10) Melhorando a Performance do PostgreSQL

O Que O Que Esperar do Tunning do SGBD

Tunning de banco de dados não é nenhuma panacéia.

Antes que o Tunning do SGBD possa trazer resultados concretos, é necessário que haja cuidado na:
Modelagem conceitual e física das tabelas
Escrita dos comandos SQL enviados pelas aplicações
Definição de índices, clustering de tabelas, etc
Dimensionamento da rede de comunicação entre clientes e o servidor

Metodologia para Tunning

Defina requisitos de desempenho
Isole medições sobre o SO, SGBD, rede e clientes
Realize bechmarks em um sistema sem carga, para obter um parâmetro de "melhor possível"
Realize benchmarks continuamente, para medir a melhora (ou piorar) a cada etapa de tunning
Pare quanto atingir os requisitos!

Tunning da Aplicação

Modelagem física
Otimização de comandos SQL

Modelagem Física

Utilize tabelas em CLUSTER caso sejam frequentemente acessadas segundo uma única ordenação
Evite a denormalização de dados; em geral, o custo de manutenção da redundância (fora o risco de perda da integridade dos dados) não compensa os pequenos ganhos de performance obtidos
Utilize bancos replicados para aplicações com padrões de acesso muito diferenciados, ex: aplicação OLAP de linha-de-negócios e aplicação OLTP, gerencial

Otimização Otimização de Comandos SQL

Utilize o comando EXPLAIN para visualizar o plano de acesso dos comandos SQL utilizados pela aplicação
Experimente criar índices para otimizar junções, filtros, agrupamento ou ordenação, e verifique os resultados com o comando EXPLAIN
Execute o comando VACUUM ANALYSE regularmente, para atualizar as estatísticas de tabelas e índices
Verifique periodicamente mudanças nos planos de acesso – pode ser necessario mudar os índices!

Arquitetura do PostgreSQL

Processos
Memória
Disco

Processos Processos do PostgreSQL

Um único processo postmaster escuta por conexões TCP ou Unix Sockets, e inicia um processo postgres para cada conexão
Assim sendo, o PostgreSQL utiliza naturalmente múltiplos processadores, mas para usuários simultâneos, não para acelerar um único comando SQL
Dois processos postgres estão sempre ativos, masnão atendem a conexões. Eles cuidam da gravação física de blocos de log ou tabelas, e da manutenção de estatísticas

Memória Memória e o PostgreSQL

Todos os processos postgres compartilhas duas áreas de memória:
O buffer cache armazena blocos lidos (ou modificados) de tabelas e índices
O write-ahead log armazena temporariamente o log de transações, até que ele possa ser armazenado em disco
Além disso, cada processo postgres tem uma área individual para operações de ordenação (sort buffer)

Disco e o PostgreSQL

Bancos de dados são nada mais do que diretórios
Tabelas e indices são arquivos dentro destes diretórios
Um diretório em separado armazena um ou mais segmentos de log de transações
Não há no PostgreSQL estruturas similares a tablespaces, log groups, etc
Conta-se com a capacidade do SO em alocar espaço em disco de forma eficiente, e com espelhamento pelo HW

Tunning do SO

O que medir

Como medir

Tunning do SO

Um SGBD, como qualquer outra aplicação, utiliza processos, memória, disco e conexões de rede
Portanto, obtenha do seu SO medidas de cada uma dessas áreas!
Está havendo muito swap?
A fila de requisições ao disco está crescendo?
A fila de requisições à placa de rede está crescendo?
O processador está ocioso?

Tunning do SO

Muitos contadores do Monitor de Performance do Windows NT/2000 apresentam valores incorretos, especialmente em discos IDE
Sistemas Unix (Linux, FreeBSD, etc) são ricos em ferramentas de monitoração – aprenda a usa-las!
vmstat
netstat
ps
/proc

Espalhando Arquivos pelos Discos

Para obter máxima performance de E/S, utilize discos dedicados para:
Log de transações
Índices de tabelas
Tabelas muito/pouco consultadas
Tabelas muito/pouco atualizadas
Utilize links simbólicos para mover tabelas, índices e etc para os discos dedicados

Objetos x Arquivos

Dado o banco de dados, qual o seu diretório:
select datname, oid from pg_database;
teste | 16976

Dado a tabela, qual o seu arquivo:
select relname, relfilenode from pg_class;
contato_pkey | 16979
contato | 16977

Então os dados da tabela "contato" do banco "teste"
estão no arquivo base/16976/16977

Recomendações

25% da RAM para shared buffer cache

2-4% da RAM para sort buffers
Pode ser necessário recompilar o kernel (do Linux)
para sistemas com mais do que 2Gb de RAM
Algumas versões do Linux não suportam mais do que
2Gb de memória compartilhada (shared buffer cache)

Apresentação de: **Fernando Lozano** **www.lozano.eti.br**

PostgreSQL Tuning: O elefante mais rápido que um leopardo

O Banco está Lento – Problemas Comuns

• 60% dos problemas são relacionados ao mau uso da linguagem SQL;
• 20% dos problemas são relacionados a má modelagem do banco de dados;
• 10% dos problemas são relacionados a má configuração do SGDB;
• 10% dos problemas são relacionados a má configuração do SO.

O Banco está Lento – Decisões erradas

Concentração de regras de negócio na aplicação para processos em lote;
• Integridade referencial na aplicação
• Mal dimensionamento de I/O (CPU, Plataforma, Disco)
• Ambientes virtualizados (Vmware, XEN, etc..) em AMD64/EMT64
• Uso de configurações padrões do SO e/ou do PostgreSQL

Melhor Hardware

Servidores dedicados para o PostgreSQL
• Storage com Fiber Channel e iSCSI: Grupos de RAID dedicados
• RAID 5 ou 10: por Hardware
• Mais memória! (Até 4GB em 32 bits)
• Processadores de 64 bits: Performance até 3 vezes do que os 32 bits (AMD64 e EMT64 - Intel)

Melhor SO

Sistemas Operacionais *nix: Linux (Debian, Gentoo), FreeBSD, Solaris, etc
• Em Linux: use Sistemas de arquivos XFS (noatime), Ext3 (writeback, noatime), Ext2
• Instale a última versão do PostgreSQL (atualmente 8.2) e à partir do código-fonte
• Não usar serviços concorrentes (Apache, MySQL, SAMBA...) em discos, semáforos e shared memory
• Usar, se possível, um kernel (linux) mais recente (e estável)

Parâmetros do SO – Modificando o *nix

```
echo "2" > /proc/sys/vm/overcommit_memory
echo "25%" > /proc/sys/kernel/shmmax
echo "25%/64" > /proc/sys/kernel/shmall
```

```
 echo "deadline" > /sys/block/sda/queue/scheduler
 echo "250 32000 100 128" > /proc/sys/kernel/sem
 echo "65536" > /proc/sys/fs/file-max
 ethtool -s eth0 speed 1000 duplex full autoneg off
 echo "16777216" > /proc/sys/net/core/rmem_default
 echo "16777216" > /proc/sys/net/core/wmem_default
 echo "16777216" > /proc/sys/net/core/wmem_max
 echo "16777216" > /proc/sys/net/core/rmem_max
pmanson:~# su - postgres
postgres@pmanson:~$ ulimit 65535

/etc/security/limits.conf
postgres soft nofile 4096
postgres hard nofile 63536
```

Como Orgaizar os Discos – O Melhor I/O

Discos ou partições distintos para:
– Logs de transações (WAL)
– Índices: Ext2
– Tabelas (particionar tabelas grandes)
– Tablespace temporário (em ambiente BI)*
– Archives
– SO + PostgreSQL
– Log de Sistema
* Novo no PostgreSQL 8.3

postgresql.conf – Memória

• max_connections: O menor número possível
• shared buffers: 33% do total -> Para operações em execução
• temp_buffers: Acesso às tabelas temporárias
• work_mem: Para agregação, ordenação, consultas complexas
• maintenance_work_mem: 75% da maior tabela ou índice
• max_fsm_pages: Máximo de páginas necessárias p/ mapear espaço livre. Importante para operações de UPDATE/DELETE.

postgresql.conf – Disco e WAL

• wal_sync_method: open_sync, fdatasync, open_datasync
• wal_buffers: tamanho do cache para gravação do WAL
• commit_delay: Permite efetivar várias transações na mesma chamada de fsync
• checkpoint_segments: tamanho do cache em disco para operações de escrita
• checkpoint_timeout: intervalo entre os checkpoints
• wal_buffers: 8192kB -> 16GB
• bgwriter: ?????
• join_collapse_limit = > 8

Tuning de SQL

• Analyze:
test_base=# EXPLAIN ANALYZE SELECT foo FROM bar;
• Ferramentas:
– Pgfouine;
– Pgadmin3;
– PhpPgAdmin;

Manutenção

• Autovacuum X Vacuum: Depende do uso (Aplicações Web, OLTI, BI)
– Vacuum:
• vacuum_cost_delay: tempo de atraso para vacuum executar automaticamente nas tabelas grandes
– Autovacuum (ativado por padrão a partir da versão 8.3):
• autovacuum_naptime: tempo de espera para execução do autovacuum.

Ferramentas de Stress

• Pgbench: no diretório do contrib do PostgresSQL, padrão de transações do tipo TPC-B.
• DBT-2: Ferramenta da OSDL, padrão de transações do tipo TPC-C.
• BenchmarkSQL: Ferramenta Java para benchmark em SQL para vários banco de dados (JDBC), padrão de transações do tipo TPC-C.
• Jmeter: Ferramenta Java genérica para testes de stress, usado para aplicações (Web, ...) e também pode ser direto para um banco de dados.

Quando o tuning não resolve
• Escalabilidade vertical:
– Mais e melhores discos;
– Mais memória;
– Melhor processador (quad core, 64bits)
• Escalabilidade horizontal:
– Pgpool I (distribuição de carga de leitura e pool de conexões)
– PgPool II (PgPool I + paralelização de grandes consultas)
– Slony I (Replicação Multi-Master Assíncrona)
– Warm Stand By
– Pgbouncer + PL/Proxy + Slony
Apresentação de: Fernando Ike e Fábio Telles

Checklist de performance do PostgreSQL 8.0

Autor: Fábio Telles Rodriguez

Tradução livre do texto "PostgreSQL 8.0 Performance Checklist", publicado por Josh Berkus em 12/01/2005 em:

- http://www.powerpostgresql.com/PerfList

Aqui está um conjunto de regras para configurar seu servidor PostgreSQL 8.0. Muito do que está abaixo é baseado em evidências ou testes de escalabilidade práticos; há muito sobre performance de bancos de dados que nós e a OSDL, ainda estamos trabalhando. Contudo, isto deve ser um inicio. Todas as informações abaixo são úteis a partir de 12 de janeiro de 2005 e serão atualizadas depois. Discussões sobre configurações abaixo superam as que eu realizei no General Bits.

Cinco Princípios de Hardware para Configurar o seu Servidor PostgreSQL

1. Discos > RAM > CPU

Se você vai gastar dinheiro em um servidor PostgreSQL, gaste em arranjos de discos de alta performance e tenha processadores medianos e uma memória adequada. Se você tiver um pouco mais de dinheiro, adquira mais RAM. PostgreSQL, como outros SGDBs que suportam ACID, utilizam E/S muito intensamente e é raro uma aplicação utilizar mais a CPU do que a placa SCSI (com algumas exceções, claro). Isto se aplica tanto a pequenos como grandes servidores; obtenha uma CPU com custo baixo se isso permitir você comprar uma placa RAID de alta performance ou vários discos.

2. Mais unidades de discos == Melhor

Tendo múltiplos discos, o PostgreSQL e a maioria dos sistemas operacionais irão paralelizar as requisições de leitura e gravação no banco de dados. Isto faz uma enorme diferença em sistemas transacionais, e uma significativa melhoria em aplicações onde o banco de dados inteiro não cabe na RAM. Com os tamanhos mínimos de discos (72GB) você será tentado a utilizar apenas um disco ou um único par espelhado em RAID 1; contudo, você verá que utilizando 4, 6 ou até 14 discos irá render um impulso na performance. Ah, e SCSI é ainda significativamente melhor em fluxo de dados em BD que um IDE ou mesmo um Serial ATA.

3. Separe o Log de Transações do Banco de Dados

Assumindo que você já investiu dinheiro num arranjo com tamanho decente num conjunto de discos, existe um monte de opções mais inteligentes do que jogar tudo num único RAID. De inicio coloque o log de transações (pg_xlog) no seu próprio recurso de discos (um arranjo ou um disco), o que causa uma diferença de cerca de 12% na performance de bancos de dados com grande volume de gravações. Isto é especialmente vital em pequenos sistemas com discos SCSI ou IDE lentos: mesmo em um servidor com dois discos, você pode colocar o log de transações sobre o disco do sistema operacional e tirar algum benefício.

4. RAID 1+0/0+1 > RAID 5

RAID 5 com 3 discos tem sido um desafortunado padrão entre vendedores de servidores econômicos. Isto possibilita a mais lenta configuração de discos possível para o PostgreSQL; você pode esperar pelo menos 50% a menos de velocidade nas consultas em relação ao obtido com discos SCSI normais. Por outro lado, foque em RAID 1 ou 1+0 para um conjunto de 2, 4 ou 6

discos. Acima de 6 discos, o RAID 5 começa a ter uma performance aceitável novamente, e a comparação tende a ser bem melhor com base na sua controladora individual. No entanto, o mais importante, usar uma placa RAID barata pode ser um risco; é sempre melhor usar RAID por software do que um incorporado numa placa Adaptec que vem com seu servidor.

5. Aplicações devem rodar bem junto

Outro grande erro que eu vejo em muitas organizações e colocar o PostgreSQL em um servidor com várias outras aplicações competindo pelos mesmos recursos. O pior caso é colocar o PostgreSQL junto com outros SGDBs na mesma máquina; ambos bancos de dados irão lutar pela banda de acesso aos discos e o cache de disco do SO, e ambos vão ter uma performance pobre. Servidores de arquivo e programas de log de segurança também são ruins. O PostgreSQL pode compartilhar a mesma máquina com aplicações que utilizam principalmente CPU e RAM intensamente, como o Apache, garantindo que exista RAM suficiente.

Doze Ajustes que Você Irá Querer Fazer no Seu Arquivo PostgreSQL.Conf

Existem um monte de novas opções verdadeiramente assustadoras no arquivo PostgreSQL.conf. Mesmo as já familiares opções das 5 últimas versões mudaram de nomes e formato dos parâmetros. Elas tem a intenção de dar ao administrador de banco de dados mais controle, mas podem levar algum tempo para serem usados.

O que segue são configurações que a maioria dos DBAs vão querer alterar, focado no aumento de performance acima de qualquer outra coisa. Existem algumas poucas configurações que particularmente a maioria dos usuários não querem mexer, mas quem o fizer irá descobri-las indispensáveis. Para estes, vocês terão de aguardar pelo livro.

Lembre-se: as configurações no PostgreSQL.conf precisam ser descomentadas para fazerem efeito, mas recomentá-las não restaurará necessariamente o valor padrão!

Conexão

listen_addresses:
Substitui as configurações tcp_ip e o virtual_hosts do 7.4. O padrão é localhost na maioria das instalações, habilitando apenas conexões pelo console. A maioria dos DBAs irá querer mudar isto para "*", significando que todas as interfaces avaliáveis, após configurar as permissões em hba.conf apropriadamente, irão tornar o PostgreSQL acessível pela rede. Como uma melhoria sobre a versão anterior, o"localhost" permite conexões pela interface de "loopback", 127.0.0.1, habilitando vários utilitários baseados em servidores web.

max_connections:
Exatamente como na versão anterior, isto precisa ser configurado para o atual número de conexões simultâneas que você espera precisar. Configurações altas vão requerer mais memória compartilhada (shared_buffers). Como o overhead por conexão, tanto do PostgreSQL como do SO do host podem ser bem altos, é importante utilizar um pool de conexões se você precisar servir um número grande de usuários. Por exemplo, 150 conexões ativas em um servidor Linux com um processador médio de 32 bits consumirá recursos significativos, e o limite deste hardware é de 600. Claro que um hardware mais robusto irá permitir mais conexões.

Memória

shared_buffers:
Como um lembrete: Este não é a memória total do com o qual o PostgreSQL irá trabalhar. Este é o bloco de memória dedicado ao PostgreSQL utilizado para as operações ativas, e deve ser a menor parte da RAM total na máquina, uma vez que o PostgreSQL usa o cache de disco também. Infelizmente, o montante exato de shared buffers requer um complexo cálculo do total de RAM, tamanho do banco de dados, número de conexões e complexidade das consultas. Assim, é melhor seguir algumas regras na alocação, e monitorar o servidor (particularmente as visões pg_statio) para determinar ajustes.

Em servidores dedicados, valores úteis costumas ficar entre 8MB e 400MB (entre 1000 e 50.000 para páginas de 8K). Fatores que aumentam a quantidade de shared buffers são grandes porções

ativas do banco de dados, consultas grandes e complexas, grande número de conexões simultâneas, longos procedimentos e transações, maior quantidade de RAM disponível, CPUs mais rápidas ou em maior quantidade obviamente, outras aplicações na mesma máquina. Contrário a muitas expectativas, alocando, muita, demasiadamente shared_buffers pode até diminuir a performance, aumentando o tempo requerido para explora-la. Aqui estão alguns exemplos baseados em experiências e testes TPC em máquinas Linux:

- Laptop, processador Celeron, 384MB RAM, banco de dados de 25MB: 12MB/1500;
- Servidor Athlon, 1GB RAM, banco de dados de 10GB para suporte a decisão: 120MB/15000;
- Servidor Quad PIII, 4GB RAM, banco de dados de 40GB, com 150 conexões e processamento pesado de transações: 240MB/30000;
- Servidor Quad Xeon, 8GB RAM, banco de dados de 200GB, com 300 conexões e processamento pesado de transações: 400MB/50000.

Favor notar que incrementando shared_buffer, e alguns outros parâmetros de memória, vão requerer que você modifique o System V do seu sistema operacional. Veja a documentação principal do PostgreSQL para instruções nisto.

work_mem:
Costuma ser chamado de sort_mem, mas foi renomeado uma vez que ele agora cobre ordenações, agregações e mais algumas operações. Esta memória não é compartilhada, sendo alocada para cada operação (uma a várias vezes por consulta); esta configuração está aqui para colocar um teto na quantidade de memória que uma única operação ocupar antes de ser forçada para o disco. Este deve ser calculado dividindo a RAM disponível (depois das aplicações e do shared_buffers) pela expectativa de máximo de consultas concorrentes vezes o número de memória utilizada por conexão. Considerações devem ser tomadas sobre o montante de work_mem por consulta; processando grandes conjuntos de de dados requisitará mais. Bancos de dados de aplicações Web geralmente utilizam números baixos, com numerosas conexões mas consultas simples, 512K a 2048K geralmente é suficiente. Contrariamente, aplicações de apoio a decisão com suas consultas de 160 linhas e agregados de 10 milhões de linhas precisam de muito, chegando a 500MB em um servidor com muita memória. Para bancos de dados de uso misto, este parâmetro pode ser ajustado por conexão, em tempo de execução, nesta ordem, para dar mais RAM para consultas específicas.

maintenance_work_mem:
Formalmente chamada de vacuum_mem, esta quantidade de RAM é utilizada pelo PostgreSQL para o VACUUM, ANALYZE, CREATE INDEX, e adição de chaves estrangeiras. Você deve aumentar quanto maior forem suas tabelas do banco de dados e quanto mais memória RAM você tiver de reserva, para fazer estas operações o mais rápidas possível. Uma configuração com 50% a 75% da sua maior tabela ou índice em disco é uma boa regra, ou 32MB a 256MB onde isto não pode ser determinado.

Disco e WAL

checkpoint_segments:
Define o tamanho do cache de disco do log de transações para operações de escrita. Você pode

ignorar isto na maioria dos bancos de dados web com a maioria das operações em leitura, mas para bancos de dados de processamento de transações ou para bancos de dados envolvendo grandes cargas de dados, o aumento dele é crítico para a performance. Dependendo do volume de dados, aumente ele para algo entre 12 e 256 segmentos, começando conservadoramente e aumentando se você ver mensagens de aviso no log. O espaço requerido no disco é igual a (checkpoint_segments * 2 + 1) * 16MB, então tenha certeza de ter espaço em disco suficiente (32 significa mais de 1GB).

max_fsm_pages:
Dimensiona o registro que rastreia as páginas de dados parcialmente vazias para popular com novos dados; se configurado corretamente, torna o VACCUM mais rápido e remove a necessidade do VACUUM FULL ou REINDEX. Deve ser um pouco maior que o total de número páginas de dados que serão tocados por atualizações e remoções entre vacuums. Os dois modos de determinar este número são rodar o VACUUM VERBOSE ANALYZE, ou se estiver utilizando autovacuum (veja abaixo) configures este de acordo com o parâmetro -V como uma porcentagem do total de páginas de dados utilizado por seu banco de dados. fsm_pages requer muito pouco memória, então é melhor ser generoso aqui.

vacuum_cost_delay:
Se você tiver tabelas grandes e um significativo montante de atividades de gravações concorrentes, você deve querer fazer uso deste novo recurso que diminui a carga de I/O do VACUUM sobre o custo de fazê-las mais longas. Como este é um novo recurso, é um complexo de 5 configurações dependentes para o qual nós temos apenas poucos testes de performance. Aumentando o vacuum_cost_delay para um valor não zero ativa este recurso; use um atraso razoável, algo entre 50 e 200ms. Para um ajuste fino, aumente o vaccum_cost_page_hit e diminua o vacuum_cost_page_limit irá diminuir o impacto dos vacuums e tornará eles mais longos; em testes de Jan Wieck's num teste de processamento de transações, um delay de 200, page_hit de 6 e limit de 100 diminuiu o impacto do vacuum em mais de 80% enquanto triplicou o tempo de execução dele.

Planejador de Consultas

Estas configurações permitem o planejador de consultas fazer estimativas mais precisas dos custos de operação e assim escolher o melhor plano de execução. Os dois valores de configurações para se preocupar são:

effective_cache_size:
Diz ao planejador de consultas o mais largo objeto do banco de dados que pode se esperar ser cacheado. Geralmente ele deve ser configurado em cerca de 2/3 da RAM, se estiver num servidor dedicado. Num servidor de uso misto, você deve estimar quanto de RAM e cache de disco outras aplicações estarão utilizando e subtrair eles.

random_page_cost:
Uma variável que estima o custo médio em buscas por páginas de dados indexados. Em máquinas mais rápidas, com arranjos de discos velozes ele deve ser reduzido para 3.0, 2.5 ou até mesmo 2.0. Contudo, se a porção ativa do seu banco de dados é muitas vezes maior que a sua RAM, você vai querer aumentar o fator de volta para o valor padrão de 4.0. Alternativamente, você pode basear seus ajustem na performance. Se o planejador injustamente a favor de buscas seqüenciais sobre buscas em índices, diminua-o. Se ele estiver utilizando índices lentos quando não deveria,

aumente-o. Tenha certeza de testar uma variedade de consultas. Não abaixe ele para menos de 2.0; se isto parecer necessário, você precisa de ajustem em outras áreas, como as estatísticas do planejador.

Logging

log_destination:
Isto substitui o intuitivo a configuração syslog em versões anteriores. Suas escolhas são usar o log administrativo do SO (syslog ou eventlog) ou usar um log separado para o PostgreSQL (stderr). O primeiro é melhor para monitorar o sistema; o último é melhor para encontrar problemas no banco de dados e para o tuning.

redirect_stderr:
Se você decidir usr um log separado para o PostgreSQL, esta configuração permitirá registrar num arquivo utilizando uma ferramenta nativa do PostgreSQL ao invés do redirecionamento em linha de comando, permitindo a rotação do log. Ajuste para True, e então ajuste o log_diretory para dizer onde colocar os logs. A configuração padrão para o log_filename, log_reotation_size e log_rotation)age são bons para a maioria das pessoas.

Autovacuum e você

Assim que você entra em produção no 8.0, você vai querer fazer um plano de manutenção incluindo VACUUMs e ANALYZEs. Se seus bancos de dados envolvem um fluxo contínuo de escrita de dados, mas não requer a maciças cargas e apagamentos de dados ou freqüentes reinícios, isto significa que você deve configurar o pg_autovacuum. Isto é melhor que agendar vaccuns porque:

- Tabelas sofrem o vacuum baseados nas suas atividades, excluindo tabelas que apenas sofrem leituras.
- A freqüência dos vacuums cresce automaticamente com o crescimento da atividade no banco de dados.
- É mais fácil calcular o mapa de espaço livre e evitar o inchaço do banco de dados.

Carga de dados no banco: http://pgdocptbr.sourceforge.net/pg80/populate.html

Dicas de Performance em aplicações com PostgreSQL
Tradução do texto original de Josh Berkus

O que se segue é a versão editada de um conjunto de conselhos que eu tenho dado ao time da Sun no redesenho de uma aplicação em C++ que foi construída para MySQL, portado para o PostgreSQL, e nunca otimizado para performance. Ocorreu que estes conselhos podem ser geralmente úteis para a comunidade, então aí vão eles.

Projeto de aplicações para performance no PostgreSQL

Escrevendo regras de consultas

Para todos os sistemas gerenciadores de bancos de dados (SGDBs), o tempo por rodada significativo. Este é o tempo que leva para uma consulta passar pelo analisador de sintaxe da linguagem, o driver, a interface da rede, o analisador de sintaxe do banco de dados, o planejador, o executor, o analisador de sintaxe novamente, voltar pela interface de rede, passar pelo manipulador de dados do driver e para o cliente da aplicação. SGDBs variam na quantidade de tempo e CPU que elas levam para processar este ciclo, e por uma variedade de razões, o PostgreSQL possui um alto consumo de tempo e recursos do sistema por rodada.

Contudo, o PostgreSQL tem um overhead significativo por transação, incluindo o log de saída e as regras de acesso que precisam ser ajustadas em cada transação. Enquanto você pode pensar que não está utilizando transações para um simples comando de leitura SELECT, de fato, cada simples comando no PostgreSQL é uma transação. Na ausência de uma transação explícita, o comando é por si mesmo implicitamente uma transação.

Passando por isto, o PostgreSQL é claramente o segundo depois do Oracle em processamento de consultas complexas e longas com transações com vários comandos com fácil resolução de conflitos de concorrência. Ele também suporta cursores, tanto rolável quanto não rolável.

Dica 1: Nunca use várias consultas pequenas quando uma grande consulta pode fazer o trabalho.

É comum em aplicações MySQL lidar com joins no código da aplicação, ou seja, consultando o ID de um registro relacionado e então iterando através dos registros filhos com aquele ID manualmente. Isto pode resultar em rodar centenas de consultas por tela de interface com o usuário. Cada uma destas consultas levam 2 a 6 milissegundos por rodada, o que significa que se você executar cerca de 1000 consultas, neste ponto você estará perdendo de 3 a 5 segundos. Comparativamente, solicitando estes registros numa única consulta levará apenas algumas centenas de milissegundos, economizando cerca de 80% do tempo.

Dica 2: Agrupe vários pequenos UPDATEs, INSERTs ou DELETEs em um único comando ou, se não for possível, em uma longa transação.

Antes, a falta de subselects nas versões anteriores do MySQL fizeram com que os desenvolvedores de aplicação projetassem seus comandos de modificação de dados (DML) da mesma forma que as junções em middleware. Esta é uma má idéia para o PostgreSQL. Ao invés, você irá tirar vantagem de subselects e joins no seu comando UPDATE, INSERT, e DELETE para tentar realizar modificações em lote com um único comando. Isto reduz o tempo da rodada e o overhead da transação.

Em alguns casos, contudo, não há uma única consulta que consiga alterar todas as linhas que você deseja e você irá usar um grupo de comandos em série. Neste caso, você irá querer se assegurar de envolver a sua séria de comandos DML em uma transação explícita (ex. BEGIN; UPDATE; UPDATE; UPDATE; COMMIT;). Isto reduz o overhead de transação e corta o tempo de execução em até 50%.

Dica 3: Considere realizar cargas em lotes ao invés de INSERTs seriais.

O PostgreSQL prove um mecanismo de carga em lote chamado COPY, que pode pegar uma entrada de um arquivo ou pipe delimitado por tabulações ou CSV. Quando o COPY pode ser usado no lugar de centenas de INSERTs, ele pode cortar o tempo de execução em até 75%.

Dica 4: O DELETE é caro

É comum para um desenvolvedor de aplicação pensar que o comando DELETE é praticamente não tem custo. Você está apenas desligando alguns nós, correto? Errado. SGDBs não são sistemas de arquivo; quando você apaga uma linha, índices precisam ser atualizados, o espaço liberado precisa ser limpo, fazendo a exclusão de fato mais cara que a inserção. Assim, aplicações que habitualmente apagam todas as linhas de detalhe e repõe elas com novas toda vez que é realizada qualquer alteração estão economizando esforço no lado da aplicação e empurrando este dentro do banco de dados. Quando possível, isto deve ser substituído pela substituição mais discriminada das linhas, como atualizar apenas as linhas modificadas.

Além disso, quando for limpar toda uma tabela, sempre use o comando TRUNCATE TABLE ao invés de DELETE FROM TABLE. A primeira forma é até 100 vezes mais rápida que a posterior devido ao processamento da tabela como um todo ao invés de uma linha por vez.

Dica 5: Utilize o PREPARE/EXECUTE para iterações em consultas

Algumas vezes, mesmo tentando consolidar iterações de consultas semelhantes em um comando mais longo, nem sempre isto é possível de estruturar na sua aplicação. É para isto que o PREPARE … EXECUTE serve; ele permite que o motor do banco de dados pule o analizador de sintaxe e o planejador para cada iteração da consulta. Por exemplo:

Preparar:
```
query_handle = query('SELECT * FROM TABLE WHERE id = ?')
(parameter_type = INTEGER)
```

Então inicie as suas iterações:
```
for 1..100
query_handle.execute(i);
end
```

Classes para a preparação de comandos no C++ são explicadas na documentação do libpqxx.

Isto irá reduzir o tempo de execução na direta proporção do tamanho do número de iterações.

Dica 6: Use pool de conexões efetivamente

Para uma aplicação web, você irá perceber que até 50% do seu potencial de performance pode ser controlado através do uso, e configuração apropriada de um pool de conexões. Isto é porque criar e destruir conexões no banco de dados leva um tempo significativo no sistema, e um excesso de conexões inativas continuarão a requerer RAM e recursos do sistema.

Há um número de ferramentas que você pode utilizar para fazer um pool de conexões no PostgreSQL. Uma ferramenta de terceiros de código aberto é o pgPool. Contudo, para uma aplicação em C++ com requisitos de alta disponibilidade, é provavelmente melhor utilizar a técnica de pseudo-pooling nativa do libpqxx chamada de "conexões preguiçosas". Eu sugiro contatar a lista de de e-mail para mais informações sobre como utilizar isto.

Com o PostgreSQL, você irá querer ter quantas conexões persistentes (ou objetos de conexão) forem definidas no seu pico normal de uso de conexões concorrentes. Então, se o uso máximo normal (no início da manhã, digamos) é de 200 conexões concorrentes de agentes, usuários e componentes, então você irá querer que ter esta quantidade definida para que sua aplicação não tenha que esperar por novas conexões durante o pico onde será lenta na criação.

Fábio Telles em:

http://www.midstorm.org/~telles/2007/01/05/dicas-de-performance-em-aplicacoes-com-postgresql/

Veja Melhorando a Performance do seu HD:
Piter Punk em: http://piterpunk.info02.com.br/artigos/hdparm.html

Gerenciando Recursos do Kernel:
http://www.postgresql.org/docs/8.3/interactive/kernel-resources.html

Escolhendo o sistema Operacional para o servidor do SGBD PostgreSQL

Sem discussão, o Linux/Unix ganha 25% a 50% de performance em cima do Windows. Isso se dá ao melhor gerenciamento de recursos.

http://vivaolinux.com.br/dicas/verDica.php?codigo=9847

Melhorando a performance do PostgreSQL com o comando VACUUM em:
http://imasters.uol.com.br/artigo/2421/postgresql/melhorando_a_performance_do_postgresql_com_o_comando_vacuum/

Otimizando bancos PostgreSQL - Parte 01

Olá! Neste meu primeiro artigo gostaria de descrever alguns ajustes finos do PostgreSQL que ajudam a melhorar a performance geral do banco de dados. Estas configurações são válidas para a versão 8.1. Caso você possua uma versão anterior, recomendo fortemente a atualização para a **8.1**, pois esta última é **muito** mais rápida que as anteriores, mesmo utilizando as configurações padrão. Infelizmente, a maior parte da degradação de performance de um banco de dados está na estrutura e/ou nos comandos SELECT mal elaborados. Nete artigo, será feita uma abordagem única e específica nos ajustes de configuração do SGBD, fazendo-o usufruir do máximo dos recursos de hardware onde está instalado o serviço do PostgreSQL.

É importante salientar também que não existe fórmula mágica para as configurações, sendo que as opções de um servidor pode não ser a melhor opção para outro.

Edição do arquivo postgresql.conf

Para começar, localize o arquivo chamado **postgresql.conf**. Este arquivo encontra-se no diretório de dados do cluster (ou agrupamento de bancos de dados) o qual você está inicializando. Em instalações normais, você pode encontrá-lo em:

- Microsoft Windows:
  - Pasta "Arquivos de Configuração" no Menu Iniciar/Programas
  ou
  - C:\Arquivos de Programas\PostgreSQL\8.1\data\

- Linux (Red Hat e Fedore)
  /var/lib/pgsql/data

Após encontrá-lo, certifique-se de criar uma cópia de backup do arquivo antes de alterá-lo, pois

erros na configuração podem fazer com que o PostgreSQL não seja inicializado.

Feito o backup, abra o arquivo em modo de edição. Se estiver no Linux, certifique-se de estar logado com o usuário **root** ou **postgres**.

Os parâmetros devem ser preenchidos seguindo o padrão **nome_do_parametro = valor**. Lembrando que para valores do tipo data e texto deve-se envolver o valor com aspas simples, e para os valores numéricos não inserir separador de milhar. Não esqueça de remover o caracter cerquilha **"#"** do início das opções que deseja ativar. Todas as alterações serão efetivadas após a reinicialização do serviço PostgreSQL.

shared_buffers

Define o espaço de memória alocado para o PostgreSQL armazenar as consultas SQL antes de devolvê-las ao buffer do sistema operacional. Esta opção pode solicitar que parâmetros do Kernel sejam modificados para liberar mais memória compartilhada do sistema operacional, pois esta passa a ser utilizada também pelo Postgre, em maior quantidade. O valor desta configuração está expresso em blocos de 8 Kbytes (128 representa 1.024 Kbytes ou 1 Mb).

Uma boa pedida é utilizar valores de 8% a 12% do total de RAM do servidor para esta configuração. Caso, após mudar o valor deste parâmetro, o PostgreSQL não inicializar o cluster em questão, altere o sistema operacional para liberar mais memória compartilhada. Consulte o manual ou equivalentes do seu sistema operacional para obter instruções de como aumentar a memória compartilhada (Shared Memory) disponível para os programas.

Exemplo:
shared_buffers = 2048 # Seta a memória compartilhada para 16 Mbytes

work_mem

Configura o espaço reservado de memória para operações de ordem e manipulação/consulta de índices. Este parâmetro configura o tamanho em KBytes utilizado no servidor para cada conexão efetivada ao SGBD, portanto esteja ciente que o espaço total da RAM utilizado (valor da opção multiplicado pelo número de conexões simultâneas) não deve ultrapassar 20% do total disponível (valor aproximado).

Exemplo:
work_mem = 2048 # Configura 2 Mbytes de RAM do servidor para operações de ORDER BY, CREATE INDEX e JOIN disponíveis para cada conexão ao banco.

maintenance_work_mem

Expressa em KBytes o valor de memória reservado para operações de manutenção (como VACUUM e COPY). Se o seu processo de VACUUM está muito custoso, tente aumentar o valor deste parâmetro.

Nota: O total de memória configurada neste parâmetro é utilizado somente durante as operações de manutenção do banco de dados, sendo liberada durante o seu uso normal.

Exemplo:
maintenance_work_mem = 16384 # 16 Mbytes reservados para operações de manutenção.

max_fsm_pages

Em bancos de dados grandes, é ideal que a cada execução do VACUUM mais páginas "sujas" sejam removidas do banco de dados do que a quantidade padrão, principalmente por questões de espaço em disco, e claro, performance. Porém, a configuração padrão traz apenas 20000 páginas. Em

bancos transacionais, com no mínimo 10 usuários, o número mensal de páginas sujas pode exceder este valor.

Para realizar uma limpeza maior, aumente o valor desta configuração. Nota: incrementando o valor deste parâmetro pode resultar em aumento do tempo para execução do VACUUM, e cada página ocupa em média 6 bytes de RAM constantemente. O comando VACUUM pode ser comparado ao comando PACK do DBASE (DBFs), porém possui mais funcionalidades.

Exemplo:
max_fsm_pages = 120000 # Realiza a procura por até 120.000 páginas sujas na limpeza pelo VACUUM utilizando cerca de 71 Kb de RAM para isto.

wal_buffers

Número de buffers utilizados pelo WAL (Write Ahead Log). O WAL garante que os registros sejam gravados em LOG para possível recuperação antes de fechar uma transação. Porém, para bancos transacionais com muitas operações de escrita, o valor padrão pode diminuir a performance. Entretanto, é importante ressalvar que aumentar muito o valor deste parâmetro pode resultar em perda de dados durante uma possível queda de energia, pois os dados mantém-se . Cada unidade representa 8 Kbytes de uso na RAM. Valores ideais estão entre 32 e 64. Se o disco conjunto do servidor (HW & SW) são quase infalíveis, tente usar 128 ou 256.

Exemplo:
wal_buffers = 64 # Seta para 512 Kbytes a memória destinada ao buffer de escrita no WAL.

effective_cache_size

Esta configuração dita o quanto de memória RAM será utilizada para cache efetivo (ou cache de dados) do banco de dados. Na prática, é esta a configuração que dá mais fôlego ao SGBD, evitando a constante leitura das tabelas e índices ,dos arquivos do disco rígido.

Como exemplo, se uma tabela do banco de dados possui 20 Mbytes, e o tamanho para esta configuração limita-se aos quase 8 Mbytes padrão, o otimizador de querys irá carregar a tabela por etapas, em partes, até que ela toda seja vasculhada em busca dos registros. Esta opção impacta diretamente aumentando a performance do banco de dados, principalmente quando há concorrência, pois diminui consideravelmente as operações de I/O de disco.

Em consultas pela internet, encontrei várias referências para utilizar no máximo 25% da RAM total. Porém, se o servidor for dedicado, ou dispôr de uma grande quantidade de memória (512 Mbytes ou mais), recomendo o uso de 50% da RAM para esta configuração. Cada unidade corresponde a 8 Kbytes de RAM.

Exemplo:
effective_cache_size = 32768 # Seta o cache de dados do PostgreSQL para 256 Mbytes de RAM.

random_page_cost

Define o custo (em tempo) para a seleção do plano de acesso aos dados do banco. Se você possui discos rígidos velozes (SCSI, por exemplo), tente utilizar valores como 1 ou 2 para esta configuração. Para os demais casos, limite-se a 3 ou 4. Isto impacta no plano estabelecido pelo otimizador interno, que pode realizar um Index_Scan ou Table_Scan, etc, conforme aquilo que ele determinar ser mais otimizado.

Exemplo:
random_page_cost = 2 # Diminui o tempo para seleção aleatória de páginas do otimizador de consultas.

Notas gerais

Caso os valores inseridos possuam algum erro, é possível que o PostgreSQL não consiga inicializar o agrupamento de banco de dados no qual o arquivo postgresql.conf foi modificado. Se isto ocorrer, retorne o backup do mesmo e certifique-se de que as modificações estão corretamente setadas.

É possível obter ganhos de performance significativos com a correta seleção dos valores para as configurações acima. Tente várias combinações dependendo da capacidade e do uso do servidor (hardware) onde está instalado o PostgreSQL 8.1.

Otimizando bancos PostgreSQL - Parte 02

Criação de Índices Parciais (Partial Indexes)
Em tabelas com muitos registros, a utilização de índices normais pode causar um desempenho insatisfatório, principalmente quando se trata decolunas que representam abstrações de dados com pouca variação. É o caso de colunas representando tempo (DATE, TIME e TIMESTAMP), ou colunas numéricas representando tipos pré-definidos(Ex.: Regiões, Sexo, Faixa Salarial, etc.). Tabelas de movimentação analítica com estes tipos de colunas podem conter milhares, ou até milhões de registros. Entretanto, em consultas SQL específicas por um determinado valor, um índice normal completo (Full Index) irá considerar na consulta todos os registros da tabela, organizados na ordem do índice.

O PostgreSQL possui um fantástico recurso para criação de índices que permite delimitar os registros que este irá considerar. Isto representa um enorme ganho de desempenho, especialmente em consultas SQL que utilizam filtros complexos no WHERE.

Vamos exemplificar este caso. Considere uma tabela de movimentação analítica de estoque de uma empresa de comércio comum. Vamos usar um modelo simples, apenas para demonstrar o caso. Utilize o código SQL abaixo para criar a tabela:

```
-- Criação da Tabela
CREATE TABLE estoque(
ID_Empresa INT2 NOT NULL,
ID_Produto INT4 NOT NULL,
ID_Local_Estoque INT2 NOT NULL,
TIPO_Entrada BOOLEAN NOT NULL DEFAULT false,
QTD_Quantidade NUMERIC(12,6) DEFAULT 0.000000,
VAL_Unitario NUMERIC(15,3) DEFAULT 0.000,
DT_Movimento DATE NOT NULL
);


-- Definição da chave primária
ALTER TABLE estoque ADD PRIMARY KEY(ID_Empresa,ID_Produto,ID_Local_Estoque);

-- Criação de Índice sobre o campo Data
CREATE INDEX idx_DATA ON estoque (DT_Movimento, ID_Empresa);
```

Imagine esta tabela com mais de 2.000.000 registros. O departamento de gerência de estoque emite relatórios mensais sobre a movimentação de estoque dos produtos para conferência. Um exemplo de relatório é de Entrada e Saída Consolidada, que considera os valores de entrada e saída por período.

Um SQL típico para demonstrar as informações do mês de Dezembro de 2006 utilizaria o filtro no WHERE mencionando o campo DT_Movimento da seguinte forma: (...) WHERE DT_Movimento BETWEEN 2006-12-01 AND 2006-12-31.

Considere o volume de dados caso a empresa possua um movimento de mais de 50.000 registros por mês, mantendo esta marca desde 01012000. Ao utilizar o WHERE acima, uma varredura completa no índice idx_DATA seria feita, considerando a massa completa de dados no índice.

Dependendo de condições de uso dos registros estes podem estar na memória cache, então o resultado seria rapidamente apresentado. Entretanto, caso uma pesquisa aleatória não armazenada em cache for executada, o custo de IO do gerenciador de banco de dados seria problemático.

A solução neste caso - e uma medida muito satisfatória - é a criação dos índices parciais sobre o campo data, combinando-os com um índice normal completo. É possível criar os índices parciais para datas muito além das atuais, para prever a população de registros na tabela no futuro, de modo a garantir o desempenho. Lembre-se de que se não existirem registros com uma data prevista no índice, este não terá tamanho, portanto não será prejudicial em nenhum aspecto (espaço ou IO).

Para aperfeiçoar o acesso a dados nestas condições, os índices parciais consideram a cláusula SQL WHERE:

```
-- Criação de Índice sobre o campo Data - Janeiro de 2006
CREATE INDEX idx_DATA_0106 ON estoque (DT_Movimento, ID_Empresa) WHERE
(DT_Movimento BETWEEN 2006-01-01 AND 2006-01-31);

-- Criação de Índice sobre o campo Data - Fevereiro de 2006
CREATE INDEX idx_DATA_0206 ON estoque (DT_Movimento, ID_Empresa) WHERE
(DT_Movimento BETWEEN 2006-02-01 AND 2006-02-28);

(...)

-- Criação de Índice sobre o campo Data - Dezembro de 2006
CREATE INDEX idx_DATA_1206 ON estoque (DT_Movimento, ID_Empresa) WHERE
(DT_Movimento BETWEEN 2006-12-01 AND 2006-12-31);

(...)
```

Desta forma, a todo SQL onde for utilizado a condição DT_Movimento BETWEEN 2006-12-01 AND 2006-12-31 ou sua equivalente DT_Movimento 2006-12-01 AND DT_Movimento 2006-12-31, o índice idx_DATA_0106 será apresentando para o otimizador interno como o mais eficaz, e portanto será usado.

Uma aplicação muito boa para os índices parciais é a utilização deste em tabelas que fazem parte de VIEWS (Visões) complexas. Todo o WHERE fixo da VIEW pode ser considerado em um índice parcial, o que resulta na diminuição considerável do tempo de resposta.

Tiago Adami em:
http://imasters.uol.com.br/artigo/4406/postgresql/otimizando_bancos_postgresql_-_parte_01/
http://imasters.uol.com.br/artigo/5328/postgresql/otimizando_bancos_postgresql_-_parte_02/

Otimizando operações de Entrada e Saída (I/O) em:
http://imasters.uol.com.br/artigo/4020/bancodedados/otimizando_operacoes_de_entrada_e_saida_io

Otimizar consultas SQL

Diferentes formas de otimizar as consultas realizadas em SQL.

A linguagem SQL é não procedimental, ou seja, nas sentenças se indica o que queremos conseguir e não como tem que fazer o intérprete para consegui-lo. Isto é pura teoria, pois na prática todos os gerenciadores de SQL têm que especificar seus próprios truques para otimizar o rendimento.

Portanto, muitas vezes não basta com especificar uma sentença SQL correta, e sim que além disso, há que indicar como tem que fazer se quisermos que o tempo de resposta seja o mínimo. Nesta seção, veremos como melhorar o tempo de resposta de nosso intérprete ante umas determinadas situações:

Design de tabelas

* Normalize as tabelas, pelo menos até a terceira forma normal, para garantir que não haja duplicidade de dados e aproveitar o máximo de armazenamento nas tabelas. Se tiver que desnormalizar alguma tabela pense na ocupação e no rendimento antes de proceder.
* Os primeiros campos de cada tabela devem ser aqueles campos requeridos e dentro dos requeridos primeiro se definem os de longitude fixa e depois os de longitude variável.
* Ajuste ao máximo o tamanho dos campos para não desperdiçar espaço.
* É normal deixar um campo de texto para observações nas tabelas. Se este campo for utilizado com pouca freqüência ou se for definido com grande tamanho, por via das dúvidas, é melhor criar uma nova tabela que contenha a chave primária da primeira e o campo para observações.

Gerenciamento e escolha dos índices

Os índices são campos escolhidos arbitrariamente pelo construtor do banco de dados que permitem a busca a partir de tal campo a uma velocidade notavelmente superior. Entretanto, esta vantagem se vê contra-arrestada pelo fato de ocupar muito mais memória (o dobro mais ou menos) e de requerer para sua inserção e atualização um tempo de processo superior.

Evidentemente, não podemos indexar todos os campos de uma tabela extensa já que dobramos o tamanho do banco de dados. Igualmente, tampouco serve muito indexar todos os campos em uma tabela pequena já que as seleções podem se efetuar rapidamente de qualquer forma.

Um caso em que os índices podem ser muito úteis é quando realizamos petições simultâneas sobre várias tabelas. Neste caso, o processo de seleção pode se acelerar sensivelmente se indexamos os campos que servem de nexo entre as duas tabelas.

Os índices podem ser contraproducentes se os introduzimos sobre campos triviais a partir dos quais não se realiza nenhum tipo de petição já que, além do problema de memória já mencionado, estamos lentificando outras tarefas do banco de dados como são a edição, inserção e eliminação. É por isso que vale a pena pensar duas vezes antes de indexar um campo que não serve de critério para buscas ou que é usado com muita freqüência por razões de manutenção.

Campos a Selecionar

* Na medida do possível há que evitar que as sentenças SQL estejam embebidas dentro do código da aplicação. É muito mais eficaz usar vistas ou procedimentos armazenados por que o gerenciador os salva compilados. Se se trata de uma sentença embebida o gerenciador deve compila-la antes de executa-la.
* Selecionar exclusivamente aqueles que se necessitem
* Não utilizar nunca SELECT * porque o gerenciador deve ler primeiro a estrutura da tabela antes de executar a sentença
* Se utilizar várias tabelas na consulta, especifique sempre a que tabela pertence cada campo, isso economizará tempo ao gerenciador de localizar a que tabela pertence o campo. Ao invés de SELECT Nome, Fatura FROM Clientes, Faturamento WHERE IdCliente = IdClienteFaturado, use: SELECT Clientes.Nome, Faturamento.Fatura WHERE Clientes.IdCliente = Faturamento.IdClienteFaturado.

Campos de Filtro

* Procuraremos escolher na cláusula WHERE aqueles campos que fazem parte da chave do arquivo pelo qual interrogamos. Ademais se especificarão na mesma ordem na qual estiverem definidas na chave.
* Interrogar sempre por campos que sejam chave.
* Se desejarmos interrogar por campos pertencentes a índices compostos é melhor utilizar todos os campos de todos os índices. Suponhamos que temos um índice formado pelo campo NOME e o campo SOBRENOME e outro índice formado pelo campo IDADE. A sentença WHERE NOME='Jose' AND SOBRENOME Like '%' AND IDADE = 20 seria melhor que WHERE NOME = 'Jose' AND IDADE = 20 porque o gerenciador, neste segundo caso, não pode usar o primeiro índice e ambas sentenças são equivalentes porque a condição SOBRENOME Like '%' devolveria todos os registros.

Ordem das Tabelas

Quando se utilizam várias tabelas dentro da consulta há que ter cuidado com a ordem empregada na cláusula FROM. Se desejarmos saber quantos alunos se matricularam no ano 1996 e escrevermos:

FROM Alunos, Matriculas WHERE Aluno.IdAluno = Matriculas.IdAluno AND Matriculas.Ano = 1996

o gerenciador percorrerá todos os alunos para buscar suas matrículas e devolver as correspondentes. Se escrevermos

FROM Matriculas, Alunos WHERE Matriculas.Ano = 1996 AND Matriculas.IdAluno = Alunos.IdAlunos

o gerenciador filtra as matrículas e depois seleciona os alunos, desta forma tem que percorrer menos registros.

De Cláudio em:
http://www.criarweb.com/artigos/otimizar-consultas-sql.html

Dicas de Desempenho em:
http://pgdocptbr.sourceforge.net/pg80/performance-tips.html

Diferença de Desempenho entre alguns Sistemas Operacionais

Fiz alguns teste com mesma máquina e configuração com windows 2000 server, Debian e FreeBSD 6.2, perdi um tempão instalado os tres SO (testei um de cada vez seperado) na máquina todos com configuração minima para não dizer que o debian e o FreeBSD pode ser instalado sem interface ai consumiria menos memória instalei o debian e o FreeBSD com KDE para ficar semelhante ao windows.

Uma consulta pesada que demorava 30s no Free e uns 33s no debian demora 1,5min em windows a diferença é grande.
Leandro DUTRA na lista pebr-geral

Melhor performance em instruções SQL

Artigo de Mauro Pichiliani no iMasters:

http://imasters.uol.com.br/artigo/222/sql_server/melhor_performance_em_instrucoes_sql/

Oi pessoal, nesta primeira coluna vamos falar um pouco sobre melhora de perfornance em instruções SQL. Para começar podemos entender instruções SQL por comandos do tipo SELECT, UPDATE e DELETE.

Cada comando possui uma série de detalhes, e no SQL Server, estes detalhes podem impactar muito na performance. No artigo desta semana, vou falar um pouco sobre talvez o mais utilizado deles: o SELECT.

Primeiro vamos à sintaxe básica do SELECT

SELECT <lista_de_campos> [FROM <lista_de_tabelas> [WHERE <lista_de_condições>]]
[GROUP BY <lista_de_campos>]

No lugar de <lista_de_campos> devemos colocar uma expressão que seja passível de entendimento ao SQL Server. Por exemplo:

SELECT GETDATE() – Retorna a data atual do sistema

E na lista de tabelas os campos da tabela. Exemplo:

SELECT CAMPO FROM TABELA1

Com isso , já temos uma dica de performance: procura NÃO utilizar o '*' para retornar todos os campos , pois isso traz para a estação cliente dados as vezes desnecessários.Exemplo:

SELECT * FROM TABELA1 - **Procure não fazer isso.... e sim:**

SELECT CAMPO1 , CAMPO2 , CAMPO3 FROM TABELA1

Bom , quanto à lista de filtros (cláusula where) devemos tomar muito cuidado , pois é ai que

geralmente temos problemas ou perda de performance. Para começar , evite utilizar os peradores IN() e subquery's , pois eles oneram a performance do banco.Evite também instruções muito grandes. Procure quebrá-las em várias instruções e ligue os resultados com o comando UNION.

Cuidado com os oepradores lógicos AND na cláusula WHERE pois para cada AND a mais que é colocado , todo conjunto de dados que será retornado tem que ser filtrado. Isto consome muito processamente as vezes desnecessário.

Sempre que possível procure utilizar a cláusula TOP n para indicar qual a quantidade de registro. Saber de antemão quandos registros a query tem que retornar ajuda o SQL Server a fazer um plano de execução da instrução menos e isso diminui o tempo de reposta.Por exemplo , se quisermos somente os 5 primeiros registros que antendem a uma condição:

```
SELECT TOP 5 FLAG1 FROM TABELA1 WHERE FLAG1 = 2
```

Não abuso muito do operador LIKE. Ele é otimo para consultas , mais devemos procurar não colocar % antes e depois:

- Se houver como , evite isto

```
SELECT NOME FROM TABELA1 WHERE NOME LIKE '%A%'
```

Outra dica interessante é não colocar muitos campos na cláusula ORDER BY (ordem da comsulta) , pois para cada campo adicional, temos um re-ordenação interna do conjundo de dados retornado. Por exemplo :

```
SELECT CAMPO1 , CAMPO2 , CAMPO3 FROM TABELA1
ORDER BY CAMPO1 , CAMPO2 , CAMPO3
```

procure usar (quando possível):

```
SELECT CAMPO1 , CAMPO2 , CAMPO3 FROM TABELA1
ORDER BY CAMPO1
```

Um ponto a ser levantado é o uso de joins. A ordem que se relaciona as tabelas na instrução SELECT não importa. Isto por quê internamente o SQL Server vai escolher a ordem de acesso às tabelas. Por exemplo:

```
SELECT A.CAMPO1 , B.CAMPO2 FROM TABELA1 A , TABELA2 B
WHERE
A.COD = B.COD
```

Tem um performance igual se a tabela A possuir mais registros, mesmo que não satisfaçam a condição do join , do que a order inversa :

```
SELECT A.CAMPO1 , B.CAMPO2 FROM TABELA1 A , TABELA2 B
WHERE
B.COD = A.COD
```

Lembrando que as duas instruções anteriores retornam SEMPRE o mesmo resultado.

Otimização do PostgreSQL - Introdução em:
http://www.sqlmagazine.com.br/Artigos/Postgre/02_Otimizacao.asp

PostgreSQL Leopardo em:
http://www.midstorm.org/~telles/uploads/postgresql_leopardo_conisli_2007.odp

Introdução ao Tunning do SGBD PostgreSQL em:
http://www.lozano.eti.br/palestras/tunning-pgsql.pdf

Tutorial completo sobre RAID 0, RAID 1,RAID 0+1 e RAID 5 em:
http://www.guiadohardware.net/comunidade/raid-tutorial/665151/