Database Tuning: Configurando o Interbase e o PostgreSQL

Gregório Baggio Tramontina

[1]Instituto de Computação
Universidade Estadual de Campinas (UNICAMP)

gregorio@ic.unicamp.br

Resumo

As bases de dados relacionais estão cada vez maiores e mais complexas. Além disso os sistemas de gerenciamento de bancos de dados (SGBDs) atuais necessitam responder eficientemente às operações feitas sobre seus dados. Neste contexto evidencia-se o tuning como um processo de refino dos sistemas de bancos de dados, objetivando melhorar seu desempenho. Este trabalho apresenta as linhas gerais de tuning do Interbase e PostgreSQL, com foco na configuração dos parâmetros desses dois sistemas. Discute-se os principais desses parâmetros e o impacto da modificação de seus valores. Para o PostgreSQL mostra-se também maneiras de aferir mais precisamente o comportamento do SGBD sobre suas bases de dados. Faz-se uma análise crítica dos aspectos de configuração apresentados e traça-se uma hierarquia de importância de cada um de acordo com seu propósito geral. Mostram-se resultados comparativos entre Firebird, PostreSQL, MySQL e Oracle, que indicam que, para este caso específico, o PostgreSQL tem desempenho comparativamente superior. Conclui-se que o tuning é parte do processo evolutivo de um banco de dados, e não deve ser negligenciado, merecendo um bom projeto e execução.

1. Introdução

A demanda por sistemas de gerenciamento de bancos de dados (SGBDs) cresce continuamente. Juntamente com a demanda, cresce também o volume de dados que estes sistemas devem gerenciar e a complexidade de suas aplicações. Em 2003 bancos de dados (BDs) com tamanhos de aproximadamente 30 terabytes já existiam [2], e para 2004 havia previsões de BDs com cerca de 100 terabytes [6]! Neste cenário, realizar operações eficientemente sobre estas grandes coleções de dados é questão fundamental, já que o desempenho de um SGBD nas consultas e *updates* mais comuns são a medida principal de sua eficiência [5].

O primeiro caminho para se conseguir um desempenho adequado de um sistema de bancos de dados é tomar boas decisões durante o projeto (*design*) desse sistema. Durante o projeto, considerações sobre vários aspectos são feitas, como o volume esperado de dados em cada relação (tabela) do sistema, quais consultas serão realizadas, e quais dessas consultas serão mais freqüentes. Mas o funcionamento real do sistema só pode ser conhecido quando da sua efetiva implementação, e muitas das considerações que os projetistas haviam feito durante a fase de *design* podem mostrar-se incorretas. Portanto uma fase subseqüente de ajustes do sistema, sob a luz dos dados reais sobre seu comportamento, e com o objetivo de maximizar seu desempenho, faz-se necessária. Esta fase é chamada de *database tuning* ou simplesmente *tuning*.

A fase de *tuning* de um banco de dados é um processo de refinamento que envolve modificações em vários aspectos desse BD, e que vão desde mudanças nos conceitos apreendidos nos diagramas entidade-relacionamento (ER) até a troca de hardware, passando pela configuração dos softwares que executam nesse sistema. A divisão entre as etapas de *design* e *tuning* não é precisa, já que no curso do *tuning* pode ser necessário que alteremos aspectos definidos durante o *design*. Mamakrishnan e Gerke [5] identificam 6 fases principais de *design* (análise de requisitos, projeto conceitual, projeto lógico, refinamento dos

esquemas das relações, projeto físico e projeto dos aspectos de segurança do dados) e a fase de *tuning* como a execução de ações relacionadas a essas 6 fases (não necessariamente de maneira ordenada) para aumentar o desempenho do SGBD.

Em termos didáticos, podemos dividir as ações de *tuning* em três grandes tipos: (1) refinamento do esquema das relações e as consultas/*updates* feitas no BD, (2) configuração do sistema operacional em uso e (3) configuração dos parâmetros dos SGBDs [10]. Em relação este último, muitos parâmetros podem estar envolvidos, como tamanho do *buffer pool* e área de memória reservada a procedimentos de ordenação. Este trabalho apresenta uma compilação sobre a configuração dos SGBDs Interbase e PostgreSQL, focando-se principalmente em aspectos que tenham relação direta com o desempenho desses dois sistemas. Os parâmetros aqui mostrados são considerados como pertencentes a um desses cinco tipos: relativos ao gerenciamento de memória (MEM), aos custos de CPU (CPU), ao controle de concorrência (CCT), aos aspectos da rede que interliga clientes e servidores (NET) e à configuração do planejador de consultas (PLN). Essa divisão servirá para uma análise do impacto dos tipo de parâmetro feita posteriormente. Cada parâmetro tem em sua definição a indicação de seu tipo.

As informações aqui apresentadas são retiradas principalmente da documentação do Interbase e do PostgreSQL. O trabalho organiza-se da forma a seguir. A seção 2 discute o sistema Interbase e mostra as linhas gerais de como configurá-lo. A seção 3 faz o mesmo com o PostgreSQL. Uma análise do potencial impacto de cada tipo de parâmetro é delineada na seção 4. Resultados comparativos entre esses dois SGBDs e mais o MySQL e o Oracle são discutidos na seção 5. O trabalho conclui-se na seção 6.

2. Interbase

O Interbase passou a fazer parte da família de produtos da Borland em 1992, quando esta adquiriu a Ashton-Tate. Em 2000 o código tornou-se aberto, e baseado neste código foi desenvolvido o Firebird [7]. Mas as versões subseqüentes do Interbase ainda são proprietárias e pagas. Sua última versão é a 7.1.

O Interbase objetiva ser compacto o suficiente para poder fazer parte de aplicações que rodam em recursos de hardware mais modestos. Segue uma arquitetura cliente-servidor e pode ser utilizado tanto em aplicações 2-*tier* quanto n-*tier*. Cada banco de dados Interbase é um arquivo em disco onde todos os objetos desse banco de dados são guardados [10]. Usa um esquema de controle de concorrência otimista baseado em versões.

Segundo a Borland [11], o Interbase é desenvolvido para ser um SGBD "auto regulável", ou seja, que realiza ações de *tuning* automaticamente, o que Rennhackhamp [10] chama de uma abordagem "caixa preta". Isso o habilita a trabalhar em ambientes onde não há suporte de um *database administrator* (DBA) ou profissional de TI. Fica então um conjunto de parâmetros configuráveis, que reside em um arquivo chamado *ibconfi g*, queé encontrado no diretório onde o Interbase foi instalado.

Esta seção apresenta os principais parâmetros configuráveis do Interbase, com foco naqueles que definem o modo como o SGBD interage com discos, memória, escalabilidade, e outros que implicam diretamente em mudança de desempenho. Essas informações são retiradas principalmente de um *white paper* da própria Borland, específico sobre a configuração do Interbase [1].

2.1. Parâmetros de Configuração do Servidor Interbase

DATABASE_CACHE_PAGES (MEM)

Cada banco de dados "carregado" pelo Interbase tem um conjunto de buffers em memória usados como um *cache* para seus dados. Cada buffer tem o tamanho exato de uma página do disco. Este

parâmetro controla quantos desses buffers serão alocados a cada banco de dados.

Todos os bancos de dados "servidos" pelo servidor terão para si a quantidade de buffers especificados neste parâmetro. Para configurar um número diferente de buffers para um banco de dados específico é necessário utilizar a ferramenta `gfix` do Interbase. Isso pode ser feito com o seguinte comando:

```
gfix -buffers <number_of_pages> <database_name>
```

onde `<number_of_pages>` é o número de buffers desejado para o banco de dados especificado em `<database_name>`. Isso fará com que o servidor use o valor especificado nesse comando ao invés daquele contido em DATABASE_CACHE_PAGES. O tamanho das páginas é tipicamente 1Kb, 2Kb, 4Kb ou 8Kb.

Para bancos de dados menores pode ser útil aumentar a quantidade de buffers já que se pode fazer com que estes caibam inteiramente em memória e evitar *swapping* de dados com o disco. No entanto, muitos buffers disponíveis a todos os bancos podem esgotar a memória e forçar operações de *swapping* indesejadas.

CPU_AFFINITY (CPU)

Específico para as versões para Windows e útil em ambientes onde se tem mais de um processador disponível (ambientes multiprocessados ou SMP). Serve para identificar quais dessas CPUs efetivamente executarão o servidor. Assume um valor inteiro que representa um vetor de bits onde, para cada CPU (na ordem inversa do vetor), um bit 1 significa que ela pode rodar o servidor, e um bit 0 significa o contrário. Por exemplo, em um sistema com 4 CPUs, caso se deseje que apenas as CPUs 1 e 2 executem o servidor, deve-se definir o este parâmetro como "`CPU_AFFINITY 3`", pois 3 em binário é 0011.

O Interbase é licenciado para um determinado número de CPUs. Ou seja, para especificar mais de uma CPU para a execução do SGBD, é necessário adquirir um número de licenças de CPU suficiente da Borland.

O aumento de CPUs que executam o servidor pode aumentar o desempenho do SGBD como um todo. As CPUs indicadas nesse parâmetro também se tornam as únicas a executar o servidor. Isso pode ser utilizado para contornar uma característica de sistemas Windows, que faz com que *threads* de um servidor sejam trocadas entre processadores quando apenas uma CPU é licenciada para o Interbase mas nenhuma é especificamente dedicada ao banco de dados, gerando um overhead desnecessário: faz-se com que apenas uma CPU do sistema execute o servidor Interbase, por exemplo, colocando-se um valor 2 (0010) neste parâmetro.

ENABLE_HYPERTHREADING (CPU)

Também específico para sistemas Windows, indica se o Interbase deve usar o *Hyper-Threading* (HT), uma tecnologia introduzida nos processadores mais recentes da Intel. Ela possibilita que um processador físico execute duas *threads* ao mesmo tempo (dependendo, claro, de quão independentes essas *threads* venham a ser entre si), funcionando como 2 processadores lógicos. Segundo a Intel, ganhos de desempenho de até 70% podem ser atingidos (para mais informações, visite http://www.intel.com).

Dependendo das *threads* do servidor Interbase que estão executando, pode haver ganhos de desempenho habilitando-se o uso de HT. Este parâmetro também pode ser combinado com o valor de CPU_AFFINITY, para que se escolha qual processador "lógico" de cada processador "físico" será usado para rodar o servidor. Deve-se lembrar no entanto que a tecnologia HT depende de suporte de sistema operacional e que mesmo o Windows 2000, segundo a própria Microsoft, não é completamente adaptado a ele (o Windows 2003 já tem suporte completo a HT).

DEADLOCK_TIMEOUT (CCT)

Um valor inteiro que indica quantos segundos o SGBD deve esperar para fazer a próxima checagem da ocorrência de deadlocks entre as transações ativas sobre o banco de dados. Em sistemas com alta concorrência pode ser útil aumentar esse valor para que as transações possam utilizar melhor os recursos de máquina que seriam tomados por cada execução da identificação de deadlocks.

DUMMY_PACKET_INTERVAL (NET)

Em um ambiente onde cliente e servidor estão conectados em rede, pode haver longos tempos durante os quais o cliente não troca dados com o servidor. Assim não há como saber quando um cliente teve problemas e sua conexão caiu. Este parâmetro é o tempo, em segundos, entre o envio de um *dummy packet* do servidor para o cliente.

O servidor envia um *dummy packet* ao cliente e caso este responda com um *acknoledgement* (ACK, ver RFC 793 [8]), considera-se o cliente como ativo. Caso não haja resposta, o servidor pode considerar o cliente como inativo e terminar a conexão, abrindo espaço para outro cliente. O valor default desse parâmetro é de 60 segundos. Em sistemas com muitas conexões de rede entre clientes e servidores, pode ser útil detectar os clientes inativos mais rápido e resolver problemas como longas esperas por conexão com o servidor.

MAX_THREADS (CCT/CPU)

A partir da versão 7.0, o Interbase suporta a execução de mais de uma *thread* em paralelo, em sistemas multiprocessados. Esse parâmetro especifica o número máximo de *threads* que podem ser executadas dessa maneira. Em sistemas de concorrência alta, mais *threads* podem aumentar o desempenho, embora quando se tem apenas um processador, a sugestão é deixar este parâmetro com o valor 1. Esse é inclusive seu valor default para esses sistemas, enquanto que para sistemas multiprocessados esse valor é 1 milhão.

ANY_LOCK_MEM_SIZE (CCT)

O Interbase mantém para cada banco de dados ativo uma tabela para guardar as requisições de *lock* de objetos. Em sistemas com grande concorrência pode ser útil aumentar o tamanho de memória reservado para esta tabela, e isso é feito através da edição deste parâmetro. Para sistemas Windows, seu valor default é de 256Kb, e em sistemas Unix/Linux, 96Kb.

SERVER_CLIENT_MAPPING (MEM)

Em sistemas Windows, cliente e servidor podem comunicar-se através de um mecanismo chamado *interprocess communication* ou IPC. Ele permite que dois processos que estejam em uma mesma máquina compartilhem uma área de memória através da qual podem trocar dados. Esse parâmetro permite que se configure quanto de memória será utilizado como um buffer para tal mecanismo. Para cada banco de dados ativo, o espaço indicado em SERVER_CLIENT_MAPPING é alocado, e para cada cliente conectado a um desses bancos de dados, um pedaço desse espaço é alocado quando este se conecta com o servidor. Mais espaço significa que mais dados podem ser passados de um lado a outro por vez. Os valores possíveis neste caso são 1024, 2048, 4096 e 8192 bytes.

2.2. Parâmetros de Configuração do Cliente Interbase

CONNECTION_TIMEOUT (NET)

Tempo que o cliente espera até assumir que o servidor não aceitou seu pedido de conexão. Se o servidor não responde ao pedido dentro do tempo especificado neste parâmetro, o cliente retorna um erro para a aplicação.

TCP_REMOTE_BUFFER (NET)

O Interbase permite conexão entre cliente e servidor com vários protocolos de transporte, como NETBeui e TCP. No caso do TCP, sockets são usados para montar essas conexões. Esses sockets possuem buffers para guardar os dados sendo transmitidos. Este parâmetro permite configurar o tamanho desses buffers, e é utilizado tanto pelo cliente quanto pelo servidor.

Com buffers maiores pode-se aumentar a velocidade de transmissão de dados entre cliente e servidor em uma rede (desde, é claro, que esta suporte o maior volume de dados). Mas esse parâmetro atinge todos os sockets dedicados ao Interbase na máquina, e caso existam muitas conexões simultâneas, recursos preciosos do sistema como memória podem tornar-se escassos, levando à perda de desempenho. É aconselhável a mudança cuidadosa desse parâmetro.

2.3. O Server Manager

Nos sitemas Windows o Interbase traz um utilitário para pequenas configurações do software servidor, chamado de *Server Manager* (também conhecido como *Interbase Manager*). Pode ser encontrado no "Painel de Controle" do Windows, e permite que se altere os parâmetros DATABASE_CACHE_PAGES e SERVER_CLIENT_MAPPING. Também mostra informações sobre a versão do software servidor, licenças e número de conexões e de bancos de dados ativos. Mas mesmo assim é limitado permitindo a alteração, por interface gráfica, de apenas dois parâmetros.

3. PostgreSQL

O PostgreSQL é um SGBD objeto-relacional de código aberto derivado do projeto POSTGRES, da Universidade de Berkeley [3]. Atualmente é desenvolvido por um grupo que envolve empresas e pessoas de várias partes do mundo. Está na versão 7.4 e é direcionado à plataformas Unix/Linux. Usa um sistema de locking chamado *Multiversion Concurrency Control*, ou MVCC. Informações completas sobre o projeto podem ser encontradas em seu site oficial [9].

O PostgreSQL é muito mais configurável que o Interbase. Existem mais parâmetros que podem ser mudados pelo DBA, mais maneiras de se perguntar o que e como o sistema está trabalhando e mais meios de se interferir neste comportamento. Esta seção discute alguns aspectos do *tuning* do PostgreSQL.

3.1. Monitorando a Atividade do PostgreSQL

Pode-se monitorar a atividade do PostgreSQL através das ferramentas padrão do Unix/Linux, como o `ps` e o `top`. Isso porque cada transação ativa nesse SGBD é executada por um processo de sistema individual. Um exemplo do uso do comando `ps` pode ser visto a seguir (alguns resultados intermediários foram omitidos por questões de espaço).

```
$ ps auxww | grep ^postgres
postgres   960  (...) postmaster -i
postgres   963  (...) postgres: stats buffer process
postgres   965  (...) postgres: stats collector process
postgres   998  (...) postgres: tgl runbug 127.0.0.1 idle
postgres  1003  (...) postgres: tgl regression [local] SELECT waiting
postgres  1016  (...) postgres: tgl regression [local] idle in transaction
```

No resultado desse comando estão listados todos os processos pertencentes ao PostgreSQL (usuário `postgres`): o servidor (960), dois processos que implementam o *statistics collector* (963 e 965), e mais três processos que tratam de transações de clientes individuais. Estas três últimas linhas mostram, além das transações, o estado atual de cada uma. Existem três estados possíveis: *idle* (esperando por um comando do usuário), *idle in trasaction* (esperando por um comando do usuário dentro de uma transação iniciada), e um comando (como o SELECT mostrado). O termo *waiting* é acrescentado quando uma transação está esperando que outra libere algum lock. Neste caso pode-se inferir que a transação executada pelo processo 1003 está esperando que outra, executada pelo processo 1016, libere um ou mais locks. A ferramenta `ps` pode ser muito útil para se levantar informações sobre a carga de trabalho do sistema e direcionar os próximos trabalhos durante a fase de *tuning*.

3.2. O Statistics Collector

O *statistics collector* é um processo que coleta informações estatísticas sobre o funcionamento do servidor PostgreSQL. Dados como o número de acessos às relações e aos índices podem ser obtidos através de *views* predefinidas especificamente para este fim. Existem muitas dessas *views* e também é possível criar outras através de funções de leitura desses dados, disponibilizadas pelo SGBD . Mostrar todos esses aspectos foge ao escopo deste texto. Para uma referência completa, consulte a documentação do PostgreSQL [3].

Uma informação é particularmente interessante: estatísticas sobre os índices das relações podem indicar o número de *index scans* feitos com cada índice e número de tuplas retornadas nestes *scans*, por exemplo. Tais informações podem ajudar a identificar quais índices estão sendo realmente utilizados e quão eficientes eles são.

3.3. O Comando EXPLAIN

Pode-se perguntar para o PostgreSQL exatamente qual plano é executado para uma determinada consulta SQL através do comando EXPLAIN. Um exemplo de uso desse comando é dado a seguir

```
EXPLAIN SELECT * FROM r1 WHERE un < 50;

                                  QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using r1_un on r1  (cost=0.00..179.33 rows=49 width=148)
   Index Cond: (un < 50)
```

Supondo-se a existência de um índice com árvore B+ no atributo `un`, o plano mostrado executa um *index scan* na relação `r1` utilizando este índice, com a condição `un < 50`. Tem um custo estimado entre 0 e 179.33 leituras de páginas do disco, retornando aproximadamente 49 tuplas com tamanho igual a 148 bytes.

Saber qual plano é executado para uma dada consulta permite analisar principalmente se o SGBD realiza tal operação como o DBA espera. Por exemplo, pode-se decidir pela remoção de um índice que não pode ser utilizado em uma consulta importante e que demande esforço extra para ser mantido no

SGBD. Podem vir à tona também algumas limitações do sistema, como uma possível falta de suporte a *index scans* mesmo que todas as condições para isso estejam presentes.

3.4. VACUUM e ANALYZE

Para estimar os custos de um plano de consulta, o PostgreSQL usa informações do catálogo do sistema, que possui dados como o número de tuplas por relação e o número de páginas que a relação ocupa no disco, por exemplo. Manter esse catálogo atualizado permite que boas estimativas sobre os planos sejam feitas, melhorando o desempenho dos sistemas na execução de comandos SQL. Essa atualização é feita pelo comando ANALYZE. Caso não seja passado nenhum parâmetro para esse comando, ele realiza a atualização do catálogo para todas as relações do sistema. Caso um nome de relação seja dado, apenas as informações sobre esta relação são atualizadas.

Além disso, quando se remove ou se altera tuplas das relações do PostgreSQL, elas não são diretamente retiradas das relações. Apenas uma nova versão da tupla, visível para as transações mais recentes, é criada. Isso porque o controle de concorrência desse SGBD, o MVCC, exige que se mantenha diferentes versões de um mesmo objeto, já que a versão anterior de uma tupla alterada por um transação mais recente ainda é potencialmente visível para uma transação mais antiga. O comando VACUUM realiza a remoção das tuplas mais antigas para três fins: recuperação de espaço em disco, atualização do catálogo do sistema (conjuntamente com o ANALYZE, utilizando a sintaxe VACUUM ANALYSE) e prevenção do problema de ciclos de identificadores (IDs) de transações (*transaction ID wraparound*, ver [3]).

3.5. A View pg_locks

A *view* pg_locks permite que se obtenha informações sobre a situação dos locks no banco de dados. Essa view possui os seguintes campos: *relation* (identificador da relação que está bloqueada por este lock), *database* (identificador do banco de dados onde a relação indicada em *relation* existe), *transaction* (identificador da transação que detém este lock), *mode* (modo de lock que a transação deseja), e *granted* (valor booleano; *true* indica que o lock foi dado à transação, e *false* indica que esta aguarda pelo lock).

Adquirir informações sobre os locks de um banco de dados pode guiar esforços de *tuning* do mecanismo de controle de concorrência de forma a aumentar o número de transações concorrentes sem afetar a consistência dos dados e portanto aumentar o *throughput* do sistema como um todo.

3.6. Parâmetros de Configuração do PostgreSQL

Os parâmetros de configuração do PostgreSQL residem em um arquivo chamado *postgresql.conf*. Cada linha desse arquivo especifica um parâmetro. Cada parâmetro assume um valor inteiro, booleano, de ponto flutuante ou string. Para mudar o valor de um parâmetro pode-se editar o arquivo ou passá-lo na linha de comando para o servidor PostgreSQL quando este é iniciado. Aqui discute-se alguns desses parâmetros.

max_connections (NET)

Especifica o número máximo de conexões de clientes ao servidor PostgreSQL. Mais conexões podem aumentar o *throughput* do sistema, mas necessitam de mais memória e estruturas de controle de acesso do sistema operacional, como semáforos, e portanto de mais recursos computacionais. Seu valor default é 100.

shared_buffers (MEM)

Controla o número de buffers de memória compartilhada utilizados pelo servidor. Cada buffer possui 8192 bytes (8Kb), a não ser que um tamanho diferente tenha sido especificado em BLCKSZ durante a compilação do servidor. Um valor mais alto nesse parâmetro pode significar maior desempenho, embora um valor muito grande possa trazer resultados inversos. 1000 é o valor default.

sort_mem (MEM)

Especifica a quantidade de memória alocada para operações de ordenação. Com mais memória é possível ordenar tuplas de uma relação em menos passos, mas mais recursos do sistema serão consumidos: se houver X ordenações acontecendo, cada uma receberá sort_mem Kb de memória, e portanto a memória total ocupada será de $X \times$ sort_mem. O valor é especificado em Kilobytes, e o defaul é 1024Kb (1Mb).

fsync (MEM)

Um valor booleano que, quando verdadeiro, determina que o sistema deve realizar a chamada de sistema `fsync()` em vários pontos da execução das transações para garantir que os dados modificados sejam escritos no disco. Manter este parâmetro ligado causa uma degradação no desempenho, já que quando ele está desabilitado o sistema operacional pode tentar fazer o seu melhor em termos de *buffering*, ordenação e retardo das escritas em disco. No entanto, desabilitá-lo deixa o sistema vulnerável a problemas de persistência nos dados, já que até operações de escrita do log em disco (que seguem o protocolo WAL) são deixadas a cargo do sistema operacional. A documentação do PostgreSQL [3] recomenda que "se você confia no seu sistema operacional, hardware e empresa de suporte (ou baterias de backup), considere desabilitar este parâmetro".

enable_hashagg (PLN)

Valor booleano. Especifica se o planejador pode fazer o cálculo de operações agregadas (agregação) utilizando tabelas hash.

enable_hasjoin, enable_mergejoin, enable_nestloop (PLN)

Habilita ou desabilita o uso de planos de consulta com junções feitas através de *hash join*, *merge join* e *nested loops*, respectivamente.

enable_indexscan (PLN)

Liga ou desliga o uso, por parte do planejador, de *index scans* nas relações, quando possível.

cpu_tuple_cost, cpu_index_tuple_cost, cpu_operator_cost (PLN/CPU)

Valores de ponto flutuante que especificam o custo de de CPU assumido pelo planejador para processar, respectivamente, uma tupla de uma relação, uma entrada de índice em um *index scan*, e um operador na cláusula WHERE de uma consulta. Esse custo é dado por uma fração do custo de ler páginas seqüen-

cialmente do disco. Os valores default são, em ordem, 0.01, 0.001, e 0.0025.

geqo (PLN)

O PostgreSQL também impementa um algoritmo genético para otimizar as consultas feitas no servidor. Este parâmetro é um valor booleano que habilita ou desabilita o uso de otimização genética. O valor default é ligado.

default_transaction_isolation (CCT)

Cada transação tem um nível de isolamento, que no PostgreSQL pode ser *read commited* ou *serializable*. Esse parâmetro especifica qual o nível de isolamento padrão para toda nova transação. *Read commited* pode fornecer maior concorrência, mas está sujeito a problemas de "fantasma" (*phantom problem*).

deadlock_timeout (CCT)

Tempo, em milissegundos, que uma transação espera por um lock antes de se fazer uma checagem de ocorrência de deadlocks. Em sistemas com grande carga de trabalho pode ser útil aumentar esse tempo, para diminuir o tempo gasto pelo PostgreSQL para realizar essa verificação e tomar menos recursos do sistema. Em um cenário ideal esse valor seria maior que o tempo médio de uma transação, para se aproveitar do fato de que os locks serão provavelmente liberados antes que se decida pela verificação. O valor default é 1000 (1 segundo).

max_locks_per_transaction (CCT)

Número máximo de locks que uma transação pode adquirir simultaneamente. O valor default é 64. Esse valor é geralmente suficiente [3], mas para sistemas com transações que tocam muitos objetos no banco de dados, pode ser necessário aumentar esse valor.

4. Análise dos Aspectos de Configuração Considerados

Cada parâmetro de configuração pode ter uma influência diferente no desempenho do SGBD dependendo primeiro do que ele pode mudar no comportamento deste último e depois das características do cenário onde o sistema está sendo utilizado. Esta seção faz uma análise dos tipos de parâmetros considerados até aqui e tenta traçar uma hierarquia de impacto desses tipos, baseado no propósito geral de cada um. Essa hierarquia não pretende desqualificar nenhum quesito analisado, de forma que uma classificação "mais baixa" não quer dizer "sem importância".

4.1. Gerenciamento de Memória (MEM)

Em primeiro lugar estão os parâmetros que influenciam o gerenciamento de memória, identificados como MEM. A memória RAM é um recurso precioso dos sistemas computacionais, e pode ser acessada muito mais rapidamente do que um disco. Idealmente, os dados do disco são carregados apenas uma vez para a memória e só voltam para o primeiro no fim das operações feitas sobre eles. Um melhor gerenciamento de memória permite que muito menos acessos a disco sejam feitos. Isso tem impacto

direto no desempenho de um SGBD, já que acesso a disco é uma operação considerada cara comparada até mesmo ao custo de CPU.

A memória também é sempre um aspecto importante a se considerar em um sistema de bancos de dados. Podem existir sistemas com baixa ou nenhuma concorrência, e de pouco ou nenhum uso de rede, mas mesmo um BD monousuário pode ter muitos dados e consultas complexas. Portanto deve-se sempre considerar aspectos de memória quando do *tuning* do SGBD.

Embora não diretamente relacionadas com o objetivo desse texto, o aumento da quantidade de memória RAM, e mudanças na memória secundária (discos) do sistema, como a utilização de discos mais rápidos (SATA, por exemplo) e técnicas de espelhamento (RAID), podem trazer ganhos significativos de desempenho.

4.2. CPU e Planejador (CPU e PLN)

Logo mais abaixo do gerenciamento de memória, mas bem próximos, vêm os parâmetros para configurar a interação do SGBD com a(s) CPU(s) a ele disponível(is) e a atuação do planejador.

Com um maior poder de processamento na CPU, as transações de um SGBD podem ficar mais rápidas, diminuindo o tempo que um cliente espera para que a operação por ele requisitada seja completada e aumentando o *throughput* do sistema. Operações que necessitam mais intensamente de uso de CPU (*CPU-intensive*) também se beneficiam, como ordenações e cálculos de funções agregadas. Há ainda a possibilidade de se aumentar o número de transações atendidas simultaneamente pelo servidor, mas com o custo de cada transação individual ficar potencialmente mais lenta.

Apenas mais poder de CPU, no entanto, não garante ganhos máximos de desempenho já que um planejador mal configurado pode, para cada consulta/*update* de uma transação, gerar planos suficientemente ruins, que acabam por "frenar" o sistema. Uma boa configuração do planejador, com estatísticas atualizadas sobre o sistema, coopera para o desempenho geral do SGBD.

4.3. Controle de Concorrência (CCT)

Em terceiro plano vêm a configuração do controle de concorrência. Isso porque um SGBD pode enfrentar situações de altíssima carga de trabalho, com muitas transações simultâneas, como também pode ter uma carga baixa e poucas transações. Dependendo da semântica dos dados do BD, as transações também podem ser independentes o suficiente para que problemas de concorrência sejam mínimos, mesmo com altas cargas de trabalho.

As configurações padrão de um SGBD podem ser suficientes para sistemas onde problemas de concorrência são baixos. Sistemas com alta concorrência podem beneficiar-se de configurações que permitam um menor nível de isolamento das transações, e que podem portanto ser executadas mais rápido e aumentar o *throughput* do sistema.

4.4. Interação com a Rede

No quarto posto vem os parâmetros de configuração da interação rede-SGBD. No porque um melhor uso da rede por parte do SGBD não seja importante, mas porque a rede em si pode ser um gargalo para o sistema, e pode mostrar-se mais vantajoso melhorar a estrutura de rede como um todo, como o aumento da capacidade de um link, uso de *switches* ao invés de *hubs* e recabeamento, do que apenas configurar o SGBD.

As redes de computadores atuais têm melhorado constantemente e hoje estão disponíveis tecnologias

como *Virutal Private Networks* (VPN), conexões ADSL de usuários domésticos e links corporativos com grande capacidade de transferência de dados. Para BDs internos a uma empresa o limite é a capacidade da rede local desta última, que costuma ser relativamente grande. A grande questão está nos bancos de dados distribuídos e/ou disponibilizados na Internet, e que enfrentam possivelmente um grande número de acessos e transações simultâneas. Para estes, a configuração do SGBD é mais importante, mas uma boa estrutura de rede é ainda fundamental.

4.5. ANALYZE, ps, EXPLAIN e VACUUM no PostgreSQL

Os comandos ANALYZE, EXPLAIN e VACUUM, juntamente com a ferramenta `ps`, também têm importância para bancos de dados em PostgreSQL. Pode-se destacar em primeiro plano o ANALYZE, cuja missão é atualizar as informações estatísticas utilizadas pelo planejador para estimar os custos das consultas. Manter essas informações atualizadas é de grande importância, e por isso a execução regular desse comando ajuda a manter o desempenho do SGBD em um bom patamar. Guimarães [4] comenta que o uso do ANALYZE melhorou consideravelmente o desempenho do PostgreSQL nos testes por ele realizados (estes testes serão comentados mais adiante).

O uso do `ps` e do EXPLAIN vêm em segundo plano pois destinam-se a verificar o comportamento do sistema sobre situações mais específicas, e portanto podem ser mais úteis quando algum problema de desempenho está acontecendo no sistema. Mas são importantes para este fim e seus resultados podem guiar esforços de *tuning* futuros.

No terceiro posto fica o VACUUM. Este comando pode ser executado menos vezes que o ANALYZE e não é um meio direto de aferir o comportamento do sistema. Mas também tem seu lugar pois considerações sobre espaço em disco em bancos de dados continuamente atualizados são de suma importância: deve-se recuperar o espaço ocupado pelos objetos mais antigos para abrir espaço para os mais recentes, além de amenizar ou retardar a compra de mais hardware por causa de falta de espaço nos discos atuais.

5. Alguns Resultados Comparativos

O trabalho de Guimarães [4] apresenta um teste comparativo de SGBDs quanto a consultas sobre uma base de dados padrão. Os SGBDs são avaliados com base no seu tempo de resposta a consultas feitas nessa base de dados. Como o teste inclui o Firebird (derivado do Interbase) e o PostgreSQL, ele é comentado aqui.

O experimento foi feito para quatro SGBDs bem conhecidos: Firebird 1.0, PostgreSQL 7.2, MySQL 3.2 e Oracle 9i. A máquina utilizada foi um Pentium 4 2.0 GHz com 512 Mb de memória RAM e com sistema operacional Red Hat Linux 7.3. A base de dados implementada nesses SGBDs para os testes é a *Winestore*, proposta por Williams em [12]. Ela é composta das relações especificadas na tabela 1. O número de tuplas existentes nas relações, e o número de atributos por tupla são também mostrados nesta tabela. Também existem índices em vários atributos dessas relações. Para uma descrição completa desse esquema, refira-se ao apêndice A.

Todos os SGBDs foram analisados de acordo com uma única consulta, que realiza a junção de todas as relações da base de dados e que possui 9 condições na cláusula WHERE. Esta consulta é mostrada a seguir.

Relação	Tuplas	Atributos
customer	650	17
grape_variety	21	2
inventory	1049	6
items	7780	7
orders	2254	6
region	9	4
wine	1048	6
winery	300	6
wine_variety	1553	3

Tabela 1: Relações, número de tuplas e atributos

```
SELECT COUNT(*)
FROM wine W, inventory I, region R, grape_variety GV, wine_variety WV,
     customer C, items IT, orders O, winery WY
WHERE
(1)  I.wine_id = W.wine_id AND
(2)  W.winery_id = WY.wynery_id AND
(3)  WY.region_id = R.region_id AND
(4)  WV.wine_id = W.wine_id AND
(5)  WV.variety_id = GV_variety_id AND
(6)  IT.cust_id = C.cust_id AND
(7)  IT.order_id = O.order_id AND
(8)  IT.wine_id = W.wine_id AND
(9)  O.cust_id = C.cust_id
```

Cada condição da cláusula WHERE foi numerada para fins de referência. Para cada SGBD, essa consulta foi realizada com todas essas condições ou removendo-se uma delas. Portanto tem-se 10 consultas diferentes. O comando ANALYZE do PostgreSQL foi executado para a tomada dos resultados, sendo que isso melhorou substancialmente os tempos de resposta desse SGBD [4]. Note que a consulta mostra apenas o número de tuplas retornadas para as condições especificadas. O tempo, em segundos, que cada SGBD levou para responder a essa consulta é mostrado na tabela 2 (melhores tempos em negrito).

Cond. retirada	Tuplas resultantes	Firebird	PostgreSQL	MySQL	Oracle
Nenhuma	11.516	0,4	**0,2**	0,9	0,4
1	12.069.794	22	17.7	129	**1,8**
2	3.454.800	18	**5,2**	6,9	10
3	103.644	0,8	**0,3**	2,4	0,7
4	12.097.870	81	**18**	52	19
5	241.836	0,8	**0,5**	2,1	0,7
6	5.280.144	6,6	**5,3**	31,3	118
7	49.741	0,6	**0,2**	0,4	1,2
8	12.090.120	371	**12,1**	15,5	165
9	5.303.176	**3,9**	5,5	25	116

Tabela 2: Tempos (s) de retorno das consultas para os SGBDs

Nota-se que o PostgreSQL, depois do comando ANALYSE, mostrou resultados comparativamente superiores. Vê-se também que para cada tipo de consulta há variações, às vezes expressiva, nos tempos

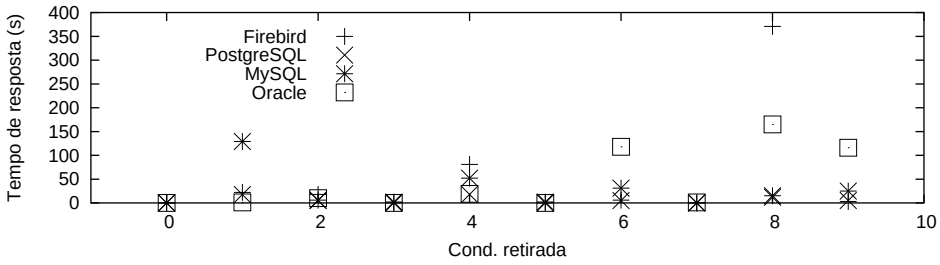

Figura 1: Tempos dos SGBDs para cada consulta

de resposta obtidos por cada SGBD. Isso pode ser explicado pelas diferentes implementações desses sistemas, o que leva a resultados também diferentes no desempenho dependendo de como cada um planeja e executa as consulta.

A figura 1 mostra uma representação gráfica dos tempos dos SGBD para cada condição retirada (o valor 0 indica o caso de nenhuma condição retirada).

6. Conclusão

O *tuning* é uma etapa importante durante o ciclo de vida de um banco de dados. Mais do que uma tarefa isolada, é um desdobramento do processo evolutivo pelo qual o BD passa para refletir sempre aquilo que se espera em termos da lógica organizacional nele representada e desempenho.

Os modos de lidar com *tuning* do Interbase e PostgreSQL são diferentes e mostram duas tendências. O primeiro, com sua abordagem "caixa preta", deixa pouco para ser diretamente configurado, e almeja adaptar-se dinamicamente às mudanças do sistema em que atua, necessitando de pouca manutenção e liberando em muito o trabalho de um DBA. O segundo, muito mais maleável, pretende deixar abertas várias escolhas, para poder ser guiado da melhor forma possível pelo DBA. A utilização de uma ou outra opção depende do cenário de atuação de cada uma.

Nos aspectos de configuração, o Interbase mostra-se mais simples, com uma documentação menor mas não por isso incompleta. As principais atividades de *tuning* são o gerenciamento de memória utilizada, o controle de execução concorrente de *threads* e o controle de conexões cliente-servidor. O PostgreSQL também permite mudanças nestes três aspectos, mas também deixa que se altere parâmetros como a quantidade de memória para operações de ordenação (importantes e também custosas computacionalmente), o uso de algoritmos específicos para junções e o uso de chamadas de sistema para forçar dados de memória em disco. Também permite, através do comando `EXPLAIN`, determinar exatamente o plano executado pelo SGBD para um comando SQL específico.

É importante que se faça o *tuning* de sistemas de bancos de dados. Deve-se inclusive considerar tal tarefa como parte do processo de desenvolvimento do SGBD, ao invés de simplesmente considerá-la como manutenção, planejando-a desde o início e a realizando cuidadosamente.

Referências

[1] Sriram Balasubramanian. Tuning Borland Interbase configuration parameters. Borland White Paper, 2003.

[2] Winter Corporation. 2003 TopTen Award Winners. Obtido via Internet em 18/06/2003, em http://www.wintercorp.com/vldb/2003_TopTen_Survey/TopTenWinners.asp, 2004.

[3] The PostgreSQL Global Development Group. PostgreSQL 7.4.2 documentation. Obtido via Internet em 14/06/2004, em http://www.postgresql.org/docs/pdf/7.4/postgresql-7.4.2-US.pdf, 2003.

[4] Célio Cardoso Guimarães. Benchmark envolvendo junções de tabelas para sgbds relacionais. Instituto de Computação (IC) - Universidade Estadual de Campinas (UNICAMP). Obtido via Internet em 15/06/2004, em http://artemis.ic.unicamp.br/ celio/benchmark.html, 2002.

[5] Raghu Mamakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 3rd. edition, 2002.

[6] Joe McKendrick. Make room for the monster databases. *Database Trends and Applications*, 15(12), 2002.

[7] Firebird Site Oficial. http://firebird.sourceforge.net/.

[8] J. Postel. Request for comments 793 - Transmission Control Protocol. The Internet Engineering Task Force, 1981.

[9] PostgreSQL Project. Site oficial. http://www.postgresql.org.

[10] Martin Rennhackhamp. Performance tuning - offerings by leading DBMS vendors for database performance tuning. *DBMS Online*, 1996.

[11] Borland USA. Interbase - site oficial. http://www.borland.com/interbase/.

[12] Hugh E. Williams and David Lane. *Web database applications with PHP & MySQL*. O'Reilly & Associates, 2002.

A. Esquema das Relações da Base de Dados Winestore

```
- create_tables.sql            BD Winestore
-- versão Postgres  Obs: bytea equivale a blob

CREATE TABLE users (
  cust_id int DEFAULT '0' NOT NULL,
  user_name varchar(50) DEFAULT '' NOT NULL,
  password varchar(15) DEFAULT '' NOT NULL,
  PRIMARY KEY (user_name)
);
   CREATE INDEX password on users(password);

CREATE TABLE customer (
  cust_id  serial,
  surname varchar(50) DEFAULT '' NOT NULL,
  firstname varchar(50) DEFAULT '' NOT NULL,
  initial char(1),
  title varchar(10),
  addressline1 varchar(50) DEFAULT '' NOT NULL,
  addressline2 varchar(50),
  addressline3 varchar(50),
  city varchar(20) DEFAULT '' NOT NULL,
  state varchar(20),
  zipcode varchar(5),
  country varchar(20),
  phone varchar(15),
  fax varchar(15),
  email varchar(30) DEFAULT '' NOT NULL,
  birth_date date DEFAULT '0000-01-01' NOT NULL,
  salary int DEFAULT '0' NOT NULL,
  PRIMARY KEY (cust_id)
);

  CREATE INDEX fullname on customer(surname,firstname);
```

```
CREATE TABLE grape_variety (
  variety_id serial,
  variety varchar(50) DEFAULT '' NOT NULL,
  PRIMARY KEY (variety_id)
);

  CREATE INDEX var on grape_variety(variety);

CREATE TABLE inventory (
  wine_id int DEFAULT '0' NOT NULL,
  inventory_id int DEFAULT '0' NOT NULL,
  on_hand int DEFAULT '0' NOT NULL,
  cost numeric(5,2) DEFAULT '0.00' NOT NULL,
  case_cost numeric(5,2) DEFAULT '0.00' NOT NULL,
  date_added timestamp,
  PRIMARY KEY (wine_id,inventory_id)
);

CREATE TABLE items (
  cust_id int DEFAULT '0' NOT NULL,
  order_id int DEFAULT '0' NOT NULL,
  item_id int DEFAULT '1' NOT NULL,
  wine_id int DEFAULT '0' NOT NULL,
  qty int,
  price numeric(5,2),
  date timestamp,
  PRIMARY KEY (cust_id,order_id,item_id)
);

CREATE TABLE orders (
  cust_id int DEFAULT '0' NOT NULL,
  order_id int DEFAULT '0' NOT NULL,
  date timestamp,
  discount numeric(3,1) DEFAULT '0.0',
  delivery numeric(4,2) DEFAULT '0.00',
  note varchar(120),
  PRIMARY KEY (cust_id,order_id)
);

CREATE TABLE region (
  region_id serial,
  region_name varchar(100) DEFAULT '' NOT NULL,
  description bytea,
  map bytea,
  PRIMARY KEY (region_id)
);

  CREATE INDEX region_name on region(region_name);

CREATE TABLE wine (
  wine_id serial,
  wine_name varchar(50) DEFAULT '' NOT NULL,
  type varchar(10) DEFAULT '' NOT NULL,
  year int DEFAULT '0' NOT NULL,
  winery_id int DEFAULT '0' NOT NULL,
  description bytea,
  PRIMARY KEY (wine_id)
);

  CREATE INDEX wine_name on wine(wine_name);
  CREATE INDEX winery_id on wine(winery_id);

CREATE TABLE wine_variety (
  wine_id int DEFAULT '0' NOT NULL,
  variety_id int DEFAULT '0' NOT NULL,
  id int DEFAULT '0' NOT NULL,
  PRIMARY KEY (wine_id,variety_id)
);
```

```
  CREATE INDEX wine_id on wine_variety(wine_id,variety_id);

CREATE TABLE winery (
  winery_id serial,
  winery_name varchar(100) DEFAULT '' NOT NULL,
  region_id int DEFAULT '0' NOT NULL,
  description bytea,
  phone varchar(15),
  fax varchar(15),
  PRIMARY KEY (winery_id)
);

  CREATE INDEX winery_name on winery(winery_name);
  CREATE INDEX region_id on winery(region_id);
```