6) Monitorando as Atividades do Servidor do PostgreSQL

1 – Habilitando e monitorando o autovacuum
2 – Entendendo as estatísticas do PostgreSQL
3 – Rotina de reindexação
4 – Monitorando o espaço em disco e tamanho de objetos
5 – Monitorando o arquivo de registros (logs)

Estão disponíveis várias **ferramentas para monitorar** a atividade do banco de dados e analisar o desempenho. Não se deve desprezar os programas regulares de monitoramento do Unix, como **ps, top, iostat e vmstat**. Também, uma vez que tenha sido identificado um comando com baixo desempenho, podem ser necessárias outras investigações utilizando o comando *EXPLAIN* do PostgreSQL.

Das ferramentas citadas, apenas **iostat** não acompanha o Ubuntu.
Podemos instalar com:

sudo apt-get install sysstat

Para monitorar o PostgreSQL no Windows, como também no Linux, usar o PGAdmin, em Ferramentas – Status do Servidor.

1 – Habilitando e monitorando o autovacuum

A rotina de manutenção vacuum (limpeza do disco rígido) é algo tão importante, que passou a vir habilitado por default para rodar automaticamente na versão 8.3.

Na versão 8.2 para automatizar o vacuum habilitar os seguintes parâmetros no postgresq.conf:

autovacuum = on
stats_start_collector = on
stats_row_level = on

Na versão 8.3:
autovacuum = on
track_counts = on

Monitorando o autovacuum

No Linux, uma forma simples de monitorar o autovacuum é executando o comando:

ps ax | grep postgres (Linux XUbuntu 7.10 com PostgreSQL 8.3)

Com isso serão mostrados:

```
6881 ?        S      0:00 /usr/local/pgsql/bin/postmaster -D /var/lib/pgsql/data
6928 ?        Ss     0:00 postgres: writer process
6929 ?        Ss     0:00 postgres: wal writer process
6930 ?        Ss     0:00 postgres: autovacuum launcher process
6931 ?        Ss     0:00 postgres: stats collector process
```

Através dos logs (ao final) podemos ter acesso a mais informações sobre o autovacuum.

No Windows usar o PGAdmin.

2 – Entendendo as estatísticas do PostgreSQL

O coletor de estatísticas

O coletor de estatísticas do PostgreSQL é um subsistema de apoio a coleta e relatório de **informações sobre as atividades do servidor**. Atualmente, o coletor pode contar acessos a tabelas e índices em termos de blocos de disco e linhas individuais.

Configuração da coleta de estatísticas

Uma vez que a coleta de estatísticas adiciona alguma sobrecarga à execução das cconsultas, o sistema pode ser configurado para coletar informações, ou não. Isto é controlado por parâmetros de configuração, normalmente definidos no arquivo **`postgresql.conf`**

O parâmetro **track_counts** deve ser definido como true para que o coletor de estatísticas seja ativado. Esta é a definição padrão e recomendada, mas pode ser desativada se não houver interesse nas estatísticas e for desejado eliminar até a última gota de sobrecarga (Entretanto, provavelmente o ganho será pequeno e talvez não compense). Deve ser observado que esta opção não pode ser mudada enquanto o servidor está executando.

`track_activities` (`boolean`)

Habilita a coleta de informações das consultas em execução atualmente de cada seção, como também o tempo que cada consulta iniciou sua execução. Este parâmetro vem habilitado por default. Somente o superusuário e o usuário dono da sessão percebem essa informação e somente superusuários podem mudar essa configuração.

`track_counts` (`boolean`)

> Habilita a coleta de informações das atividades dos bancos. Habilitado por default, pois o processo do autovacuum requer este parâmetro. Somente superusuário pode alterar.

`update_process_title` (`boolean`)

Habilita a atualização dos títulos dos processos cada vez que uma nova consulta SQL é recebida pelo servidor. O título do processo é tipicamente visualizado pelo comando **ps,** ou se em windows, pelo **Process Explorer**
http://technet.microsoft.com/pt-br/sysinternals/bb896653(en-us).aspx

Somente superusuário pode alterar esta configuração.

Monitorando Estatísticas

`log_statement_stats` (`boolean`)
`log_parser_stats` (`boolean`)
`log_planner_stats` (`boolean`)
`log_executor_stats` (`boolean`)

Para cada consulta, escreve as estatísticas de desempenho do respectivo módulo para o log (registro) do servidor. Este é um instrumento rústico. `log_statement_stats` reporta todas as declarações de estatística, enquanto os demais reportam as estatísticas por módulo. `log_statement_stats` não pode ser habilitado para nenhum com a opção por módulo. Todos estes parâmetros vêm desabilitados por default e somente o superusuário pode alterar seus status (através do comando SET).

Mostra o PID de cada consulta e a própria consulta em execução atualmente:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Alerta: sempre ao trabalhar com estas funcionalidades e ainda outras é útil verificar a versão do PostgreSQL em uso para procurar a respectiva versão de documentação.

Utilização do comando EXPLAIN

EXPLAIN -- mostra o plano interno de execução de uma consulta.

Sintaxe:
EXPLAIN [ANALYZE] [VERBOSE] consulta

Este comando mostra o *plano de execução* gerado pelo planejador do PostgreSQL para a consulta fornecida. O plano de execução mostra como as tabelas referenciadas pelo comando serão varridas — por uma varredura seqüencial simples, varredura pelo índice, etc. — e, se forem referenciadas várias tabelas, quais algoritmos de junção serão utilizados para juntar as linhas requisitadas de cada uma das tabelas de entrada.

Usar o comando ANALYZE com o EXPLAIN para que assim o EXPLAIN tenha uma estimativa de custo mais realista.

Do que é mostrado, a parte mais importante é o custo estimado de execução do comando, que é a estimativa feita pelo planejador de quanto tempo vai demorar para executar o comando (medido em unidades de acesso às páginas do disco). Na verdade, são mostrados dois números: **o tempo inicial** para que a primeira linha possa ser retornada, e **o tempo total** para retornar todas as linhas. **Para a maior parte dos comandos o tempo total é o que importa**, mas em contextos como uma

sub-seleção no EXISTS, o planejador escolhe o menor tempo inicial em vez do menor tempo total (porque o executor vai parar após ter obtido uma linha). Além disso, se for limitado o número de linhas retornadas usando a cláusula LIMIT, o planejador efetua uma interpolação apropriada entre estes custos para estimar qual é realmente o plano de menor custo.

ANALYZE faz o comando ser realmente executado, e não apenas planejado. O tempo total decorrido gasto em cada nó do plano (em milissegundos) e o número total de linhas realmente retornadas são adicionados ao que é mostrado. Esta opção é útil para ver se as estimativas do planejador estão próximas da realidade.

Se for desejado utilizar EXPLAIN ANALYZE para um comando INSERT, UPDATE, DELETE ou EXECUTE sem deixar o comando afetar os dados, deve ser utilizado o seguinte procedimento (dentro de uma transação):

```
BEGIN;
EXPLAIN ANALYZE INSERT INTO clientes (cod, nome) VALUES (1, 'João');
ROLLBACK;
```

VERBOSE

Mostra a representação interna completa da árvore do plano, em vez de apenas um resumo. Geralmente esta opção é **útil apenas para finalidades especiais de depuração.**

Consulta - Qualquer comando SELECT, INSERT, UPDATE, DELETE, EXECUTE ou DECLARE, cujo plano de execução se deseja ver.

O PostgreSQL concebe um *plano de consulta* para cada consulta recebida. **A escolha do plano correto, correspondendo à estrutura do comando e às propriedades dos dados, é absolutamente crítico para o bom desempenho**. Pode ser utilizado o comando *EXPLAIN* para ver o plano criado pelo sistema para qualquer comando. A leitura do plano é uma arte que merece um estudo aprofundado. Aqui são fornecidas apenas algumas informações básicas.

Os números apresentados atualmente pelo EXPLAIN são:

- O **custo inicial** estimado (O tempo gasto para poder começar a varrer a saída como, por exemplo, o tempo para fazer a classificação em um único nó de classificação).
- O **custo total** estimado (Se todas as linhas fossem buscadas, o que pode não acontecer: uma consulta contendo a cláusula LIMIT pára antes de gastar o custo total, por exemplo).
- **Número de linhas (rows)** de saída estimado para este nó do plano (Novamente, somente se for executado até o fim).
- **Largura média estimada** (**width**) (em bytes) das linhas de saída deste nó (fase) do plano.

Os custos são medidos em termos de **unidades de páginas de disco** buscadas (O esforço de CPU estimado é convertido em unidades de páginas de disco, utilizando fatores estipulados altamente arbitrários.

Linhas de saída é **um pouco enganador**, porque *não* é o número de linhas processadas/varridas pelo comando, geralmente é menos, refletindo a seletividade estimada de todas as condições da cláusula WHERE aplicadas a este nó. **Idealmente, a estimativa de linhas do nível superior estará próxima do número de linhas realmente retornadas, atualizadas ou excluídas pela consulta**.

Antes da realização de testes, devemos executar o comando:
VACUUM ANALYZE; -- Para atualizar as estatísticas internas (do banco atual)

Exemplos:
Testes no banco cep_brasil, após adicionar a chave primária na tabela cep_full.

Observe a saída destes comandos no psql ou através da query tool do PGAdmin.

```
EXPLAIN SELECT * FROM cep_full; (varredura sequencial, já que não usamos o
índice e retornarão todos os registros).

EXPLAIN SELECT logradouro FROM cep_full WHERE cep = '60420440'; (varredura com o
índice, pois retornamos apenas parte dos registros com a cláusula where).
```

Uma forma de ver outros planos é forçar o planejador a não considerar a estratégia que sairia vencedora, ativando e desativando sinalizadores de cada tipo de plano (Esta é uma forma deselegante, mas útil):

```
SET enable_nestloop = off;
EXPLAIN SELECT * FROM cep_full t1, cep_full_index t2 WHERE t1.cep < '60000' AND
t1.cep = t2.cep;
```

Examinando Índices

É muito ruim usar conjuntos de registros muito pequenos. Enquanto que selecionando 1000 de um total de 100.000 registros pode ser um candidato para um índice, selecionando 1 de 100 deve dificilmente ser, porcausa de que 100 deve provavelmente caber em uma simples página de disco e não existe nenhum plano que pode ser sequencialmente coletado em uma única página de disco.

Também seja cuidadoso quando criando dados de teste, que não estarão disponíveis quando a aplicação estiver em produção. Valores que são muito similares, completamente aleatórios ou inseridos em alguma ordem devem atrapalhar as estatísticas diferente do que teriam uma distribuição real de dados.
Então uma cópia dos registros em produção deve ser mais adequada para testes.

Quando índices não são usados é útil fazer testes para forçar seu uso. Existem parâmetros para run-time que podem desativar varios tipos de planos. Por exemplo, desabilitando o sequential scan (enable_seqscan) com:

Via SQL:
set enable_setscan to off;

No postgresql.conf:
enable_seqscan = off

e joins em loops aninhados (enable_nestloop), que são os planos mais básicos, irá forçar o sistema a usar planos diferentes. Caso o sistema continue selecionando um sequential scan ou nested-loop join (join com loops aninhados) então esta é provavelmente a mais fundamental razão de que o índice não é usado; por exemplo, a condição da consulta não corresponde ao índice.

Detalhes em: http://www.postgresql.org/docs/8.3/interactive/runtime-config-query.html

Controle do planejador com cláusulas JOIN explícitas

É possível ter algum controle sobre o planejador de comandos utilizando a sintaxe JOIN explícita. Para saber por que isto tem importância, primeiro é necessário ter algum conhecimento.

Em uma consulta de junção simples, como

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

o planejador está livre para fazer a junção das tabelas em qualquer ordem. Por exemplo, pode gerar um plano de comando que faz a junção de A com B utilizando a condição a.id = b.id do WHERE e, depois, fazer a junção de C com a tabela juntada utilizando a outra condição do WHERE. Poderia, também, fazer a junção de B com C e depois juntar A ao resultado. Ou, também, fazer a junção de A com C e depois juntar B, mas isto não seria eficiente, uma vez que deveria ser produzido o produto Cartesiano completo de A com C, porque não existe nenhuma condição aplicável na cláusula WHERE que permita a otimização desta junção (Todas as junções no executor do PostgreSQL acontecem entre duas tabelas de entrada sendo, portanto, necessário construir o resultado a partir de uma ou outra destas formas). **O ponto a ser destacado é que estas diferentes possibilidades de junção produzem resultados semanticamente equivalentes, mas podem ter custos de execução muito diferentes. Portanto, o planejador explora todas as possibilidades tentando encontrar o plano de consulta mais eficiente.**

Mais detalhes em: http://pgdocptbr.sourceforge.net/pg80/explicit-joins.html e
http://pgdocptbr.sourceforge.net/pg80/catalog-pg-statistic.html
http://pgdocptbr.sourceforge.net/pg80/catalog-pg-class.html

Estatísticas utilizadas pelo planejador

Conforme visto na seção anterior, o planejador de comandos precisa estimar o número de linhas buscadas pelo comando para poder fazer boas escolhas dos planos de comando.

Um dos componentes da estatística é o número total de entradas em cada tabela e índice, assim como o número de blocos de disco ocupados por cada tabela e índice. Esta informação é mantida nas colunas reltuples e relpages da tabela pg_class, podendo ser vista utilizando consultas semelhantes à mostrada abaixo:

```
SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname LIKE
'cep%';
```

Em: http://pgdocptbr.sourceforge.net/pg80/planner-stats.html

Mais detalhes em: http://pgdocptbr.sourceforge.net/pg80/monitoring-stats.html, http://www.postgresql.org/docs/8.3/interactive/monitoring-stats.html e http://www.postgresql.org/docs/8.3/interactive/runtime-config-statistics.html#GUC-TRACK-ACTIVITIES.

Entendendo os Planos internos do PostgreSQL para a Execução das Consultas

Após o servidor do PostgreSQL receber uma consulta de uma aplicação cliente, o texto da consulta é entregue ao parser (analisador). O analisador procura por toda a consulta e checa por erros de sintaxe. Caso a consulta esteja sintaticamente correta o parser deve transformar o texto da consulta em estrutura de árvore analisada. Um parse tree (gráfico em forma de árvore analisada) é uma estrutura de dados que representa o significado de sua consulta em um formulário formal e sem ambiguidades.

Tendo como base esta consulta:

SELECT cpf_cliente, valor FROM pedidos WHERE valor > 0 ORDER BY valor;

Após o parser ter completado a análise da consulta, o parse tree é entregue ao planejador/otimizador.

O planejador é responsável por percorrer o parse tree e procurar todos os possíveis planos de execução da consulta. O plano deve incluir um ***sequential scan*** através de toda a tabela e ***index scan*** se índices úteis forem definidos. Caso a consulta envolva duas ou mais tabelas o planejador pode sugerir um número de diferentes métodos para unir (join) as tabelas. Os planos de execução são desenvolvidos em termos de operadores de consulta. Cada operador de consulta transforma um ou mais conjuntos de entrada (input set) em um intermediário result set.

O operador seq scan, transforma um input set (tabela física) em um result set, filtrando qualquer registro que não satisfaça a constraint da consulta.

O operador sort produz um result set por ordenar o input set de acordo com uma ou mais chaves de ordenação.

Internamente complexas consultas são quebradas em consultas simples.

Quando todos os possíveis planos de execução forem gerados, o otimizador procurará pelo plano mais econômico. Para cada plano é atribuído um custo de execução. As estimativas de custo são medidos em unidades de disco I/O. **Um operador que lê um bloco simples de 8.192 bytes (8KB) de um disco tem o custo de uma unidade**. **O tempo de CPU também é medido em unidades de disco I/O mas usualmente como uma fração. Por exemplo, o total de tempo de CPU requerido para processar um simples registro é assumido como sendo 1% do simples disco I/O**. Podemos ajustar muitas das estimativas de custo. Cada operador de consulta tem uma diferente estimativa de custo.

Por exemplo, **o custo de um sequential scan de uma tabela completa é computado como o número de 8K blocos na tabela mais alguma sobrecarga de CPU.**

Após escolher o mais econômico (aparentemente) plano de execução o executor de consulta inicia no começo do plano e pede para o operador mais acima para produzir o result set. Cada operador transforma seu input set em um result set. O input set deve provir de outro operador mais abaixo na árvore/tree. Quando o operador mais acima completa sua transformação os results são retornados para a aplicação cliente.

Resumo das Operações Internas na Execução de uma Consulta

Cliente solicita uma consulta ->
Analisador recebe o texto da consulta ->
Analisador procura na consulta por erros de sintaxe ->
Analisador transforma texto da consulta (correta) em estrutura de árvore analisada (parse tree) ->
O planejador percorre o parse tree a procura todos os possíveis planos de execução ->
Internamente complexas consultas são quebradas em consultas simples ->
Para cada plano é atribuído um custo de execução ->
Após todos os possíveis planos de execução serem gerados, o otimizador procurará pelo mais econômico ->
Após escolher o plano mais econômico o executor de consulta vai ao começo do plano e pede para o operador mais acima para produzir o result set ->
Cada operador transforma seu input set em um result set ->
O input set deve provir de outro operador mais abaixo na árvore/tree ->
Quando o operador mais acima completa sua transformação os results são retornados para a aplicação cliente.

O explain é utilizado somente para analisar as consultas com SELECT, INSERT, DELETE, UPDATE e DECLARE... CURSOR.

Exemplo Simples de Uso do Explain:

```
cep_brasil=# explain analyze select * from cep_full_index;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------
 Seq Scan on cep_full_index  (cost=0.00..31671.01 rows=633401 width=290)
                                                  (actual time=0.032..10863.173 rows=633401
loops=1)
 Total runtime: 12298.089 ms
(2 rows)
```

Vamos analisar o resultado.
Realmente o formato do plano de execução pode parecer um pouco misterioso no início.
Em cada passo no plano de execução o EXPLAIN mostra as seguintes informações:

- O tipo de operação requerida

- O custo estimado para a execução
- Caso se especifique EXPLAIN ANALYZE, a consulta será de fato executada e mostrado o custo atual.
Em se omitindo ANALYZE a consulta será apenas planejada e não executada e o custo atual não é mostrado.

- Tipo de operação - **Seq Scan**, foi o tipo de operação que o PostgreSQL elegeu para a consulta
- Custo estimado - **(cost=0.00..31671.01 rows=633401 width=290).** É composto de três partes: **cost=0.00..31671.01**, que nos informa quanto gastou a consulta. O gasto é medido em termos de leitura de disco. Dois números são mostrados:

0.00... – o primeiro nos informa o velocidade para que um registro do result set seja retornado pela operação.

31671.01 – o segundo, que normalmente é mais importante, representa o gasto para a operação completa.

rows=633401 – Este nos informa quantos registros o PostgreSQL espera que retorne na operação. Não é exatamente o número real de registros.

width=290 – Esta última parte do custo estimado. É uma estimativa do tamanho, em bytes, do registro médio, no result set. Para comprovar experimente mudar o exemplo para a tabela clientes do dba_projeto e veja como não bate.

Lembrando que os operadores de consulta...

Exemplo sem o Analyze:

```
cep_brasil=# explain select * from cep_full_index;
                          QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on cep_full_index  (cost=0.00..31671.01 rows=633401 width=290)
(1 row)
```

Outro Exemplo:

```
cep_brasil=# explain select * from cep_full;
                        QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on cep_full  (cost=0.00..31671.01 rows=633401 width=290)
(1 row)
```

Agora sem utilizar o Explain, ou seja, o atual (real):

```
\timing
select count(*) from cep_full_index;
 count
--------
 633401
Time: 10574,343 ms
```

Veja que o tempo é similar ao retornado no Exemplo Simples em: (actual time=0.032..10863.173.

Em consultas mais complexas o resultado é dividido em vários passos e os primeiros aparecem abaixo. Os últimos acima, como nestes exemplos:

Uma consulta contendo apenas 23 registros e prdenando por um campo que tem índice:

```
\c dba_projetodba_projeto=# explain analyze select * from clientes order by cpf;
                                      QUERY PLAN
---------------------------------------------------------------------------------------------------
 Sort  (cost=1.75..1.81 rows=23 width=64) (actual time=0.571..0.616 rows=23 loop
s=1)
   Sort Key: cpf
   Sort Method:  quicksort  Memory: 20kB
   ->  Seq Scan on clientes  (cost=0.00..1.23 rows=23 width=64) (actual time=0.0
11..0.137 rows=23 loops=1)
 Total runtime: 0.865 ms
```

Agora uma tabela com uma grande quantidade de registros (mais de meio milhão), mas sem um índice no campo pelo qual está ordenando:

```
\c cep_brasil
cep_brasil=# explain analyze select * from cep_full order by cep;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Sort  (cost=352509.44..354092.94 rows=633401 width=290) (actual time=33675.802.
.53168.029 rows=633401 loops=1)
```

```
  Sort Key: cep
  Sort Method:  external sort  Disk: 189376kB
  ->  Seq Scan on cep_full  (cost=0.00..31671.01 rows=633401 width=290) (actual
time=0.027..17997.280 rows=633401 loops=1)
Total runtime: 54728.569 ms
```

Em uma tabela com uma grande quantidade de registros (mais de meio milhão), mas agora com um índice no campo pelo qual está ordenando:

explain analyze select * from cep_full_index order by cep;

```
cep_brasil=# explain analyze select * from cep_full_index order by cep;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Index Scan using cep_pk on cep_full_index
(cost=0.00..44605.36 rows=633401 width=290)
(actual time=90.159..18145.128 rows=633401 loops=1)
 Total runtime: 19699.895 ms
```

Outro exemplo, agora usando a cláusula WHERE na PK:

```
cep_brasil=# explain select * from cep_full_index where cep='60420440';
                            QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using cep_pk on cep_full_index  (cost=0.00..8.35 rows=1 width=290)
   Index Cond: (cep = '60420440'::bpchar)
```

O Sec Scan ocorrerá primeiro e depois dele o Sort. ***Veja que quando temos um índice E uma grande quantidade de registros no campo pelo qual estamos ordenando, o planejador/otimizador do PostgreSQ decide usar o operador Index Scan.***

Após o Index Scan finalizar a construção do seu result set intermediário ele será então colocado no próximo passo do plano. O passo final deste plano será a operação Sort, que é requerida para satisfazer a cláusula order by. Este operador reordena o result set produzido pelo Sec Scan e retorna o final result set para a aplicação cliente. Observe qua quando o Index Scan é eleito, a ordenação é feita pelo próprio índice e não pelo Sort.

Alguns dos Operadores de Consultas

Seq Scan – é o mais básico operador de consultas. O Seq Scan trabalha assim:
- iniciando no começo da tabela e
- varrendo até o final da tabela
- para cada linha da tabela, Sec Scan testa a constraint da consulta (no caso, where). Caso a constraint seja satisfeita, as colunas requeridas são retornadas para o result set.

Registros Mortos

Uma tabela pode ainda conter os registros excluídos, que não mais são visíveis. O Sec Scan não inclui registros mortos no result set, mas ele precisa ler esses registros o que pode ser custoso em uma tabela com pesada atualização.

Sec Scan não lerá a tabela completa antes de retornar o primeiro registro.
Já o operador Sort lerá todos os registros do input set antes de retornar o primeiro registro.

O planejador/otimizador escolherá o Sec Scan quando nenhum índice possa ser usado para satisfazer a consulta ou quando a consulta retorna todos os registros (neste caso o índice não será usado).

Index Scan – funciona percorrendo uma estrutura índice. Caso especifiquemos um valor inicial para uma coluna índice, como por exemplo, WHERE codigo >= 100, o Index Scan deve iniciar pelo valor apropriado. Caso especifiquemos um valor final, como WHERE codigo <= 200, então o Index Scan deverá terminar tão logo encontre um valor maior que 200.

O Index Scan tem duas vantagens sobre o operador Sec Scan: primeiro, o Sec Scan precisa ler todos os registros da tabela e somente pode remover registros do result set avaliando a cláusula WHERE para cada registro. Index Scan não precisa ler todos os registros se indicarmos um valor inicial ou final. Segundo: o Sec Scan retorna os registros na ordem da tabela e não em uma sorteada ordem. O Index Scan deve retornar os registros na ordem do índice.

Nem todos os tipos de índices podem ser utilizados no Index Scan, somente: *B-Tree, R-Tree e GiST* podem mas o tipo *Hash* não pode.

O planejador/otimizador usa um operador Index Scan quando ele pode reduzir o tamanho do result set percorrendo a faixa de valores do índice ou
quando ele pode evitar uma ordenação porcauxa da implícita ordenação oferecida pelo índice.

Sort – impõe uma ordenação no result set. Podemos ajustar o PostgreSQL através do parâmetro de runtima *sort_men*. Caso o result set caiba no sort_men * 1024 bytes, o Sort será feito na memória, usando o algoritmo Qsort.
O Sort nunca reduz o tamanho do result set e também não remove registros ou campos.
Sort lerá todos os registros do input set antes de retornar o primeiro registro.
Obviamente um Sort pode ser utilizado pela cláusula ORDER BY.

Para os demais operadores veja este capítulo do Livro sugerido abaixo, que inclusive pode ser lido online: http://www.iphelp.ru/faq/15/ch04lev1sec3.html

Referências:
http://www.postgresql.org/docs/8.3/interactive/using-explain.html
http://www.postgresql.org/docs/8.3/interactive/sql-explain.html
http://www.postgresql.org/docs/8.3/interactive/row-estimation-examples.html

http://pgdocptbr.sourceforge.net/pg80/sql-explain.html
Livro: The comprehensive guide to building, programming, and administering PostgreSQL databases, Second Edition, Capítulo 4: Understanding How PostgreSQL Executes a Query, disponível como e-book em:
http://www.iphelp.ru/faq/15/ch04lev1sec3.html

Introdução ao otimizador de consultas do PostgreSQL

Walter Rodrigo de Sá Cruz Criado em Wed Aug 30 11:48:04 2006 em:
http://artigos.waltercruz.com/postgresql/otimizador

O caminho de uma consulta

Antes de entrarmos no assunto da otimização propriamente dito, precisamos rever qual é o caminho de uma consulta no banco de dados.

(A seção a seguir é adaptada da documentação do PostgreSQL)
O programa aplicativo transmite um comando para o servidor, e aguarda para receber de volta os resultados transmitidos pelo servidor.

1. O estágio de análise verifica o comando transmitido pelo programa aplicativo com relação à correção da sintaxe, e cria a árvore de comando.
2. O sistema de reescrita recebe a árvore de comando criada pelo estágio de análise, e procura por alguma regra(armazenada nos catálogos do sistema) a ser aplicada na árvore de comando. Realiza as transformações especificadas no corpo das regras. Uma das aplicações do sistema de reescrita é a criação de visões. Sempre que é executado um comando em uma visão (ou seja, uma tabela virtual), o sistema de reescrita reescreve o comando do usuário como um comando acessando as tabelas base especificadas na definição da visão, em vez da visão.
3. O planejador/otimizador recebe a árvore de comando (reescrita), e cria o plano de comando que será a entrada do executor. Isto é feito criando primeiro todos os caminhos possíveis que levam ao mesmo resultado. Por exemplo, se existe um índice em uma relação a ser varrido, existem dois caminhos para a varredura. Uma possibilidade é uma varredura seqüencial simples, e a outra possibilidade é utilizar o índice. Em seguida é estimado o custo de execução de cada umdos caminhos, e escolhido o mais barato. O caminho mais barato é expandido em um plano completo para que o executor possa utilizá-lo.
4. O executor caminha recursivamente através da árvore do plano, e traz as linhas no caminho representado pelo plano. O executor faz uso do sistema de armazenamento ao varrer as relações, realiza classificações e junções, avalia as qualificações e, por fim, envia de volta as linhas derivadas.

O papel do otimizador

(Adaptado de http://en.wikipedia.org/wiki/Query_optimizer)

O otimizador de consultas é o componente do banco de dados que tenta determinar o o modo mais eficiente de executar uma consulta.

O PostgreSQL usa algumas estatísticas mantidas em tabelas do sistema para direcionar o otimizador. Se essas estatísticas estiverem desatualizadas, você provavelmente não obterá o melhor plano para a consulta.

Para atualizar as estatísticas, devemos executar o comando ANALYZE.

É possível vermos o plano que o otimizador de consultas do PostgreSQL gerou, usando a cláusula EXPLAIN antes da consulta.

Adicionando a cláusula EXPLAIN ANALYZE antes da consulta, ela é de fato executada, de forma que possamos ter a medida do tempo.

Exemplos

Pagila

Vamos usar o banco de dados exemplo pagila, disponível em: http://pgfoundry.org/projects/dbsamples/, que é um exemplo de banco de dados de uma locadora.

Vamos começar com a configuração padrão do otimizador do Postgres. Todas as variáveis no postgresql.conf estão comentadas, o que significa que elas usarão os valores padrão.

Como teste, criei dois bancos iguais, o pagila e o pagila2, com a diferença que não rodei o ANALYZE no pagila2.

A configuração do banco é a padrão do Debian.

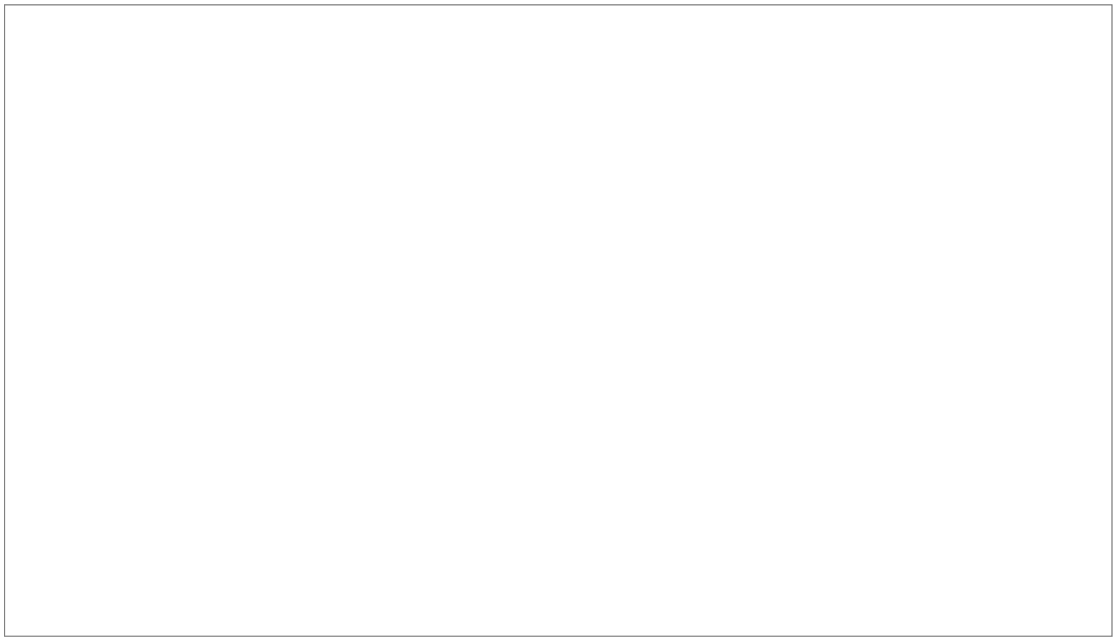
A consulta a seguir retorna os filmes e os atores associados a ela:

```
SELECT f.title, a.first_name FROM film f
INNER JOIN  film_actor ON film_actor.film_id=f.film_id
INNER JOIN actor a on film_actor.actor_id = a.actor_id
```

Aqui está o plano gerado para essa consulta no pagila2 (o que não recebeu ainda o ANALYZE):

```
Merge Join  (cost=708.95..795.88 rows=5462 width=185) (actual
time=56.274..85.883 rows=5462 loops=1)
```

```
  Merge Cond: ("outer".film_id = "inner".film_id)
  ->  Sort  (cost=94.83..97.33 rows=1000 width=149) (actual time=8.489..9.926
rows=1000 loops=1)
        Sort Key: f.film_id
        ->  Seq Scan on film f  (cost=0.00..45.00 rows=1000 width=149) (actual
time=0.010..4.517 rows=1000 loops=1)
  ->  Sort  (cost=614.12..627.77 rows=5462 width=42) (actual time=47.775..56.152
rows=5462 loops=1)
        Sort Key: film_actor.film_id
        ->  Merge Join  (cost=0.00..275.06 rows=5462 width=42) (actual
time=0.027..33.019 rows=5462 loops=1)
              Merge Cond: ("outer".actor_id = "inner".actor_id)
              ->  Index Scan using actor_pkey on actor a  (cost=0.00..12.20
rows=200 width=44) (actual time=0.010..0.450 rows=200 loops=1)
              ->  Index Scan using film_actor_pkey on film_actor
(cost=0.00..194.08 rows=5462 width=4) (actual time=0.006..13.315 rows=5462
loops=1)
Total runtime: 94.576 ms
```



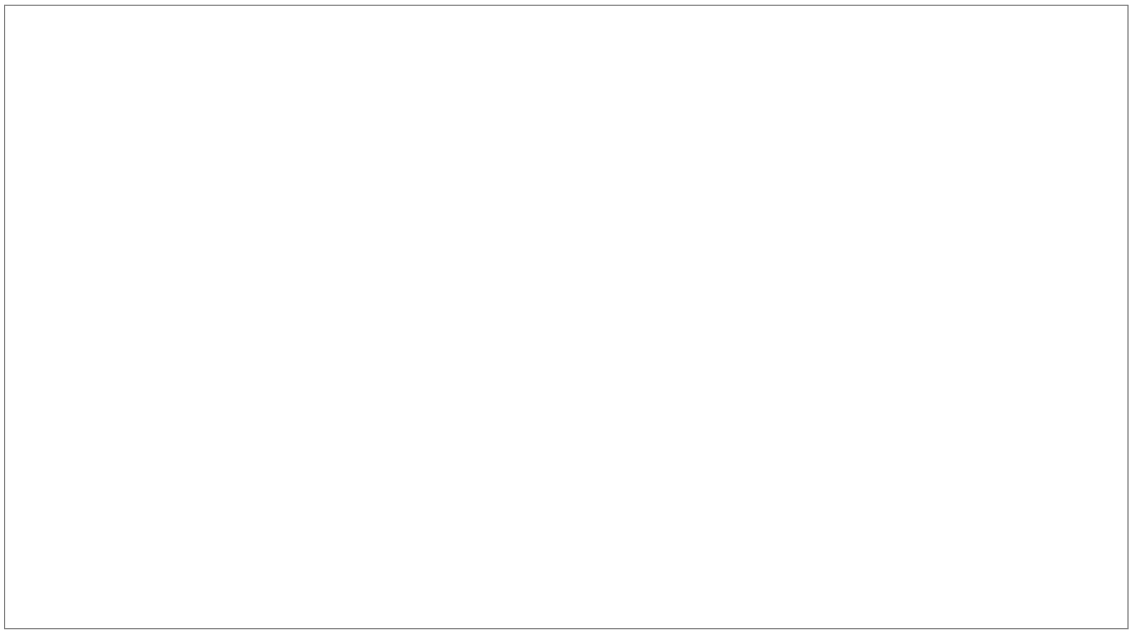
Sem entrar nos detalhes, vamos entender esse plano. Primeiro, temos que saber olhar de baixo pra cima.

1. Foi feito um index scan na tabela film_actor (a última linha)
2. Foi feito um index table scan na tabela actor
3. Essas duas tabelas foram unidas usando o algoritmo Merge Join
4. O resultado foi ordenado pela coluna fil,_actor.film_id
5. Foi feito um table scan na tabela film, e o resultado foi ordenado pela coluna film id
6. O resultado do scan na tabela film foi juntado com o resultado anterior, usando o algoritmo Merge Join

E aqui está o plano gerado pelo pagila (o banco no qual foi rodado o ANALYZE):

```
Merge Join  (cost=562.49..697.92 rows=5462 width=27) (actual time=47.691..78.064
```

```
rows=5462 loops=1)
  Merge Cond: ("outer".film_id = "inner".film_id)
  ->  Index Scan using film_pkey on film f  (cost=0.00..51.00 rows=1000
width=22) (actual time=0.011..2.442 rows=1000 loops=1)
  ->  Sort  (cost=562.49..576.15 rows=5462 width=11) (actual time=47.668..56.011
rows=5462 loops=1)
        Sort Key: film_actor.film_id
        ->  Merge Join  (cost=0.00..223.43 rows=5462 width=11) (actual
time=0.025..32.304 rows=5462 loops=1)
              Merge Cond: ("outer".actor_id = "inner".actor_id)
              ->  Index Scan using actor_pkey on actor a  (cost=0.00..6.20
rows=200 width=13) (actual time=0.007..0.459 rows=200 loops=1)
              ->  Index Scan using film_actor_pkey on film_actor
(cost=0.00..148.46 rows=5462 width=4) (actual time=0.008..12.557 rows=5462
loops=1)
Total runtime: 86.610 ms
```



Vamos agora analizar esse plano:

1. Foi feito um scan na tabela film_actor pelo índice film_actor_pkey (a chave primária da tabela)
2. Agora, um scan na tabela actor pelo índice
3. As tabelas foram JOINadas pelo actor_id, e o resultado foi ordenado por film_actor.actor_id
4. A tabela film foi varrida pelo índice.
5. O resultado do JOIN anterior foi juntado com a tabela film.

Procurando clientes e filiais:

```
SELECT 'Filial ' || filial.store_id as filial, c.first_name || c.last_name as
client FROM customer c
INNER JOIN store filial on filial.store_id = c.store_id WHERE c.active = 1
```

Aqui, o explain do banco sem ANALYZE:

```
Nested Loop  (cost=1.02..17.65 rows=1 width=84) (actual time=0.042..15.491
rows=584 loops=1)
  Join Filter: ("inner".store_id = "outer".store_id)
  ->  Seq Scan on customer c  (cost=0.00..16.49 rows=3 width=82) (actual
time=0.013..1.621 rows=584 loops=1)
        Filter: (active = 1)
  ->  Materialize  (cost=1.02..1.04 rows=2 width=4) (actual time=0.002..0.006
rows=2 loops=584)
        ->  Seq Scan on store filial  (cost=0.00..1.02 rows=2 width=4) (actual
time=0.004..0.011 rows=2 loops=1)
Total runtime: 16.480 ms
```

Repare que foi usado um materialize = um plano foi salvo num arquivo temporário.

E agora, o explain do banco com ANALYZE:

```
Nested Loop  (cost=4.05..46.50 rows=584 width=23) (actual time=0.175..6.097
rows=584 loops=1)
  ->  Seq Scan on store filial  (cost=0.00..1.02 rows=2 width=4) (actual
time=0.008..0.015 rows=2 loops=1)
  ->  Bitmap Heap Scan on customer c  (cost=4.05..16.80 rows=300 width=21)
(actual time=0.136..1.248 rows=292 loops=2)
        Recheck Cond: ("outer".store_id = c.store_id)
        Filter: (active = 1)
        ->  Bitmap Index Scan on idx_fk_store_id  (cost=0.00..4.05 rows=300
width=0) (actual time=0.122..0.122 rows=300 loops=2)
              Index Cond: ("outer".store_id = c.store_id)
Total runtime: 7.160 ms
```

Nesse caso, como as estatísticas já estavam atualizadas, o banco optou por uma otimização e usou o Bitmap Index Scan e o Bitmap Heap Scan.

O Bitmap Index Scan é usado em colunas que tem pouca variação (colunas de baixa cardinalidade). No caso, para cada cliente, eu tenho apenas duas opções de filias às quais ele pode estar associado (1 e 2). Então, o banco cria um mapa de bits para cada um desses valores. Esse mapa é carregado em memória, e é juntado com o resultado do table scan em store, que tem apenas dois valores possíveis.

No último select, ainda assim foi feito um Table Scan na tabela store, onde alguém poderia argumentar que seria melhor um index scan.

Random Page Cost

Existe um parâmetro de configuração, chamado **random_page_cost** que determina qual o peso que o PostgreSQL dá a leituras não sequencias no disco. Aumentar esse valor favorece o uso de table scans, abaixá-lo favorece o uso de índices. O valor padrão é 4.0.

Para um pequeno teste, vamos baixá-lo para 2.0 e executar a query no banco pagila.

```
Merge Join  (cost=0.00..42.34 rows=584 width=23) (actual time=0.128..6.012
rows=584 loops=1)
```

```
  Merge Cond: ("outer".store_id = "inner".store_id)
  ->  Index Scan using store_pkey on store filial  (cost=0.00..3.02 rows=2
width=4) (actual time=0.092..0.098 rows=2 loops=1)
  ->  Index Scan using idx_fk_store_id on customer c  (cost=0.00..27.64 rows=584
width=21) (actual time=0.015..2.150 rows=584 loops=1)
        Filter: (active = 1)
Total runtime: 6.996 ms
```

Como o custo do índice estava mais baixo, o otimizador preferiu usar o índice do que gerar o índice bitmap em memória.

Agora, no pagila2, o plano gerado é semelhante ao plano do banco que está com as estatísticas atualizadas:

```
Nested Loop  (cost=2.01..12.74 rows=1 width=84) (actual time=0.264..5.829
rows=584 loops=1)
  ->  Seq Scan on store filial  (cost=0.00..1.02 rows=2 width=4) (actual
time=0.012..0.019 rows=2 loops=1)
  ->  Bitmap Heap Scan on customer c  (cost=2.01..5.82 rows=3 width=82) (actual
time=0.182..1.236 rows=292 loops=2)
        Recheck Cond: ("outer".store_id = c.store_id)
        Filter: (active = 1)
        ->  Bitmap Index Scan on idx_fk_store_id  (cost=0.00..2.01 rows=3
width=0) (actual time=0.165..0.165 rows=300 loops=2)
              Index Cond: ("outer".store_id = c.store_id)
Total runtime: 6.861 ms
```

Executada no pagila2, a query gera o mesmo plano que havia sido gerado para random_page_cost = 4.0. E, com random_page_cost = 2.0, a primeira query do nosso teste passa a usar o índice, mesmo sem o analyze. Mas os custos não são os mesmos - isso porque o pagila2 ainda não teve as estatísticas atualizadas.

Configurações do otimizador

O arquivo de configuração do postgresql (postgresql.conf) contém várias opções que tratam da configuração do otimizador e dos vários métodos de consulta. São eles:

enable_bitmapscan
enable_hashagg
enable_hashjoin
enable_indexscan
enable_mergejoin

enable_nestloop
enable_seqscan
enable_sort
enable_tidscan

Todos eles podem ser configurados para on ou off, e por padrão todos vem habilitados. Alguns pontos a serem notados:

enable_seqscan = off não desabilita de fato o scan sequencial, já que essa pode ser a única forma de executar certas consultas. O que esse parâmetro faz ne verdade é aumentar o custo de um table scan. O mesmo vale para enable_nestloop (algumas vezes não há forma de resolver uma consulta, exceto por esse tipo de loop) e enable_sort.

Esses valores podem ser alterados em tempo de execução para uma sessão em particular, usando o comando SET.

Esse é apenas um resumo. As descrição completa das opções relativas ao otimizador são encontradas em: http://www.postgresql.org/docs/8.0/static/runtime-config.html .

Listando atores e filmes

A primeira consulta, embora trouxesse a lista de filmes e atoes, não trazia o resultado agrupado pelos filmes, com uma lista de todos os atores.

Vamor fazer essa consulta então.

Existem duas sintaxes que podem ser usadas. A primeira:

```
select
     film.title AS title,
     array_to_string(array_accum(actor.first_name || ' ' ||
actor.last_name),',') AS actors
from
     film
     inner join film_actor on film.film_id = film_actor.film_id
     inner join actor on film_actor.actor_id = actor.actor_id
GROUP BY film.title
ORDER BY film.title
```

E seu explain:

```
Sort  (cost=768.81..771.31 rows=1000 width=37) (actual time=151.996..153.650
rows=997 loops=1)
  Sort Key: film.title
  ->  HashAggregate  (cost=698.98..718.98 rows=1000 width=37) (actual
time=132.534..142.902 rows=997 loops=1)
```

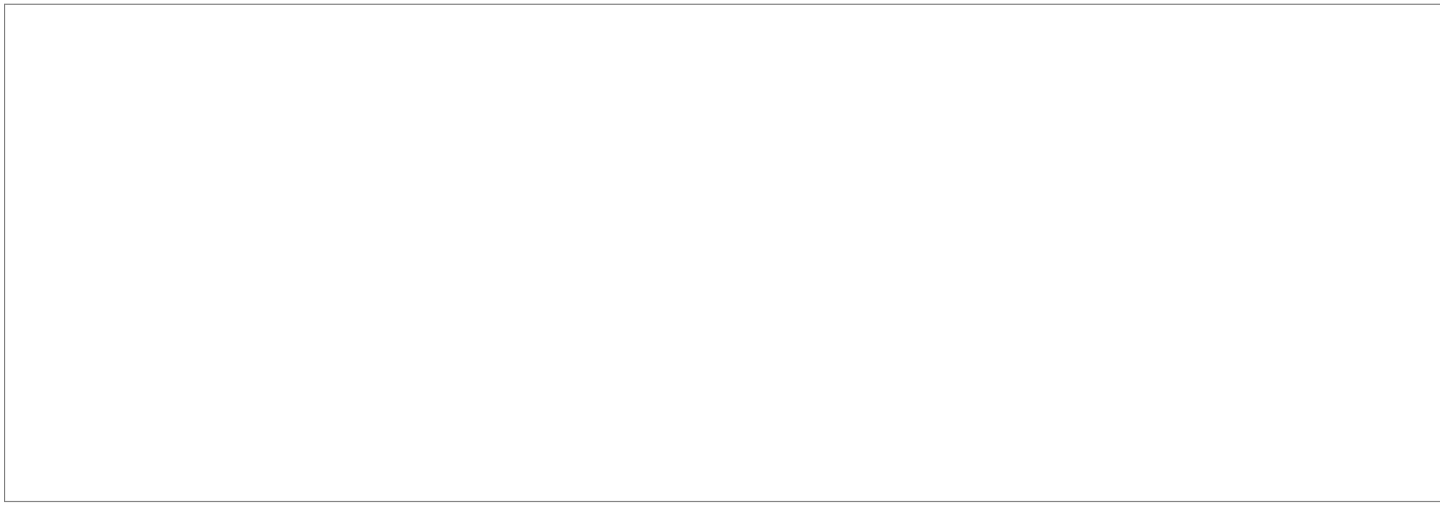
```
        ->  Merge Join  (cost=536.24..671.67 rows=5462 width=37) (actual
time=53.739..97.676 rows=5462 loops=1)
              Merge Cond: ("outer".film_id = "inner".film_id)
              ->  Index Scan using film_pkey on film  (cost=0.00..51.00
rows=1000 width=22) (actual time=0.012..3.780 rows=1000 loops=1)
              ->  Sort  (cost=536.24..549.90 rows=5462 width=21) (actual
time=53.716..63.214 rows=5462 loops=1)
                    Sort Key: film_actor.film_id
                    ->  Merge Join  (cost=0.00..197.18 rows=5462 width=21)
(actual time=0.026..36.636 rows=5462 loops=1)
                          Merge Cond: ("outer".actor_id = "inner".actor_id)
                          ->  Index Scan using actor_pkey on actor
(cost=0.00..6.20 rows=200 width=23) (actual time=0.007..0.536 rows=200 loops=1)
                          ->  Index Scan using film_actor_pkey on film_actor
(cost=0.00..122.21 rows=5462 width=4) (actual time=0.009..14.883 rows=5462
loops=1)
Total runtime: 159.766 ms
```

Uma outra forma possível de fazer essa consulta é essa:

```
SELECT f.title,
array_to_string(array(select first_name FROM film_actor fa
join actor a ON fa.actor_id = a.actor_id and fa.film_id = f.film_id
),', ') as film_actor
FROM film f
```

Será que essa forma, além de ser menos compacta, é mais rápida também?

```
Seq Scan on film f  (cost=0.00..16382.69 rows=1000 width=22) (actual
time=1.214..712.919 rows=1000 loops=1)
  SubPlan
    ->  Merge Join  (cost=9.56..16.34 rows=5 width=9) (actual time=0.199..0.687
rows=5 loops=1000)
          Merge Cond: ("outer".actor_id = "inner".actor_id)
          ->  Index Scan using actor_pkey on actor a  (cost=0.00..6.20 rows=200
width=13) (actual time=0.005..0.347 rows=165 loops=1000)
          ->  Sort  (cost=9.56..9.57 rows=5 width=2) (actual time=0.056..0.069
rows=5 loops=1000)
                Sort Key: fa.actor_id
                ->  Bitmap Heap Scan on film_actor fa  (cost=2.02..9.50 rows=5
width=2) (actual time=0.020..0.033 rows=5 loops=1000)
                      Recheck Cond: (film_id = $0)
                      ->  Bitmap Index Scan on idx_fk_film_id  (cost=0.00..2.02
rows=5 width=0) (actual time=0.013..0.013 rows=5 loops=1000)
                            Index Cond: (film_id = $0)
Total runtime: 714.655 ms
```



Parece que não.

Vemos que na consulta, o maior tempo gasto está no merge join (de 0.199 até 0.687). Vamos desabilitá-lo então. Isso é feito com o comando: set enable_mergejoin = off.

Vejamos o novo plano:

```
Seq Scan on film f  (cost=0.00..24679.64 rows=1000 width=22) (actual
time=0.368..162.552 rows=1000 loops=1)
  SubPlan
    ->  Nested Loop  (cost=2.02..24.63 rows=5 width=9) (actual time=0.037..0.136
rows=5 loops=1000)
          ->  Bitmap Heap Scan on film_actor fa  (cost=2.02..9.50 rows=5
width=2) (actual time=0.020..0.039 rows=5 loops=1000)
                Recheck Cond: (film_id = $0)
                ->  Bitmap Index Scan on idx_fk_film_id  (cost=0.00..2.02 rows=5
width=0) (actual time=0.014..0.014 rows=5 loops=1000)
                      Index Cond: (film_id = $0)
          ->  Index Scan using actor_pkey on actor a  (cost=0.00..3.01 rows=1
width=13) (actual time=0.007..0.009 rows=1 loops=5462)
                Index Cond: ("outer".actor_id = a.actor_id)
Total runtime: 164.212 ms
```

Fiz um pequeno teste com o beta do PostgreSQL 8.2, e para minha surpresa, o plano gerado para essa consulta é o mesmo que é gerado no 8.1 usando enable_mergejoin = off (com random_page_cost = 2.0).

Entendendo os ícones usados no pgadmin

Escrevi uma pequena descrição para os ícones usados no pgadmin, que está disponível em: http://artigos.waltercruz.com/postgresql/explain/

Outra coisa interessante pra se notar é que a grossura das setas é proporcional ao custo das operações.

3 – Rotina de reindexação

Em algumas situações vale a pena reconstruir índices periodicamente utilizando o comando REINDEX. Existe, também, o aplicativo `contrib/reindexdb` que pode reindexar todo o banco de dados. **Entretanto, a partir da versão 7.4 o PostgreSQL reduziu de forma substancial a necessidade desta atividade se comparado às versões anteriores.**

REINDEX -- reconstrói índices

Sintaxe

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } nome [ FORCE ]
```

Exemplos
Reconstruir um único índice:

```
REINDEX INDEX meu_indice;
```

Reconstruir os índices da tabela minha_tabela:

```
REINDEX TABLE minha_tabela;
```

Reconstruir todos os índices de um determinado banco de dados, sem confiar que os índices do sistema estejam válidos:

```
$ export PGOPTIONS="-P"
$ psql bd_danificado
...
bd_danificado=> REINDEX DATABASE bd_danificado;
bd_danificado=> \q
```

Compatibilidade
Não existe o comando REINDEX no padrão SQL.

Notas

[1] Oracle — O comando ALTER INDEX é utilizado para alterar ou reconstruir um índice existente. A cláusula de reconstrução (rebuild_clause) é utilizada para recriar um índice existente ou uma de suas partições ou sub-partições. Se o índice estiver marcado como UNUSABLE, uma reconstrução bem-sucedida irá marcá-lo como USABLE. Para um índice baseado em função, esta cláusula também ativa o índice. Se a função sobre a qual o índice se baseia não existir, o comando de reconstrução irá falhar. Oracle® Database SQL Reference 10g Release 1 (10.1) Part Number B10759-01 (N. do T.)

[2] SQL Server — O comando ALTER INDEX modifica um índice de visão ou de tabela desativando, reconstruindo ou reorganizando o índice; ou definindo opções para o índice. *Reconstruindo índices* — A reconstrução do índice remove e recria o índice. Esta operação remove a fragmentação, recupera espaço em disco compactando as páginas com base na definição do fator de preenchimento especificado ou existente, e reordena as linhas do índice em páginas contíguas. Quando é especificada a opção ALL, todos os índices da tabela não removidos e reconstruídos em uma única transação. As restrições FOREIGN KEY não precisam ser previamente removidas. *Reorganizando índices* — A reorganização do índice utiliza recursos do sistema mínimos. Esta operação defragmenta o nível-folha (*leaf level*) dos índices agrupados (*clustered*) e não agrupados (*nonclustered*) de tabelas e visões, reordenando fisicamente as páginas do nível-folha para corresponderem a ordem lógica, esquerda para direita, dos nós folha. A reorganização também compacta as páginas de índice. A compactação é baseada no valor do fator de preenchimento existente. SQL Server 2005 Books Online — ALTER INDEX (Transact-SQL) (N. do T.)

Existe também o contrib reindexdb, que reindexa todo um banco de dados:

Exemplos:

Para reindexar todos os índices do banco de dados teste:

```
$ reindexdb -U postgres teste
```

Para reindexar o índice clientes_idx da tabela clientes do banco de dados dba_projeto:

```
$ reindexdb -U postgres --table clientes --index clientes_idx dba_projeto
```

Mais detalhes em: http://pgdocptbr.sourceforge.net/pg82/sql-reindex.html

4 – Monitorando o espaço em disco e tamanho de objetos

Cada tabela possui um arquivo em disco, onde a maior parte dos dados são armazenados. Se a tabela possuir alguma coluna com valor potencialmente longo, também existirá um arquivo TOAST associado à tabela, utilizado para armazenar os valores muito longos para caber confortavelmente na tabela principal. Haverá um índice para a tabela TOAST, caso esta esteja presente. Também podem haver índices associados à tabela base. **Cada tabela e índice é armazenado em um arquivo em disco separado — possivelmente mais de um arquivo, se o arquivo exceder um gigabyte.**

O espaço em disco pode ser monitorado a partir de três lugares: do psql utilizando as informações do VACUUM, do psql utilizando as ferramentas presentes em `contrib/dbsize`, e da linha de comando utilizando as ferramentas presentes em `contrib/oid2name`. Utilizando o psql em um banco de dados onde o comando VACUUM ou ANALYZE foi executado recentemente, podem ser efetuadas consultas para ver a utilização do espaço em disco de qualquer tabela:

```
\c cep_brasil
VACUUM ANALYZE;

SELECT relname, relfilenode, relpages
FROM pg_class
WHERE relname LIKE 'cep%'
ORDER BY relname;

Ou:
SELECT relfilenode, relpages FROM pg_class WHERE relname = 'cep_full_index';

Ou:
SELECT relfilenode as tabela_inode, relpages*8*1024/(1024*1024) AS "Espaço em
MB" FROM pg_class WHERE relname = 'cep_full_index';

Ou:
SELECT relfilenode as tabela_inode, relpages*8/1024 AS "Espaço em MB" FROM
pg_class WHERE relname = 'cep_full_index';
```

Cada página possui, normalmente, 8 kilobytes (Lembre-se, relpages somente é atualizado por VACUUM, ANALYZE e uns poucos comandos de DDL como CREATE INDEX). O valor de relfilenode possui interesse caso se deseje examinar diretamente o arquivo em disco da tabela.

Para ver o espaço utilizado pelas tabelas TOAST deve ser utilizada uma consulta como a mostrada abaixo:

```
-- Ver as tabelas TOAST da tabela pg_rewrite
SELECT relname, relpages
    FROM pg_class,
         (SELECT reltoastrelid FROM pg_class
          WHERE relname = 'pg_rewrite') ss
    WHERE oid = ss.reltoastrelid
       OR oid = (SELECT reltoastidxid FROM pg_class
                 WHERE oid = ss.reltoastrelid)
    ORDER BY relname;
```

Os tamanhos dos índices também podem ser facilmente exibidos:

```
SELECT c2.relname, c2.relpages
    FROM pg_class c, pg_class c2, pg_index i
    WHERE c.relname = 'cep_full_index'
        AND c.oid = i.indrelid
        AND c2.oid = i.indexrelid
    ORDER BY c2.relname;
```

Adaptação de: http://pgdocptbr.sourceforge.net/pg80/diskusage.html

Falha de disco cheio

A tarefa mais importante de monitoramento de disco do administrador de banco de dados é garantir que o disco não vai ficar cheio. Um disco de dados cheio não resulta em corrupção dos dados, mas pode impedir que ocorram atividades úteis. Se o disco que contém os arquivos do WAL ficar cheio, pode acontecer do servidor de banco de dados entrar na condição de pânico e encerrar a execução.

Se não for possível liberar espaço adicional no disco removendo outros arquivos, podem ser movidos alguns arquivos do banco de dados para outros sistemas de arquivos utilizando espaços de tabelas (tablespaces).

> **Dica:** Alguns sistemas de arquivos têm desempenho degradado quando estão praticamente cheios, portanto não se deve esperar o disco ficar cheio para agir.

Se o sistema possuir cotas de disco por usuário, então naturalmente o banco de dados estará sujeito a cota atribuída para o usuário sob o qual o sistema de banco de dados executa. Exceder a cota ocasiona os mesmos malefícios de ficar totalmente sem espaço.
De: http://pgdocptbr.sourceforge.net/pg80/disk-full.html

5 – Monitorando o arquivo de registros (logs)

Estes arquivos guardam informações detalhadas de todas as operações do PostgreSQL, tanto as operações bem sucedidas quanto às más sucedidas, erros e alertas.

São arquivos com nomes no seguinte formato:

postgresql-2008-03-16_072122.log

postgresql – data – horadaprimeiraocorrência.log

E internamente exibe os registros em linhas assim:
2008-03-16 08:56:46 GMT ERROR: relation "wlw" does not exist
2008-03-16 08:56:46 GMT STATEMENT: select * from wlw;

Uma linha para o erro e a seguinte para a entrada que causou o erro.

Recomenda-se que o DBA mantenha sempre um olho nos logs.

Cada vez que o PostgreSQL é iniciado é criado um arquivo de log. Neste arquivo são registradas todas as ocorrências a partir daí e criado um novo arquivo na próxima vez que o servidor iniciar ou quando chegar a hora de rolar (configurado no postgresql.conf logrotale).

No postgresql.conf estão as regras:
#log_rotation_age = 1d
#log_rotation_size = 10MB

A cada dia ou quando chegar a 10MB um novo arquivo será criado.

Fique atento também para o volume que os logs ocupam em disco, pois podem comprometer o servidor.

No Windows localizan-se em:
C:\Program Files\PostgreSQL\8.3\data\pg_log

No Linux varia de acordo com a distribuição ou forma de instalação do PostgreSQL.

No postgresql.conf estão parâmetros que controlam e configuram os logs:

#log_directory = 'pg_log'

#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'

E muitas outras configurações dos logs.

É uma boa prática, periodicamente, remover os arquivos mais antigos. Se será feito backup dos mesmos ou não, depende da importância dos mesmos.

O diretório pg_xlog contém os arquivos de logs das transações. Tamanhos fixos de 16MB.

O diretório pg_clog guarda o status das transações.

pg_subtrans - status das subtransações

pg_tblspc - links simbolicos de tablespace

Ativando os Logs no PostgreSQL 8.3 do Linux Ubuntu 7.10

Editar o script postgresql.conf
sudo gedit /etc/postgresql/8.3/main/postgresql.conf

E alterar a linha:
#logging_collector = off
Copiando-a para uma linha abaixo:
logging_collector = on

Restartar o servidor:
sudo /etc/init.d/postgresql-8.3 restart

Depois disso podemos acessar os logs como usuário postgres:
su – postgres
cd /var/lib/postgresql/8.3/main/pg_log

Sobre as configurações dos logs veja:
http://pgdocptbr.sourceforge.net/pg80/runtime-config.html#RUNTIME-CONFIG-LOGGING-WHERE