. Explicação de como foi feita a busca na tabela usando o índice.

O uso de índices é um procedimento de *tuning* muito prático, lembrando que se pode criar um índice concatenado, ou seja, para mais de um campo, como mostra o código da **Listagem 5**.

Listagem 5. Criação de um índice concatenado para as colunas *idade* e *ativo*

```
CREATE INDEX cl_idade_ativo ON clientes USING btree (idade, ativo);
```

Quando vários índices são candidatos para serem utilizados numa consulta, o otimizador escolhe o mais restritivo (desde que existam estatísticas). Como os índices usam o método B-tree, comandos de INSERT, DELETE e UPDATE desbalanceiam a árvore, e para que isso não ocorra existem operações de rotação e split que acontecem automaticamente. Quando uma tabela com intensas modificações nos dados possui muitos índices, as operações de rotação e split consomem uma grande quantidade de recursos do sistema, podendo deixá-lo lento.

Técnica de Tuning: Indexação Parcial

O PostgreSQL oferece a opção de indexação parcial, sendo muito útil para eliminação de linhas com valores mais concentrados e, assim, deixar somente os valores mais seletivos, como mostra o código da **Listagem 6**.

Listagem 6. Criação de um índice parcial para as colunas *idade* e *ativo*

```
CREATE INDEX ind_idade_where ON clientes USING btree (idade) WHERE idade >= 18 AND idade <= 25 AND ativo IS TRUE;
```

Imagine uma situação onde a maioria das consultas à tabela *clientes* seja com a condição de idade entre 18 e 25 e que os clientes estejam ativos. Nesse sentido, poderíamos criar um índice que fizesse essa condição, e assim melhoraríamos o tempo de resposta da consulta.

Em algumas situações é necessário reconstruir os índices do sistema. Isto ocorre quando um índice foi corrompido. Embora raro, isto pode acontecer devido a falhas de hardware ou software, ou um índice contém muitos blocos com informações mortas (linhas removidas/atualizadas). A reindexação elimina estes blocos com a reconstrução do índice. A reindexação é feita pelo comando `REINDEX`, como mostra o código da **Listagem 7**, onde estamos recriando o índice *cl_idade*, criado originalmente na **Listagem 4**.

Listagem 7. Reindexação do índice *cl_idade*

```
REINDEX INDEX cl_idade;
```

Qualquer função ou operador sobre uma coluna indexada impede a utilização do índice, como mostra a **Figura 6**.

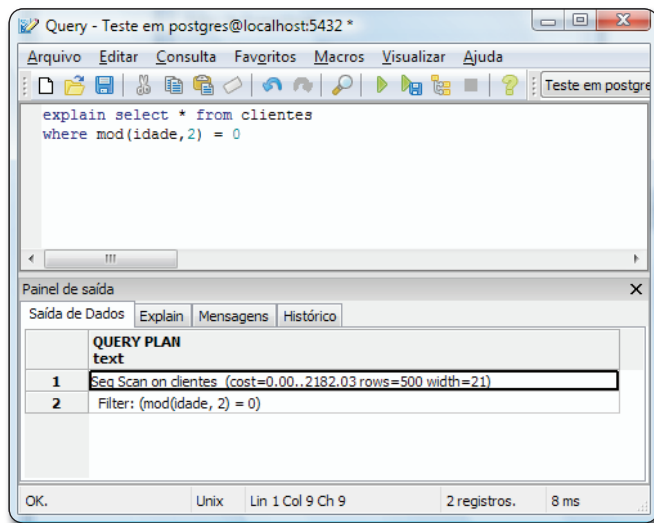


Figura 6. Função e Operador sobre índices.

Técnica de Tuning: Criação de Índices usando Funções

Um operador ou uma função em uma consulta impede que o SGBD utilize o índice para melhor desempenho do banco. Caso seja *rigorosamente* necessário utilizar funções ou operadores sobre colunas indexadas, o PostgreSQL oferece o recurso de criação de índices utilizando funções, como mostra o código da **Listagem 8**.

Listagem 8. Criação de índice sobre funções ou operadores.

```
CREATE INDEX ind_idade_funcao ON clientes USING btree
(idade) WHERE mod(idade, 2) = 0;
```

Após a criação do índice com a função, podemos observar que o SGBD já faz a busca usando este índice, como mostra a **Figura 7**.

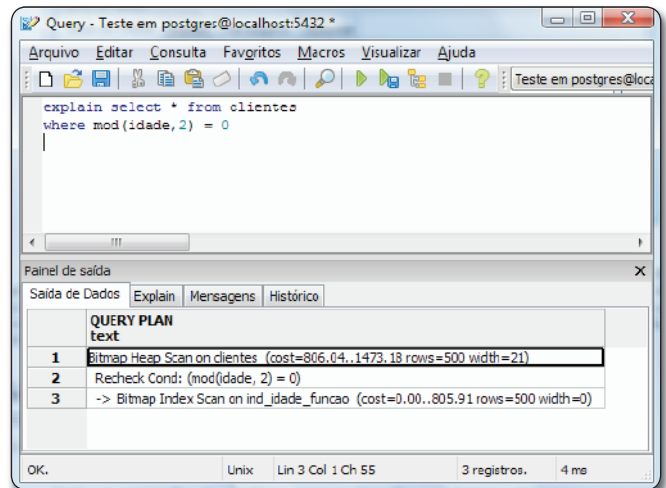


Figura 7. Consulta com índice sobre função.

Técnica de Tuning: O método CLUSTER

Imagine a situação onde a coluna *clientes.idade* tenha muitas linhas com valores duplicados. Um índice sobre idade possibilitaria uma recuperação rápida das linhas. Entretanto, se um número alto de linhas existe, elas estarão espalhadas pelo disco.

O comando `CLUSTER` reordena fisicamente uma tabela de acordo com um índice existente. Este recurso é útil para índices com muitas chaves duplicadas. Repetindo, ele reordena a tabela colocando as linhas com chave duplicadas juntos. Isto diminui o tempo de acesso aos dados. Se uma nova linha é adicionada, a ordem é quebrada. Mas o acesso indexado continua eficiente, o comando `CLUSTER` também é útil para consultas com intervalos.

Existe um *script* chamado *clusterdb* que pode ser utilizado para clusterizar o banco, como mostra a **Listagem 9**.

Listagem 9. Clusterizar um Banco.

```
C:\Arquivos de programas\PostgreSQL\8.2\bin>Clusterdb -d
SQLMagazine -U postgres
```

Com este comando o banco *SQLMagazine* será todo clusterizado. Para outras opções e maiores informações utilize *clusterdb --help*.

Otimizações Sintáticas

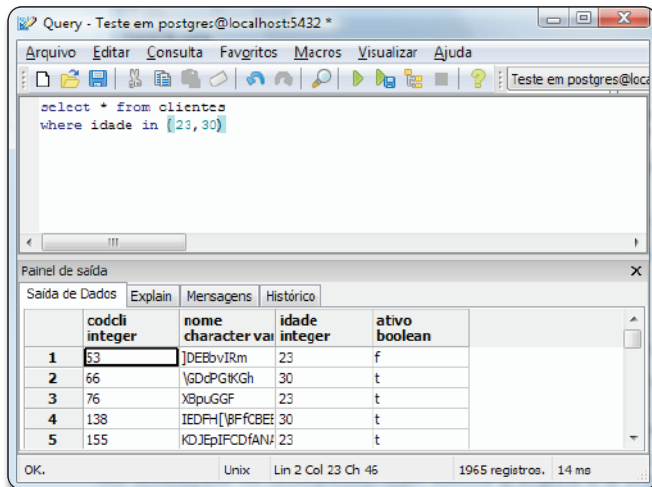
Nem sempre o índice ajuda no desempenho do banco de dados, caso a sintaxe da consulta não esteja normalizada.

Para otimizar uma consulta com base na sintaxe, deve-se desconsiderar os fatores não sintáticos: índices, tamanho da tabela, etc. Vale ressaltar que apenas algumas instruções SQL possuem opções que tornam isto possível. O comando `SELECT` é o principal candidato a otimizações sintáticas.

Por simplicidade, consideraremos somente consultas com uma única tabela e otimizações na cláusula `WHERE`.

O PostgreSQL usa índice em consultas que tenham `LIKE` e comecem com um caractere real, mas não utilizará um índice se começar com um coringa (`%` ou `_`).

Apesar de terem o mesmo significado, as condições OR e IN podem ter diferenças de desempenho, como mostram as Figuras 8 e 9.



Query - Teste em postgres@localhost:5432 *

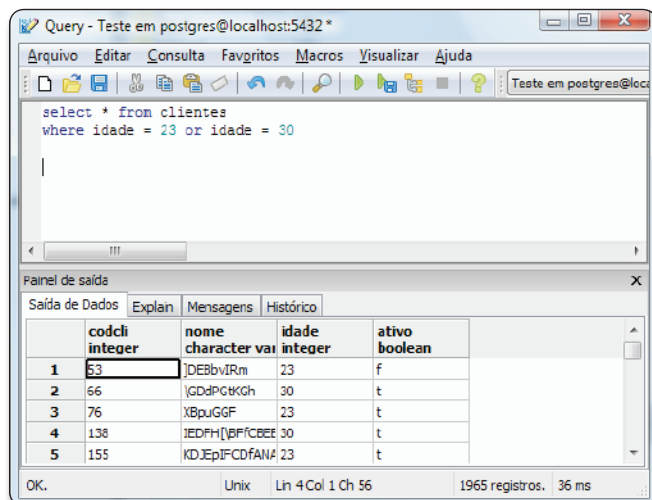
```
select * from clientes
where idade in (23,30)
```

Painel de saída

	codcli integer	nome character vari	idade integer	ativo boolean
1	63	JDEBbvIRm	23	f
2	66	YGdPGKqGh	30	t
3	76	XBpuGGF	23	t
4	138	JEDFH[BFFCBEE	30	t
5	155	KDJEplFCDFANA	23	t

OK. Unix Lin 2 Col 23 Ch 45 1965 registros. 14 ms

Figura 8. Consulta usando o IN.



Query - Teste em postgres@localhost:5432 *

```
select * from clientes
where idade = 23 or idade = 30
```

Painel de saída

	codcli integer	nome character vari	idade integer	ativo boolean
1	63	JDEBbvIRm	23	f
2	66	YGdPGKqGh	30	t
3	76	XBpuGGF	23	t
4	138	JEDFH[BFFCBEE	30	t
5	155	KDJEplFCDFANA	23	t

OK. Unix Lin 4 Col 1 Ch 56 1965 registros. 36 ms

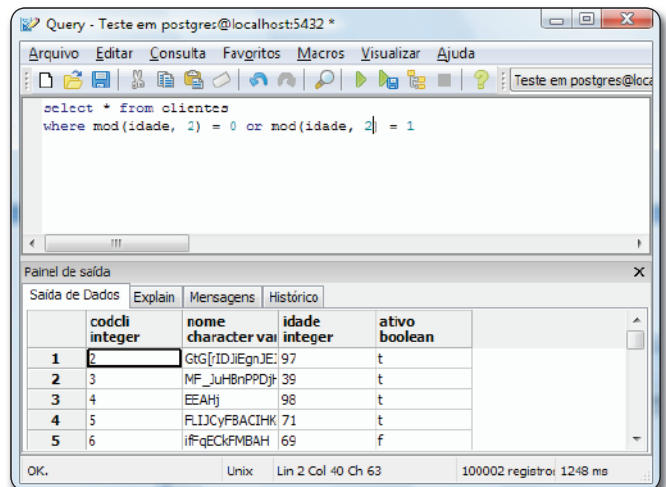
Figura 9. Consulta usando o OR.

Algumas operações sintáticas com uso de CASE também podem melhorar o desempenho do banco, como mostram as Figuras 10 e 11.

O PostgreSQL avalia uma condição AND da esquerda para direita. Colocar a expressão menos provável primeiro pode trazer ganhos. Simetricamente, numa série de expressões OR, coloque a mais provável primeiro, e sempre que possível, transforme um UNION em OR.

Avaliando os Planos de Execução

O plano de execução mostra como as tabelas referenciadas pela consulta serão varridas e quais algoritmos de junção serão usados para unir os registros requisitados de cada tabela. A parte mais crítica exibida é o custo estimado da execução da query, que é a estimativa feita pelo otimizador relativa à duração da execução da query (medida em unidade de acesso às páginas do disco).



Query - Teste em postgres@localhost:5432 *

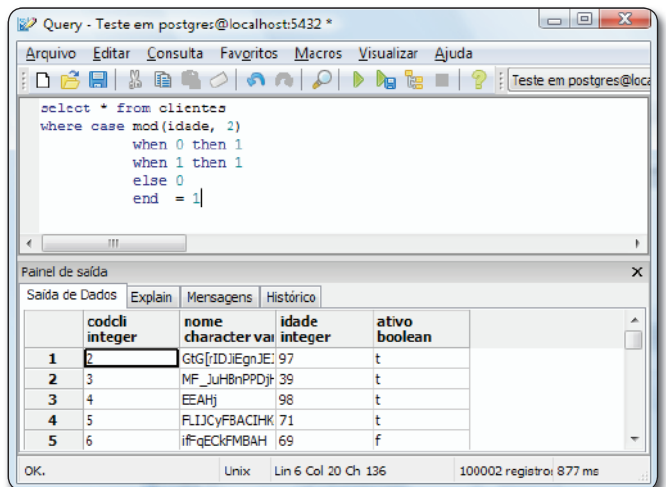
```
select * from clientes
where mod(idade, 2) = 0 or mod(idade, 2) = 1
```

Painel de saída

	codcli integer	nome character vari	idade integer	ativo boolean
1	2	GTG[IDJEgnJE	97	t
2	3	MF_JuHbnPPDj	39	t
3	4	EEAHj	98	t
4	5	FLJCYFBACIHK	71	t
5	6	iffqECKfMBAH	69	f

OK. Unix Lin 2 Col 40 Ch 63 100002 registros 1248 ms

Figura 10. Consulta usando função.



Query - Teste em postgres@localhost:5432 *

```
select * from clientes
where case mod(idade, 2)
when 0 then 1
when 1 then 1
else 0
end = 1
```

Painel de saída

	codcli integer	nome character vari	idade integer	ativo boolean
1	2	GTG[IDJEgnJE	97	t
2	3	MF_JuHbnPPDj	39	t
3	4	EEAHj	98	t
4	5	FLJCYFBACIHK	71	t
5	6	iffqECKfMBAH	69	f

OK. Unix Lin 6 Col 20 Ch 136 100002 registros 877 ms

Figura 11. Consulta usando CASE.

O comando *EXPLAIN* é o responsável por mostrar o plano de execução que o otimizador do PostgreSQL escolhe para a query fornecida. O *EXPLAIN* gera as seguintes saídas:

- Custo Inicial: custo de operações realizadas antes da recuperação de linhas como a ordenação.
- Custo Total: custo total de execução da operação.
- Número de linhas geradas pelo plano.
- Tamanho médio dos registros que serão retornados pelo plano de execução.

Para a maioria das consultas, o tempo total é o que interessa, mas em contextos como os das sub-consultas EXISTS, o planejador escolhe o menor tempo inicial em vez do menor tempo total.

Execute o comando *ANALYZE* (ou *VACUUM ANALYZE*) antes do comando *EXPLAIN*, como mostra a sintaxe:

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] consulta
```

A opção *ANALYZE* faz a consulta ser realmente executada, e não apenas planejada. O tempo total de duração gasto dentro de cada parte do plano (em milissegundos) e o número

total de linhas realmente retornadas são adicionados ao que normalmente é mostrado. Esta opção é útil para ver se as estimativas do otimizador estão próximas da realidade. Tenha em mente que a consulta é realmente executada quando a opção ANALYZE é usada. Embora o EXPLAIN despreze qualquer saída que o SELECT possa produzir, os outros efeitos colaterais da consulta acontecem na forma usual.

A opção VERBOSE fornece a representação interna completa da árvore do plano, em vez de apenas seu sumário. Geralmente esta opção é útil apenas para fazer o *debug* do próprio PostgreSQL, como mostra a **Figura 12**.

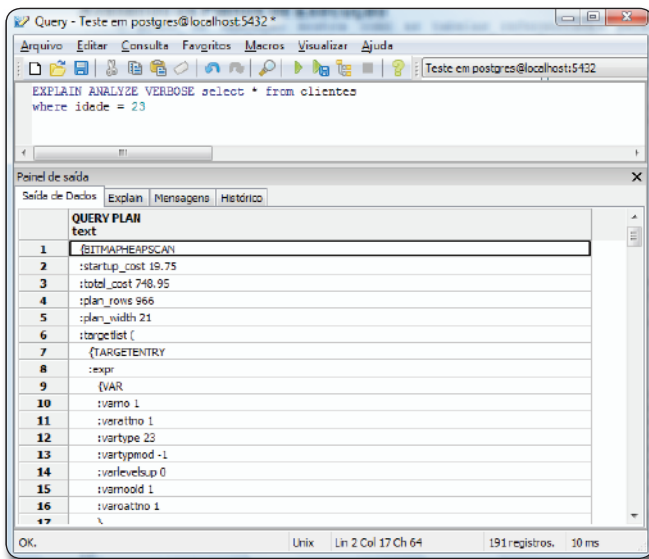


Figura 12. Comando VERBOSE.

O Otimizador pode escolher entre três tipos de junção (join) quando a query tem mais de uma tabela:

- **Nested loop:** A tabela da direita é varrida para cada linha encontrada na tabela da esquerda. Este tipo de estratégia pode ser bastante lenta caso não seja utilizado um índice para acessar a tabela da direita.
- **Merge sort:** Cada tabela é ordenada pelas colunas usada no *join* da cláusula *WHERE* antes de iniciar a junção. Depois disto, duas tabelas são combinadas considerando a ordenação e as cláusulas do *WHERE*. Este tipo de junção é interessante por que cada tabela é varrida somente uma vez.
- **Hash:** A tabela da direita é primeiramente varrida e carregada numa tabela de *hashing*, usando as colunas no *WHERE* como chaves de *hash*. Depois, a tabela da esquerda é varrida e as colunas apropriadas de cada linha são utilizadas como chave de *hash* e verificadas com a tabela de *hashing*.

É possível dirigir a escolha do otimizador em relação aos tipos de junção. Os parâmetros abaixo do *postgresql.conf* configuraram o funcionamento do otimizador. Todos eles também podem ser alterados para a sessão somente:

- **enable_seqscan (padrão: true):** habilita/desabilita leituras sequenciais. O otimizador utilizará índices sempre que houver.

- **enable_nestloop (padrão: true):** habilita/desabilita junções nested loop.
- **enable_mergejoin (padrão: true):** habilita/desabilita junções merge.
- **enable_hashjoin (padrão: true):** habilita/desabilita junções hash.

É possível controlar a ordem de varredura das tabelas utilizando cláusulas de junção explícitas. Quando existem poucas tabelas, a preocupação com a ordem não é muito importante. Entretanto, quando muitas tabelas são envolvidas, o número de possibilidades de varredura cresce fatorialmente e as verificações exaustivas podem tomar muito tempo do servidor.

Quando há um número alto de tabelas, o PostgreSQL adota um algoritmo chamado “*genetic probabilistic search*” e estabelece um número limitado de possibilidades. As buscas genéticas nem sempre geram os melhores planos de execução.

Como dito anteriormente, a aplicação de *tuning* para melhoria de desempenho de um SGBD pode muitas vezes ser aplicada através da configuração de hardware e software.

Existem dois tipos de configuração de memória no PostgreSQL, a compartilhada e a individual. A compartilhada tem um tamanho fixo. Ela é alocada sempre que o PostgreSQL inicializa e então compartilha para todos os clientes. A memória individual tem um tamanho variável e é alocada separadamente para cada conexão.

Memória Cache Compartilhada do PostgreSQL

Na configuração default do PostgreSQL 8.1.3, ele aloca 1000 *shared buffers*. Cada buffer usa 8KB, o que soma 8MB. Aumentar o número de buffers fará com que os clientes encontrem as informações que procuram em cache e evita requisições onerosas ao sistema operacional. Mas cuidado, pois aumentar muito a memória compartilhada (*shared buffers*) pode acarretar uso da memória virtual (swap). As alterações podem ser feitas através do comando *postmaster* na linha de comando ou através da configuração do valor do *shared_buffers* no arquivo *postgresql.conf*.

Tamanho do Cache

O tamanho do cache deve ser grande o suficiente para conseguir manipular as tabelas mais comumente acessadas, e pequeno o bastante para evitar atividades de *swap* *pagein*.

Exemplo:

Queremos X MB para memória compartilhada $(X / 8) * 1024 =$ Resultado a ser configurado em *shared_buffer*

Se X = 768 MB

$(768 / 8) * 1024$

Resultado a ser configurado em *shared_buffer* = 98304.

Memória Individual (Sort Memory)

Esta memória é utilizada em ordenações de registros, *merge join* e em operações de criação de índices. Ela pode ser configurada através do parâmetro *sort_mem* do arquivo *postgresql.conf*.