PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 20/06

Estado da Arte em Auto-Sintonia de Transações

José Maria Monteiro
Sérgio Lifschitz
Ângelo Brayner

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 20/06 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Julho, 2006

Estado da Arte em Auto-Sintonia de Transações *

José Maria Monteiro, Sérgio Lifschitz, Ângelo Brayner[1]

[1]Mestrado em Informática Aplicada – Universidade de Fortaleza (UNIFOR)

monteiro@inf.puc-rio.br, sergio@inf.puc-rio.br, brayner@unifor.br

Abstract.

Database Management Systems (DBMSs) are a vital component for the modern companies operation, and as such, they must provide great data availability and high performance in the data access. With this purpose, practically all DBMS supply tuning knobs that can be adjusted in accordance with the workload characteristics. However, this task becomes each time more difficult, due to increasing complexity of the databases (DBs). The autonomic computing systems have detached as a promising approach to deal with the tuning complexity. Such systems are intelligent enough to manage its proper performance, having as main characteristic the capacity to perceive the environment, particularly its workload, and automatically reconfigure its resources and parameters in a proper manner. This characteristic is called of self-tuning, or database self-tuning. Typically, databases are used in environments where many users work concurrently. The concurrent data access can generate inconsistencies in the database. For this reason, the concurrent data access need be controlled. The leaders approaches to control concurrency are based on locks. However, the great number of lock conflicts can lead to a data access containment, and consequently to undesirable delays and reworks that could be prevented. To minimize these problems the majority of the DBMSs provides a set of tuning knobs, related with the transaction management, that can be adjusted. In this work, we present the state of the art in transactions self-tuning.

Keywords: Self-Tuning, Transactions, Concurrency Control, Heuristics, Workload.

Resumo. Sistemas de bancos de dados (SGBDs) são um componente de vital importância para o funcionamento das empresas modernas, e como tal, devem prover alta performance no acesso a dados e grande disponibilidade destes dados. Com esta finalidade, praticamente todos os bancos de dados fornecem parâmetros que podem ser ajustados de acordo com as características da carga de trabalho das aplicações. Entretanto, esta tarefa se torna cada dia mais difícil, devido à crescente complexidade dos BDs. Os sistemas computacionais autônomos têm se destacado como uma abordagem promissora para lidar com a complexidade destes ajustes. Tais sistemas são suficientemente inteligentes para gerenciar sua própria performance, tendo como principal característica a capacidade de perceber o ambiente, particularmente suas cargas de trabalho (*workload*), e re-configurar (ajustar) automaticamente os seus recursos e parâmetros de forma adequada. Esta característica é denominada de auto-sintonia de bancos de dados. Tipicamente, bancos de dados são utilizados em ambientes onde muitos usuários trabalham de forma concorrente, o que pode gerar inconsistências. Por este motivo, o acesso concorrente aos dados é controlado, principalmente, através de mecanismos baseados no conceito de bloqueios (*locks*). Porém, o número excessivo de conflitos na obtenção de bloqueios pode levar a uma contenção no acesso aos dados, e consequentemente a demoras indesejáveis e re-trabalhos que poderiam ser evitados. Para minimizar estes problemas a maioria dos SGBDs oferece um conjunto de parâmetros, relacionados com o gerenciamento das transações, que podem ser dinamicamente ajustados. Neste trabalho, apresentamos o estado da arte em auto-sintonia de transações.

Palavras-chave: Ajustes Automáticos, Transações, Controle de Concorrência, Heurísticas, Carga de Trabalho.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introdução

Sistemas de bancos de dados (SGBDs) são um componente de vital importância para muitos sistemas de informação de missão crítica nas empresas modernas, e como tal, devem prover alta performance no acesso a dados e grande disponibilidade destes dados. Com esta finalidade, praticamente todos os bancos de dados, que armazenam grandes volumes de dados, são gerenciados por administradores (DBAs - *Database Administrators*), os quais devem ser especialistas na tarefa de sintonia (*tuning*) de bancos de dados: ajuste dos parâmetros do SGBD de acordo com as características da carga de trabalho das aplicações. A atividade do DBA envolve o planejamento da capacidade de hardware, o projeto físico do banco de dados, ajuste das configurações dos recursos gerenciados e gerenciamento das dependências inter-sistemas (como por exemplo, entre o *middleware* e o SGBD). Entretanto, esta tarefa se torna cada dia mais difícil, devido à crescente complexidade dos BDs, consumindo um tempo cada vez maior. Aplicações atualmente comuns como os sistemas ERP *(Enterprise Resource Planning*) tipicamente criam mais de 20.000 tabelas e suportam centenas de usuários simultâneos. Imagine o tempo necessário para se definir e criar índices para um número tão grande de tabelas, ou mesmo, a dificuldade para ajustar manualmente a granularidade dos bloqueios. Os sistemas computacionais autônomos (*autonomic computing systems*) têm se destacado como uma abordagem promissora para lidar com a complexidade dos ajustes de performance dos bancos de dados modernos. Sistemas computacionais autônomos são sistemas suficientemente inteligentes para gerenciar sua própria performance. Uma importante característica destes sistemas é a capacidade de perceber o ambiente, particularmente suas cargas de trabalho (*workload*), e re-configurar (ajustar) automaticamente os seus recursos e parâmetros de forma adequada. Esta característica é denominada de auto-sintonia, ou sintonia automática, de bancos de dados. O principal princípio utilizado para a implementação da auto-sintonia de bancos de dados é o conceito de ciclo de controle de realimentação (*online feedback control loop*) [1]. Nesta abordagem, o sistema continuamente observa (monitora) certas medidas (métricas) de performance, e quando estas ultrapassam um determinado valor limite o sistema dinamicamente ajusta um ou mais parâmetros de performance (*tuning knobs*). Desta forma, esta abordagem se divide em três fases: monitoramento, predição e reação. Observe que devido às suas características de inteligência, agência, autonomia e reatividade, os agentes de software constituem-se em uma alternativa extremamente eficiente para a implementação da auto-sintonia de bancos de dados [27].

Tipicamente, bancos de dados são utilizados em ambientes onde muitos usuários trabalham de forma concorrente, ou seja, concorrem pelo acesso aos dados. Entretanto, este acesso concorrente pode gerar inconsistências no banco de dados. Por este motivo, o acesso concorrente aos dados é controlado por um módulo do SGBD denominado "gerenciador de transações". Os principais mecanismos utilizados com esta finalidade baseiam-se no conceito de bloqueios (*locks*). Porém, o número excessivo de conflitos na obtenção de bloqueios pode levar à uma contenção no acesso aos dados, e consequentemente a demoras indesejáveis e re-trabalhos que poderiam ser evitados. Para evitar, ou minimizar, problemas de performance causados pela contenção no acesso aos dados a maioria dos SGBDs oferece um conjunto de parâmetros, relacionados com o gerenciamento das transações, que podem ser dinamicamente ajustados, como por exemplo, nível de multi-programação, nível de isolamento, número máximo de bloqueios permitidos, dentre outros.

Neste trabalho, apresentamos o estado da arte em auto-sintonia de transações. Nossa principal contribuição consiste na investigação, descrição e análise das principais abordagens e projetos de pesquisa existentes nesta área.

O restante deste trabalho está organizado da seguinte forma: a seção 2 apresenta os conceitos básicos referentes à sintonia de transações. A seção 3 discute as principais propostas para auto-sintonia de transações presentes na literatura. Na seção 4 discutimos os parâmetros dos principais SGBD's comerciais que podem ser utilizados para sintonia e auto-sintonia de transações. A seção 5 conclui este trabalho e aponta direções para trabalhos futuros.

2 Sintonia de Transações

As versões mais recentes dos principais produtos de bancos de dados existentes no mercado já conseguem automatizar praticamente todos os ajustes internos com relação ao gerenciamento de transações. Desta forma, o DBA supostamente não deve se preocupar com tópicos tais como alocação de memória para páginas oriundas dos arquivos, dimensionamento de áreas para armazenamento de transações, nem controle de mecanismos de proteção de dados para que duas transações não gravem o mesmo dado ao mesmo tempo. Entretanto, dois problemas ainda são bastante corriqueiros em ambientes onde muitos usuários trabalham de forma concorrente: espera e *deadlocks*. Assim sendo, mesmo nos sistemas mais atuais, podem acontecer demoras indesejáveis e retrabalhos que poderiam ser evitados.

Os dois problemas que motivam este trabalho, espera (*blocking*) e *deadlock*, costumam ser confundidos. O primeiro acontece quando um usuário deve aguardar enquanto outro "prende" recursos que ele necessita. O fato de um usuário esperar não chega a ser um problema, já que decorre da convivência entre vários usuários, ou mais tecnicamente, concorrência. Mas, caso estas esperas comecem a acontecer com muita freqüência e com grandes durações, os usuários passarão a reclamar. Já o segundo problema, *deadlock*, resulta em espera mútua sem perspectiva de solução. Neste caso, enquanto o sistema não detectar e resolver o impasse, os usuários ficarão esperando um pelo outro [30].

Estes dois problemas são resultantes da estratégia utilizada pelos SGBD's para garantir a consistência dos dados mediante o acesso concorrente e compartilhado realizado por um conjunto de usuários: o protocolo de bloqueio em duas fases (2PL).

O principal objetivo de um mecanismo de controle de concorrência, como o bloqueio em duas fases, consiste em garantir que uma transação deve obedecer a quatro propriedades fundamentais, reunidas sob o acrônimo ACID [30, 31]:

• Atomicidade:	os comandos que constituem uma transação devem ser executados com sucesso e sem exceção, isto é, caso um falhe, todos devem ser desfeitos.
• Consistência:	uma transação deve iniciar tendo o Banco em estado consistente e, ao final, deve aplicar uma transformação que também deixe o Banco em outro estado consistente. Em outras palavras, nenhuma regra de integridade deve ser violada (chaves primárias duplicadas, chaves estrangeiras sem primária relacionada etc.).
• Isolamento:	em principio, uma transação não deve ter acesso aos dados manipulados por outra.

• Duração:	se uma transação foi concluída com sucesso, isto é, terminou por COMMIT (explícito ou não), existe o compromisso de que as atualizações realizadas por ela valham. Mesmo que não tenha sido possível gravar os dados alterados em disco, o SGBDR deve assegurar que isto acontecerá antes que o Banco esteja em estado consistente.

A seguir, descreveremos em maiores detalhes como os SGBD's gerenciam as transações e que ajustes podem ser realizados.

2.1. Gerenciamento de Bloqueios (Locks)

2.1.1. O que é um Bloqueio?

Um bloqueio (*lock*) é um objeto usado internamente pelo SGBD para indicar que um usuário possui o direito de aceso sobre um determinado recurso. Este usuário é dito o proprietário do bloqueio sobre o recurso. Aos demais usuários não será permitido executar nenhuma operação que seja incompatível (possa gerar inconsistências) com o bloqueio já existente sobre o recurso. Os SGBD's utilizam o conceito de bloqueio para controlar a concorrência entre as aplicações de diversos usuários que estejam operando sobre o mesmo banco de dados, no mesmo intervalo de tempo. Esta estratégia possibilita que os diversos usuários possam ler e escrever sobre o mesmo banco de dados, com a garantia de que as leituras não irão recuperar valores inconsistentes e que uma escrita não irá sobrescrever as modificações de uma outra operação de escrita [28].

2.1.2. O Gerenciador de Bloqueios

O gerenciador de bloqueios (*lock manager*) tem objetivo garantir que as requisições de bloqueios efetuadas pelos usuários sejam atendidas de forma eficiente, além de proporcionar alto grau de concorrência entre as aplicações OLTP (*Online Transaction Processing*), especialmente quando dados são inseridos freqüentemente.

Em geral, o *Lock Manager* faz parte do gerenciador de transações (*Transaction Manager*), como mostrado na figura 1. Todo comando SQL, uma vez preparado para execução, precisa interagir com o *Lock Manager*, para informar o recurso que precisa ser bloqueado, caso os recursos necessários não possam ser bloqueados, o *Lock Manager* envia uma notificação de espera. Ao final de uma transação, ou comando de leitura em níveis de isolamento baixos, também interage-se com o *Lock Manager* avisando-o que bloqueios podem ser liberados.

O *Lock Manager* também gerencia uma estrutura de dados interna onde são armazenadas as informações sobre os recursos que estão bloqueados (identificador da transação, tipo de bloqueio, identificador do objeto bloqueado, etc) e uma lista de espera, a qual indica que transações estão esperado por quais recursos [28].

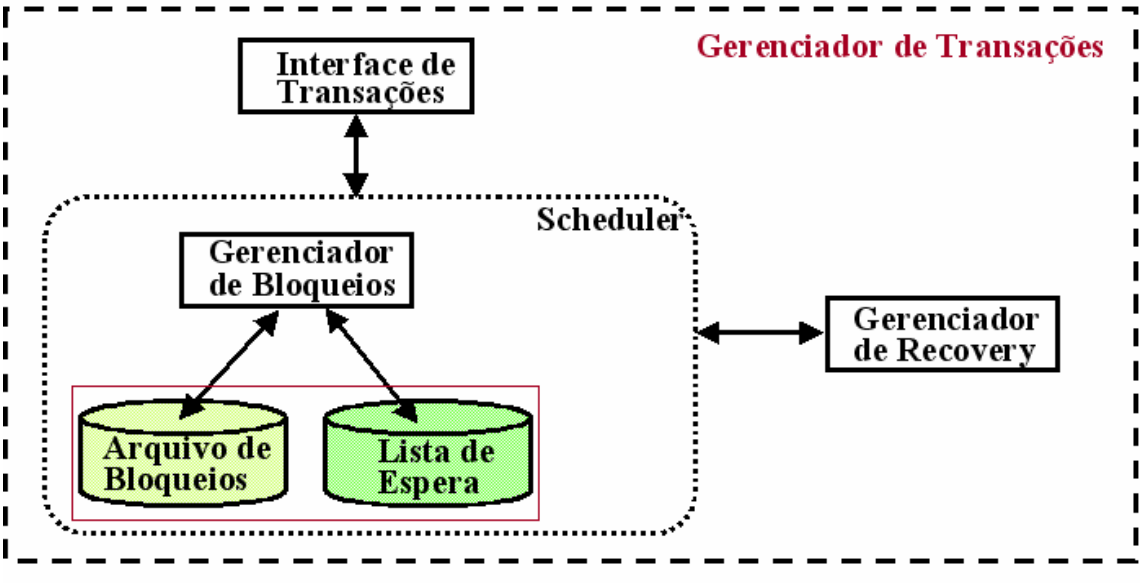

Figura 1. Arquitetura do Gerenciador de Transações do SQL Server 2000

No SQL Server 200, por exemplo, um *lock* representa uma estrutura de memória ocupando 64 bytes. Cada processo (identificado internamente por um SPID) possuindo um *lock*, também precisa outros 32 bytes para armazenar um *Lock Owner Block*. Vale ressaltar que um *lock* pode possuir vários processos associados, como por exemplo, quando há *locks* do tipo *Shared* [28].

2.1.3 Granularidade de Bloqueios

Em geral, os principais SGBD's utilizam múltiplas granularidades de bloqueio, ou seja, os bloqueios podem ser atribuídos em diferentes níveis, os quais variam entre granularidade fina e grossa. Um bloqueio de granularidade fina corresponde a um bloqueio sobre um recurso pequeno, como por exemplo, uma *tupla* ou uma página. Já um bloqueio de granularidade grossa atua sobre um recurso de maior tamanho, como uma tabela ou um banco de dados. O bloqueio com granularidade mais fina corresponde ao bloqueio a nível de *tuplas*, já o de granularidade mais grossa ao bloqueio a nível de banco de dados. A seguir, descrevemos os níveis de bloqueio utilizados pelos principais SGBD's [28]:

- **Tupla (*Row*):** Bloqueia uma única linha de uma tabela.
- **Chave (Key):** Bloqueio de linha aplicado sobre um índice.
- **Página (Page):** Bloqueia uma página de dados ou de índices (O *SQL Server* utiliza páginas de 8KB).
- **Extensão (Extent):** Bloqueia um conjunto de páginas (No *SQL Server*, bloqueia oito páginas de dados ou índices, ou ainda uma unidade de 64KB).

- **Tabela (Table):** Bloqueia uma tabela inteira, incluindo todos os seus dados e índices.

- **Banco de Dados (Database):** Bloqueia o banco de dados por completo.

Quanto mais fina for a granularidade do bloqueio maior será o grau de concorrência entre os usuários, porém, maior também será o *overhead* no gerenciamento dos bloqueios. Por outro lado, quanto mais grossa for a granularidade, menor será o grau de concorrência, mas menor também será o *overhead* despendido para o gerenciamento dos bloqueios. Por exemplo, ao utilizarmos bloqueios a nível de *tupla*, o SGBD terá que criar e gerenciar um bloqueio para cada linha da tabela acessada por um determinado usuário (ou transação), o que fatalmente irá causar um grande *overhead*. Porém, o grau de concorrência é elevado, uma vez que as *tuplas* que não estão bloqueadas podem ser utilizadas pelos demais usuários. Agora, se utilizarmos bloqueios a nível de tabela, teremos obviamente um menor grau de concorrência, já que uma tabela bloqueada por um determinado usuário não poderia ser acessada pelos demais usuários, mesmo que este esteja utilizando apenas uma linha da tabela. Contudo, o overhead gasto no gerenciamento de bloqueios é mínimo, uma vez que o SGBD teria que criar e gerenciar um único bloqueio para cada tabela [28, 29].

2.1.4. Modos de Bloqueios

O SGBD aloca bloqueios em um modo específico. O modo de um bloqueio determina como o recurso bloqueado pode ser acessado pelas transações concorrentes. Os principais SGBD's utilizam os seguintes modos [30, 31]:

- **Compartilhado (*Shared*)**: Usado por operações somente de leitura, como por exemplos cláusulas SELECT. Se uma transação obteve um *shared lock* sobre um determinado recurso, uma segunda transação também pode adquirir um *shared lock* sobre o mesmo recurso, mesmo que a primeira transação ainda não tenha concluído sua execução.

- **Exclusivo (*Exclusive*):** Usado por cláusulas que modificam os dados, tais como INSERT, UODATE e DELETE; garante que nenhum usuário além do proprietário do bloqueio poderá modificar ou consultar (ler) o recurso.

- **Atualização (*Update*):** Usado sobre recursos que serão atualizados futuramente pela transação (quando essa informação está disponível). Neste caso, o recurso é bloqueado inicialmente com um *update lock*. Porém, antes do recurso ser atualizado, o SGBD promove o *update lock* para um *exclusive lock*. Esta estratégia é utilizada para prevenir conflitos e *deadlocks*.

- **Intencional (*Intent*):** São bloqueios utilizados internamente para minimizar conflitos. *Intent locks* estabelecem uma hierarquia de bloqueios. Por e-

xemplo, se uma transação detém um *exclusive row lock* sobre um determinado registro, um *intent lock* sobre a tabela é atribuído à essa transação. Isso previne que uma outra transação obtenha um *exclusive lock* a nível de tabela (*table-level lock*).

- **Esquema (S*chema*):** Usado por operações que executam cláusulas DDL (*Data Definition Language*), tais como adição de uma coluna em uma tabela, remoção de uma tabela, etc. Assegura que uma tabela ou índice não será excluída, ou que o esquema não será modificado, enquanto estiverem sendo utilizados por outras sessões.

- **Atualização Volumosa (*Bulk Update*):** Usando durante a cópia de grandes volumes de dados para uma determinada tabela, por exemplo, durante a carga de dados, importação de dados, etc.

2.1.5. Níveis de Isolamento

Um nível de isolamento protege (isola) uma determinada transação das operações realizadas pelas demais transações. O nível de isolamento define o comportamento das cláusulas que compõem a transação, ou seja, define o escopo e a duração dos bloqueios. O escopo de um bloqueio pode ser de dois tipos: objeto e predicado. Um bloqueio com escopo a nível de objeto bloqueia um conjunto determinado de *tuplas*. Já um bloqueio com escopo a nível de predicado bloqueia as *tuplas* que satisfazem um determinado predicado. A duração de um bloqueio pode ser longa ou curta. Em um bloqueio de longa duração o objeto é retido até que a operação de *commit* ou *abort* seja executada. Já os bloqueios de curta duração são liberados imediatamente após a execução da operação associada ao bloqueio. A seguir, descrevemos os diversos níveis de isolamento existentes [28, 30].

• Read Uncommitted:	Bloqueios de leitura não são necessários e os bloqueios de escrita são de curta duração tanto pra objetos quanto para predicados. Permite leituras sujas (*dirty reads*), ou seja, pode-se ler dados cujas atualizações ainda não foram confirmadas ou canceladas. Trata-se do modo mais brando de controle de concorrência, porém cresce o risco de inconsistências, já que não garante-se a integridade dos dados lidos.
• Read Committed:	Bloqueios de leitura são de curta duração tanto pra objetos quanto para predicados. Bloqueios de escrita são de longa duração tanto para objetos quanto para predicados. Modo padrão, impede-se as leituras sujas.

• Repeated Read:	Os bloqueios de leitura são de longa duração para objetos e de curta duração para predicados. Já os bloqueios de escrita são de longa duração tanto para objetos quanto para predicados. Além de evitar leituras sujas, garante-se a repetição de uma leitura, isto é, uma vez lida uma linha, ela não poderá ser alterada por outra transação. Na prática, *locks* de leitura transformam-se em *locks* de atualização.
• Serializable:	Os bloqueios de leitura e de escrita são de longa duração tanto para objetos quanto para predicados. Aumenta ainda mais o isolamento da transação evitando que linhas sejam inseridas junto às que foram lidas. Por exemplo, suponha que tenham sido lidos os alunos mineiros. Enquanto durar a transação, nenhum mineiro poderá ser inserido por outra transação. As linhas recusadas recebem a denominação genérica de *fantasmas*.

2.1.6. Lock Hints

Os SGBD's oferecem um conjunto sugestões (*hints*) que podem ser utilizadas pelos usuários na sintaxe das cláusulas SQL. Essas sugestões indicam ao sistema como deve ser feito o bloqueio e o acesso às tabelas utilizadas pela consulta. É importante observar que os *locking hints* sobrescrevem o nível de isolamento corrente utilizado pelo sistema. Os *hints* modificam o comportamento do sistema permitindo que o DBA (ou usuário) explore a performance do SGBD [31].

Geralmente, os otimizadores de consulta realizam de forma automática a escolha das opções de bloqueio que são mais convenientes (adequadas). Assim, os *hints* devem ser utilizados somente quando for absolutamente necessário. A utilização incorreta ou inadequada de um *hint* pode afetar negativamente a performance do sistema. A seguir, descrevemos os principais *locking hints* disponibilizados pelo *SQL Server* 2000 [28].

- **HOLDLOCK:** Força o gerenciador de bloqueios a manter os bloqueios compartilhados até que a transação seja concluída. Difere do comportamento padrão, onde os bloqueios compartilhados são liberados tão logo não sejam necessários. Comportamento equivalente a *serializable*.

- **UPDLOCK:** Força a utilização de um *update lock* ao invés de um *shared lock* durante a leitura de uma tabela.

- **TABLOCK, TABLOCKX:** O *hint TABLOCK* força o gerenciador de bloqueios a utilizar um *shared table lock* ao invés de outra granularidade mais fina. O bloqueio será mantido até o final da execução da cláusula SQL. Se usado em conjunto com o *hint HOLDLOCK*, o bloqueio será mantido até a conclusão da transação. Já o *hint TABLOCKX* força o gerenciador a utilizar um *exclusive table lock*, impedindo que outra transação leia ou atualize a tabela até a conclusão da cláusula SQL ou da transação.

- **PAGLOCK:** Força o gerenciador a utilizar um *page lock*.

- **ROWLOCK:** Força o gerenciador a utilizar um row *lock*.

- **NOLOCK:** Utilizado em cláusulas SELECT, indica que o gerenciador deve executar a operação sem a necessidade de obter bloqueios.

- **READPAST:** Direciona o gerenciador a saltar as *tuplas* que por ventura se encontrarem bloqueadas. Neste caso, em vez da transação ficar esperando pela liberação do recurso bloqueado, ela simplesmente salta essa *tupla* (que deveria pertencer ao resulta da consulta).

- **READUNCOMMITTED:** Equivalente ao *NOLOCK*.

- **READCOMMITTED:** Força que o gerenciador utilize a mesma semântica do nível de isolamento *READCOMMIT*.

- **REPEATABLEREAD:** Força que o gerenciador utilize a mesma semântica do nível de isolamento *REPEATABLEREAD*.

- **SERIALIZABLE:** Força que o gerenciador utilize a mesma semântica do nível de isolamento *SERIALIZABLE*. Equivalente a *HOLDLOCK*.

2.1.7. Regras para Minimizar Bloqueios

O grande desafio para os DBA's com relação ao ajuste de transações é a relação entre performance e consistência. Freqüentemente, os profissionais de bancos de dados se deparam com a necessidade de optar por um destes requisitos (em detrimento do outro). Para isso, fazem escolhas quanto ao número de bloqueios que cada transação obtém, quanto ao tipo de bloqueio utilizado e quanto ao período de tempo que a transação mantém o bloqueio.

Do ponto de vista da performance, a transação ideal é aquela que mantém poucos bloqueios, utiliza bloqueios compartilhados e de curta duração. Entretanto, uma transação com estas características pode causar inconsistências no banco de dados. A seguir, apresentamos um conjunto de regras práticas para minimizar a ocorrência de conflitos e *deadlocks* [30, 31].

- Utilize transações curtas: Aumenta grau de concorrência e reduz tempo de UNDO.

- Transações somente de leitura podem utilizar níveis de isolamento mais baixo (Ex. OLAP)

- Elimine o uso de *locks* quando desnecessário: Por exemplo, quando a transação executa sozinha (ex. carga do BD) ou quando todas as transações são somente de leitura.

- Transações de longa duração devem bloquear a tabela, pois evita a formação de *deadlocks*.
- Evite *hot spots*: Por exemplo, execute comandos DDL fora do horário de pico. Lembrar que o *Log* é um hot spot.

- Acesse os *hot spots* tão tarde quanto possível.

- Evitar comandos de interação com o usuário: Estes comandos podem demorar indefinidamente e fazer com que as transações sejam de longa duração.

- Converter longos INSERT/SELECT em únicos INSERTs dentro de um laço. Desta forma, tranca-se uma linha por vez e não a tabela completa.

- Quando for necessário atualizar um grande número de registros deve-se utilizar a atualização em *loop* e diminuir a transação (ou seja, diminuir o número de cláusulas que compõem a transação). Por exemplo, agrupar atualizações (a cada 10) por transação. Neste caso, não há risco de desfazer muitas operações e o commit mais cedo libera as *tuplas* que não serão mais utilizadas.

- Evite transações aninhadas

- Gere valores auto-incrementáveis com a propriedade *identity*, pois é mais eficiente.

- Execute transações que alteram muitos dados fora dos horários de pico.

- Adquira bloqueios em uma mesma ordem (se possível). Evita a formação de *deadlocks*.

- Evite o *Lock Hint HOLDLOCK*, já que este emula níveis de isolamento altos (*REPEATABLE READ* ou *SERIALIZABLE*).

- Utilize opções de processamento de cursores que minimizem os bloqueios (*READ_ONLY*, para cursores de leitura e *OPTIMISTIC* para atualização).

- Ajuste o *deadlock* interval. Este parâmetro define a freqüência de execução do procedimento que verifica a existência de *deadlocks*. Uma freqüência elevada conduz o sistema a um *overhead*. Já uma freqüência baixa pode fazer com que *deadlocks* demorem muito para serem descobertos e solucionados.

- Ajuste o *lock table size*. Este parâmetro define o número de entradas na tabela de bloqueios. Se pequeno o sistema será forçado a aumentar a granularidade mesmo para transações de curta duração. Se o a quantidade de entradas na tabela de bloqueios passar de um limiar existem duas opções: Alterar a granularidade dos bloqueios ou aumentar o espaço reservado para a tabela de bloqueios.

- Otimizar o incremento da granularidade (*lock escalation*). Se temos como saber a priori a seletividade das consultas de uma transação sobre uma determinada tabela, podemos usar *hints* para otimizar o *lock escalation*.
- Criar índices em tabelas manipuladas via UPDATE ou DELETE com cláusula WHERE.

- Transações de longa duração devem usar, na maioria das vezes, *table locks*, a fim de evitar a formação de *deadlocks*, e as transações curtas devem usar *row locks*, para alcançar uma maior concorrência. O tamanho da transação aqui diz respeito ao tamanho da tabela manipulada: uma transação de longa duração é aquela que acessa praticamente todas as páginas da tabela.

- Use operações de escrita o mais próximo (possível) do final.

- Acesse as tabelas mais concorridas por último, especialmente para escrita.

3 Auto-Sintonia de Transações

Nesta seção, apresentamos uma classificação dos principais trabalhos encontrados na literatura que abordam questões relativas à auto-sintonia de transações em bancos de dados relacionais. Nossa preocupação, para a elaboração desta classificação, foi a de buscar trabalhos que consigam, de alguma forma, modelar o ambiente em que o módulo gerenciador de transações está inserido e ajustar este componente automaticamente a mudanças no ambiente. Neste sentido, uma proposta para um novo algoritmo para o controle de concorrência, por exemplo, não necessariamente seria considerado um trabalho sobre auto-sintonia de transações em nossa classificação; já um método que modele a carga de trabalho submetida ao sistema para ajustar automaticamente o seu nível de multiprogramação (MPL – *multiprogramming level)* seria incluído em nosso estudo.

Até o momento, muito progresso foi feito no sentido de tornar os SGBDs relacionais mais auto-sintonizáveis com relação ao processamento de transações. No entanto, ainda existem alguns desafios a serem vencidos. No estante deste documento, procuramos apresentar um breve resumo dos trabalhos que foram realizados e indicar alguns desafios que ainda precisam ser superados.

3.1. Classificação Automática da Carga de Trabalho

O tipo de carga de trabalho em um sistema de gerenciamento de bancos de dados (SGBD) é uma informação de fundamental importância na tarefa de sintonia (ajuste dos parâmetros) do sistema. A alocação dos recursos, tais como memória principal, nível de multiprogramação (MPL), dentre outros, pode variar bastante dependendo do tipo da carga de trabalho: OLTP (*Online Transaction Processing*) ou OLAP (*Online Analytical Processing*), também conhecido como DSS (*Decision Support Systems*). Além disso, o SGBD tipicamente experimenta mudanças no tipo da carga de trabalho durante o seu ciclo normal de processamento. Desta forma, os DBAs (*Database Administrators*) devem percebem mudanças significativas no tipo de carga de trabalho e em seguida proceder o ajuste dos parâmetros do sistema, buscando manter níveis aceitáveis de performance.

Basicamente, existem três modos de operação possíveis sobre o qual um SGBD pode operar (de acordo com o tipo de carga de trabalho). O primeiro modo de operação é denominado de modo "*Default*", neste modo o SGBD utiliza configurações apropriadas para uma carga de trabalho genérica (*mixed workloads*). O segundo modo é denominado "Carga de Trabalho Dominante", neste modo o SGBD é ajustado (sintonizado) para um determinado tipo de carga de trabalho (OLTP ou OLAP), que será dominante.

Em [17] os autores apresentam uma abordagem para identificar automaticamente o tipo de carga de trabalho que está sendo submetida ao SGBD: OLTP ou OLAP. A identificação automática do tipo de carga de trabalho é o primeiro passo para a construção de SGBDs autônomos, auto-ajustáveis, que consigam gerenciar sua própria performance de forma automática. O passo seguinte seria ajustar os parâmetros do sistema de acordo com a carga de trabalho e de forma também automática. Os autores construíram um modelo de classificação baseado nas características mais relevantes destes dois tipos de carga de trabalho. Este modelo é construído utilizando-se técnicas de classificação utilizadas em mineração de dados (*data mining*), especificamente árvores de decisão. O modelo proposto é utilizado para identificar mudanças no tipo da carga de trabalho que está sendo submetida ao SGBD. Outra contribuição deste trabalho foi a implementação de um classificador de carga de trabalho (sistema que monitora continuamente a carga de trabalho submetida ao sistema e indica se esta carga é OLTP ou

OLAP). Este classificador (*workload classifier*), através da análise da performance (desempenho) de um conjunto de recursos, cujos valores são coletados do SGBD em um pequeno intervalo de tempo, determina uma métrica denominada *DSSness index*, a qual representa quanto (em percentagem) o sistema avaliado é um sistema DSS. Ou seja, um *DSSness* de 80% significa que 80% da carga de trabalho pode ser classificada como DSS e 20% como OLTP. Experimentos demonstraram que este classificador identifica de forma correta o tipo de carga de trabalho submetida ao SGBD.

Entretanto, como os SGBD's podem estar sujeitos a mudanças no tipo de *workload* durante seu ciclo normal de processamento, não é suficiente para os SGBD's autônomos identificar o tipo de *workload* corrente (atualmente submetido ao SGBD), mas principalmente predizer quando uma mudança na carga de trabalho irá ocorrer. Por exemplo, um SGBD de um banco pode ser submetido a uma carga de trabalho OLTP durante a maior parte do mês, devido às transações de débito e crédito (que são transações de curta duração). Porém, nos últimos dias do mês, a carga de trabalho apresenta um comportamento (padrão) mais semelhante a DSS (*DSS-like*), devido à necessidade de relatórios financeiros, consultas gerenciais (as quais envolvem resumos e valores agregados). Essas mudanças podem ser previstas através da análise de dados históricos.

Com a finalidade de prever mudanças no comportamento da carga de trabalho submetida ao sistema poderíamos simplesmente deixar o módulo (componente) que classifica a carga de trabalho (*workload classifier*) ativo e monitorar o sistema constantemente. Porém, esta abordagem impõe um *overhead* desnecessário. Experimentos mostrados em [10] demonstraram que esta estratégia reduz o *throughput* do SGBD em torno de 10%. Neste mesmo trabalho os autores propõem o PSP (*Psychic-Skeptic Prediction Framework*), o qual permite ao SGBD aprender sobre o comportamento dinâmico da carga de trabalho ao longo do tempo e prever, de maneira pró-ativa, quando ocorrerá uma nova mudança significativa no tipo da carga de trabalho, através de um método de predição misto *on-line* (requer o monitoramento contínuo do sistema enquanto este estiver *on-line* e operacional) e *off-line* (não requer monitoramento contínuo, apresenta melhor desempenho, mas é menos seguro, pois não irá prever mudanças de comportamento excepcionais, que ocorrem ao longo de um dia, por exemplo. Predições erradas, podem levar a ajustes incoerentes e conseqüentemente à perda de performance), possibilitando que o SGBD de maneira autônoma re-configure os seus parâmetros de performance a fim de se adaptar ao novo comportamento (padrão) da carga de trabalho. O PSP tira vantagem das abordagens de predição *on-line* e *off-line*, possibilitando uma previsão efetiva e de baixo custo. Assim, o foco principal do PSP consiste nos padrões de comportamento repetitíveis e diários. Ou seja, não é adequado em situações com mudanças bruscas, que ocorrem subitamente durante um determinado dia por alguma razão inesperada, por exemplo. Os resultados experimentais mostram que o PSP apresentou ganhos de performance comparado aos demais modos de operação.

3.2. Controle de Carga

O controle de carga é necessário para prevenir que o sistema de banco de dados entre em "*thrashing*" (excessiva transferência de páginas/segmentos entre a memória principal e a memória secundária) devido a contenção no acesso aos dados, o que é causado pelo número excessivo de conflitos na obtenção de bloqueios. O método de controle de carga adotado por praticamente todos os sistemas de bancos de dados comerciais consistem em limitar o grau de multiprogramação, isto é, o número máximo de transações

que podem executar concorrentemente. Este método possui a limitação de não poder reagir à mudança da carga de trabalho.

Para superar esta limitação, o trabalho apresentado em [21] propõe um método adaptativo de controle de carga, o qual adapta de forma dinâmica e automática o nível de multiprogramação de acordo com a execução (evolução) da carga de trabalho. Neste artigo, os autores apresentam e comparam através de simulações dois algoritmos para ajustar de forma adaptativa o nível de multiprogramação. Os dois métodos propostos são: o método de passos incrementais (*Incremental Steps* - IS) e o método da aproximação parabólica (*Parabola Aproximation* - PA).

Já em [1], os autores apresentam e avaliam a performance de um outro método adaptativo, isto é, auto-ajustável, para o controle de carga. O princípio básico deste método consiste em monitorar a contenção no acesso aos dados através de uma métrica de performance denominada razão de conflito (*conflict ratio*), e reagir a mudanças críticas no valor desta métrica. Esta reação pode ser implementada através da suspensão temporária da admissão de novas transações ou através do cancelamento de transações bloqueadas que estão bloqueando outras transações. Com a finalidade de demonstrar a viabilidade prática do método proposto, foram executadas avaliações de performance. Os resultados obtidos mostram que o método proposto funciona bem com flutuações dinâmicas na carga de trabalho. A principal diferença entre essa abordagem e a proposta apresentada em [2] está na métrica de performance utilizada para verificar se é necessário ajustar o nível de multiprogramação e a política para admissão de novas transações. Na primeira a métrica de performance é razão de conflito, razão entre o número de bloqueios mantidos por todas as transações e o número de bloqueios mantidos por transações ativas. A condição necessária para que seja efetuado um ajuste no nível de multiprogramação é que a razão de conflito ultrapasse o valor limite de 1.3. Neste caso, a reação consiste em suspender temporariamente a admissão de novas transações. Quando o valor da razão de conflito for menor que 1.3 admite-se uma ou mais transações. Já na abordagem apresentada em [2] a métrica de performance utilizada é a vazão (*throughput*), em um determinado intervalo de tempo. Neste caso, a condição necessária para que seja efetuado um ajuste no nível de multiprogramação (DMP - *Degree of multiprogramming*) é que a vazão tenha decrescido no último intervalo de tempo. Neste caso, o ajuste consistem em diminuir o nível de multiprogramação. Caso a vazão tenha crescido no último intervalo de tempo, a ajuste consistem em aumentar o nível de multiprogramação. Os resultados obtidos pelos testes e simulações indicam que o método proposto apresenta uma melhor performance que as demais abordagens existentes.

Em [19], é apresentada uma análise sobre diversos problemas de sintonia, classificando-os em: problemas já solucionados e problemas ainda não resolvidos. Nesta classificação, é particularmente importante destacar que o autor afirma que o problema do controle de carga (limitaçao do MPL, controle de admissão e escalonamento), apesar de já solucionado para as tradicionais cargas de trabalho OLTP, ainda não possui solução para cargas de trabalho mistas (que envolvem características OLTP e OLAP).

3.3. Priorização Dinâmica de Transações

Cargas de trabalho transacionais e OLTP são cada vez mais comuns em sistemas computacionais, estando presentes em uma grande variedade de aplicações, desde de sistemas de comércio eletrônico até sistemas de apoio à força de vendas. Entretanto, em muitos casos, estas cargas de trabalho impõem aos usuários longas esperas e tempos de respostas impraticáveis, gerando insatisfação. Porém minimizar a espera é, em

muitas aplicações, muito mais importante para alguns usuários do que para outros. Por exemplo, comerciantes que realizam centenas de negócios por dia podem estar dispostos a pagar mais para reduzir a espera durante a execução de um negócio. Por outro lado, comerciantes que realizam uma única transação por mês podem aceitar uma variação maior no tempo de espera. Desta forma, a priorização de transações é importante em vários contextos, principalmente para reduzir o tempo de resposta dos clientes mais importantes ("*big spenders*"[1]), os quais frustrados por uma demora excessiva podem decidir realizar seus negócios em outro lugar.

A priorização de transações é dificilmente alcançada nos SGBDs tradicionais, devido ao protocolo de bloqueio em duas fases, comumente utilizado nestes sistemas, uma vez que uma determinada transação pode esperar por um recurso bloqueado por uma outra transação, independentemente de sua prioridade. As soluções tradicionais para escalonamento de bloqueios (*lock scheduling*), incluindo as soluções preemptivas e não preemptivas, possuem problemas de performance para cargas de trabalho do tipo TPC-C (*Transaction Processing Performance Council Benchmark C*) [5].

Em [5], os autores classificaram as transações em dois tipos: transações de alta e baixa prioridade. Além disso, estabeleceram como metas priorizar as transações de alta prioridade para que estas executem como se estivessem isoladas das transações de baixa prioridade (para isso fazem com que as transações de baixa prioridade não possam esperar por bloqueios de posse de transações de alta prioridade) e garantir que as transações de baixa prioridade não sejam excessivamente penalizadas. Este trabalho apresenta duas contribuições principais:

i) Uma análise estatística detalhada do mecanismo de bloqueio, em cargas de trabalho TPC-C, considerando-se diversas políticas de priorização de bloqueios (premptivas e não preemptivas). Esta análise descreve por quê as políticas não preemptivas falham na tentativa de ajudar as transações de alta prioridade e por quê as políticas preemptivas prejudicam excessivamente as transações de baixa prioridade.

ii) Uma política para priorização de bloqueios, denominada POW, é proposta e implementada. Esta política apresenta todos os benefícios da priorização preemptiva, sem suas desvantagens.

Já em [8], os autores propõem e analisam diversos mecanismos de priorização para cargas de trabalho transacionais em bancos de dados relacionais. As principais contribuições deste artigo são:

i) Uma análise detalhada dos recursos usados pelos *workloads* transacionais (TPC-C e TPC-W) em diversos bancos de dados, comerciais e não comerciais (IBM DB2, PostgreSQL e Shore), sobre diferentes configurações, identificando que recursos constituem potenciais "gargalos". Os resultados experimentais mostraram que o principal "gargalo" para cargas de trabalho TPC-C, em bancos de dados que utilizam 2PL (Shore e DB2) é a espera por bloqueios. Já para cargas de trabalho TPC-C, em bancos de dados que utilizam MVCC (*Multi Version Concurrency Control*) o principal "gargalo" é a sincronização nas operações de I/O. Para as cargas de trabalho TPC-W, a CPU foi o principal "gargalo" em todos os bancos de dados utilizados.

ii) A implementação e avaliação de performance de diversas políticas para priorização de transações (preemptivas e não preemptivas) nos bancos de dados PostgreSQL e Shore. Os resultados experimentais mostram que a utilização de políticas simples de priorização poder prover ganhos de performance (de duas a cinco vezes) para as transações de alta prioridade, sem prejudicar excessivamente a performance das transações de baixa prioridade.

3.4. Diagnóstico Automático de Problemas de Performance

Os principais bancos de dados utilizados atualmente caminham na direção da computação autônoma (*autonomic computing*) [9], neste sentido já proporcionam um conjunto de parâmetros que podem ser ajustados dinamicamente. Logicamente, o próximo passo nesta caminhada consiste no diagnóstico automático dos problemas de performance, a seleção dos parâmetros que devem ser dinamicamente ajustados e o ajuste automático destes parâmetros. No trabalho apresentado em [9], os autores introduzem um método para diagnosticar problemas de performance de forma automática e em seguida descrevem como este método pode ser incorporado nos atuais SGBDs utilizando-se o conceito de reflexão. Os autores demonstram a viabilidade da abordagem proposta através de uma implementação (prova de conceito) para o IBM DB2. No método de diagnóstico automático proposto, os usuários definem modelos para representar os recursos do sistema e a carga de trabalho. Além disso os usuários definem um conjunto de regras de diagnóstico. A partir destas informações é gerada uma árvore de diagnóstico, a qual pode ser usada para identificar potenciais origens de problemas de performance. Este artigo também descreve uma metodologia de como implementar a abordagem proposta nos SGBDs atuais, que oferecem parâmetros que podem ser dinamicamente ajustados. Esta metodologia baseia-se no conceito de reflexão computacional e utiliza características comuns nos principais SGBDs, como *triggers* e funções definidas pelo usuário. A viabilidade da abordagem proposta é comprovada através da implementação, para o IBM DB2, de uma ferramenta de diagnóstico automático, a qual analisa as informações disponíveis, sugere e executa uma série de ajustes a fim de incrementar a performance (*throughput*) do sistema.

3.5. Ajuste Automático do Nível de Isolamento

Os trabalhos apresentados em [22,23,24,25] discutem o resultado da investigação realizada sobre o projeto "*Isolation Testing Project*" [26] desenvolvido pelo departamento de Ciência da Computação da Universidade de Massachusetts (at Boston). O objetivo deste projeto consiste em tentar aumentar a performance das aplicações de bancos de dados que utilizam transações através da utilização do menor nível de isolamento que produz execuções corretas.

Em [22], os autores apresentam novas especificações (definições) para os níveis de isolamento (SQL-ANSI). A especificação proposta é mais precisa e portável (independente de implementação), no sentido em que pode ser aplicada não somente às abordagens baseadas em bloqueio (pessimistas), mas também à uma grande variedade de técnicas de controle de concorrência, incluindo, além dos protocolos baseados em bloqueio, os mecanismos de controle de concorrência otimistas e os protocolos baseados em múltiplas versões. Já em [23], os autores estendem o trabalho anterior apresentando dois níveis de isolamento adicionais, os quais são mais fortes que o "nível de isolamento 2" do padrão SQL-ANSI.

No trabalho apresentado em [25], os autores apresentam um novo algoritmo de controle de concorrência denominado "*generalized snapshot isolation* (GSI)", o qual é uma extensão do *snapshot isolation convencional* (CSI). A principal característica do protocolo GSI é que ele não atrasa as transações, as quais podem observar um snapshot antigo (desatualizado).

Porém, este projeto ("*Isolation Testing Project*") não conseguiu definir um mecanismo que conseguisse selecionar dinamicamente o menor nível de isolamento que pro-

duz execuções corretas.

Em [19], é apresentada uma análise sobre diversos problemas de sintonia, classificando-os em: problemas já solucionados e problemas ainda não resolvidos. Nesta classificação, é particularmente importante destacar que o autor classifica a sintonia automática no nível de isolamento das transações como um problema em aberto. Entretanto, a principal dificuldade para este ajuste automático consiste no fato de que a escolha do nível de isolamento está intimamente ligada à semântica das aplicações. Por exemplo, não seria razoável mudar automaticamente o nível de isolamento para "*Read Uncommitted*" em aplicações bancárias, mesmo que o ganho de performance fosse bastante elevado, ou mesmo que esta mudança solucionasse problemas extremos de desempenho.

3.6. Ajuste Automático da Granularidade de Bloqueios

Os principais SGBD's escolhem dinamicamente a granularidade e o modo dos bloqueios a serem atribuídos à uma transação (ou usuário), o DBA (*Database Administrator*) não necessita configurar nada para que este processo ocorra de forma eficiente. Esta escolha, do tipo de bloqueio com melhor relação custo-benefício, ocorre quando uma consulta é submetida, e baseia-se nas características do esquema do banco de dados e da própria consulta. Esta estratégia proporciona grande facilidade ao DBA e ganhos de performance, uma vez que o *overhead* é reduzido. Além disso, os SGBD's gerenciam dinamicamente o incremento automático da granularidade do bloqueio (*lock escalation*), ou seja, a conversão de muitos bloqueios de granularidade fina em alguns poucos de granularidade mais grossa. Esta conversão reduz o *overhead* existente quando são mantidos um grande número de bloqueios com granularidade fina (grande número de entradas na tabela de bloqueios). Por exemplo, o *SQL Server* inicialmente utiliza bloqueios a nível de *tupla*, contudo, converte vários bloqueios a nível de *tupla* em um único bloqueio a nível de tabela quando o número de bloqueios de *tupla* ultrapassa um determinado valor limite. Este limite é determinado dinamicamente pelo *SQL Server* e não necessita ser configurado.

4 Oportunidades para Sintonia e Auto-Sintonia de Transações nas Principais Ferramentas Comerciais

Esta seção apresenta o resultado da investigação realizada sobre os parâmetros de performance (*tuning*) existentes nos principais sistemas de gerenciamento de bancos de dados comerciais, como por exemplo, SQL Server [28, 29, 32, 33] e Oracle [2, 7].

4.1. As Opções de Configuração no SQL Server

4.1.1. *Locks:*

O SQL Server possibilita que os administradores configurem manualmente o número máximo de bloqueios permitidos, utilizando para isso o parâmetro *locks*. Entretanto, na maioria das vezes, isto não é recomendado. O parâmetro *locks* determina o número máximo de bloqueios que o *SQL Server* pode alocar. O valor padrão (*default*) para este parâmetro é 0, o que indica que o *SQL Server* irá dinamicamente e eficiente-

mente alocar e desalocar bloqueios de acordo com as necessidades do sistema. Inicialmente, dois por cento da memória alocada ao *SQL Server* é reservada para alocar as estruturas utilizadas para o gerenciamento dos bloqueios. Cada uma dessas estruturas consome 96 bytes de memória. Quando o espaço reservado inicialmente para as estruturas de bloqueio for utilizado e se mais bloqueios forem necessários, o *SQL Server* irá expandir este espaço, caso exista memória disponível. Entretanto, o espaço reservado para as estruturas de bloqueio não podem ultrapassar quarenta por cento do espaço total destinado ao *SQL Server*.

Caso seja atribuído um valor diferente de zero ao parâmetro locks, estaremos definindo um limite para o número de bloqueios que podem ser alocados. Se este limite é excedido o SGBD envia uma mensagem "*out of locks*". Neste caso, o DBA deve incrementar o valor do parâmetro *locks* [28, 29].

4.1.2. *Deadlock Interval:*

Este parâmetro define a freqüência de execução do procedimento que verifica a existência de *deadlocks*. Uma freqüência elevada conduz o sistema a um *overhead*. Já uma freqüência baixa pode fazer com que *deadlocks* demorem muito para serem descobertos e solucionados.

4.1.3. Lock Table Size:

Este parâmetro define o número de entradas na tabela de bloqueios. Se pequeno o sistema será forçado a aumentar a granularidade mesmo para transações de curta duração. Se o a quantidade de entradas na tabela de bloqueios passar de um limiar existem duas opções: Alterar a granularidade dos bloqueios ou aumentar o espaço reservado para a tabela de bloqueios.

4.1.4. Lock Escalation:

Se temos como saber a priori a seletividade das consultas de uma transação sobre uma determinada tabela, podemos usar *hints* para otimizar o *lock escalation,* ou seja, para selecionar a granularidade do bloqueio (*tupla*, tabela, etc).

4.1.5. Locking Timeout:

Define o período máximo de tempo (em milisegundos) que o *SQL Server* permitirá que uma transação espere pela liberação de um recurso bloqueado.

Ex:

```
SET LOCK_TIMEOUT  180000
```

PS: O valor –1 indica a não utilização de *timeout*.

O valor corrente do *lock timeout* pode ser consultado da seguinte forma:

```
SELECT @@lock_timeout
```

4.1.6. Procedure SP_LOCK:

Retorna informações sobre os bloqueios mantidos pelo *SQL Server*.

Ex:
EXECUTE sp_lock

4.1.7. Deadlock Priority:

Seta a prioridade da transação corrente ser escolhida como vítima, caso se envolva em um *deadlock*.

Ex:
SET DEADLOCK_PRIORITY { LOW | NORMAL | @deadlock_var }

4.2. As Opções de Configuração no Oracle

4.2.1 Enqueue Resources:

O parâmetro denominado ENQUEUE_RESOURCES determina o número de recursos que podem ser alocados (e gerenciados) simultaneamente pelo gerenciador de bloqueios, ou seja, determina o número de recursos que podem estar bloqueados simultaneamente.

4.2.2 Enqueue Locks:

O parâmetro ENQUEUE_LOCK determina a quantidade máxima de transações que podem esperar simultaneamente por um recurso bloqueado.

4.2.3. DML_LOCKS:

Um bloqueio DML (*DML lock*) é obtido sobre uma tabela que está sendo utilizada por uma cláusula DML (inserção, remoção, atualização). Um bloqueio DML é concedido para cada tabela modificada em uma transação. O parâmetro DML_LOCKS determina o número máximo de bloqueios DML que o gerenciador de bloqueios pode conceder simultaneamente. Este valor deve ser maior ou igual ao total de bloqueios sobre todas as tabelas que estão sendo acessadas concorrentemente por todos os usuários. O valor *default* para este parâmetro assume uma media de quarto tabelas referenciadas para cada transação. Em algumas aplicações, este valor pode não ser suficiente.

5 Conclusões e Trabalhos Futuros

As versões mais recentes dos principais produtos de bancos de dados existentes no mercado já conseguem automatizar praticamente todos os ajustes internos com relação ao gerenciamento de transações. Entretanto, dois problemas ainda são bastante corriqueiros em ambientes onde muitos usuários trabalham de forma concorrente: espera e *deadlocks*. Assim sendo, mesmo nos sistemas mais atuais, podem acontecer demoras indesejáveis e re-trabalhos que poderiam ser evitados.

Estes dois problemas são resultantes da estratégia utilizada pelos SGBD's para garantir a consistência dos dados mediante o acesso concorrente e compartilhado realizado por um conjunto de usuários: o protocolo de bloqueio em duas fases (2PL).

Assim, a contenção na utilização de bloqueios pode comprometer a performance do sistema. Contudo, o ajuste dos parâmetros relacionados com a gerência de bloqueios não é uma tarefa trivial, tanto pela necessidade de compreensão dos algoritmos que são usados pelo sistema, quanto pela possibilidade de inter-relações entre os ajustes. Atualmente, estes ajustes são realizados por DBA's extremamente especializados. Esta dependência de recursos humanos dificulta a implantação de SGBD's em larga escala. Desta forma, a construção de sistemas que sejam capazes de auto-sintonia de transações, e que alcancem, de forma automática, níveis de desempenho satisfatórios, torna-se de fundamental importância.

Referências Bibliográficas

[1] A. Mönkeberg and G. Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. pages 432–443, USA, 1992. Proceedings of the 18th International Conference on Very Large Data Bases (VLDB).

[2] Oracle White paper. Oracle 8i with Oracle Fail Safe 3.0, 2000.

[3] D. Patterson and A. Brow. Recovery-oriented computing. HPTS Workshop, 2001.

[4] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. pages 422–433. 21st International Conference on Data Engineering (ICDE'05), 2005.

[5] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Improving preemptive prioritization via statistical characterization of oltp locking. pages 446–457. 21st International Conference on Data Engineering (ICDE'05), 2005.

[6] S. Chaudhuri and G. Weikum. Foundations of automated database tuning. pages 964–965. SIGMOD Conference, 2005.

[7] Oracle White paper. Oracle Database 10g Release 2: The Self-Managing Database, 2005.

[8] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for oltp and transactional web applications. Page 535. 20th International Conference on Data Engineering (ICDE'04), 2004.

[9] P. Martin, W. Powley, and D. G. Benoit. Using reflection to introduce self-tuning technology into dbmss. pages 429–438. 8th International Database Engineering and Applications Symposium (IDEAS 2004), 2004.

[10] S. Elnaffar and P. Martin. An intelligent framework for predicting shifts in the workloads of autonomic database management systems. Luxembourg, November 2004. IEEE International Conference on Advances in Intelligent Systems - Theory and Applications (AISTA 04).

[11] D. J. Lilja K. Wu. Self-tuning speculation for maintaining the consistency of client-cached data. pages 91–100, Newport Beach, CA, USA, July 2004. IEEE Computer Society 2004.

[12] S. Chaudhuri, B. Dageville, and G. M. Lohman. Self-managing technology in database management systems. page 1243, Toronto, Canada, September 2004. Thirtieth International Conference on Very Large Data Bases (VLDB 2004).

[13] S. Chaudhuri, P. Ganesan, and V. R. Narasayya. Primitives for workload summarization and implications for sql. pages 730–741, Berlin, Germany, September 2003. 29th International Conference on Very Large Data Bases (VLDB 2003).

[14] S. Elnaffar, W. Powley, D. G. Benoit, and T. P. Martin. Today's dbmss: How autonomic are they? pages 651–655, Prague, Czech Republic, September 2003. Workshops of the 14th International Conference on Database and Expert Systems Applications (DEXA'03).

[15] E. Kwan, S. Lightstone, K. B. Schiefer, A. Storm, and L.Wu. Automatic database configuration for db2 universal database: Compressing years of performance expertise into seconds of execution. pages 620–629. 10th Conference on Database Systems for Business, Technology and Web, 2003.

[16] D. K. Burselon. Creating a Self-Tuning Oracle Database: Automating Oracle9i Dynamic SGA Performance. Rampant Techpress, March 2003.

[17] S. Elnaffar, P. Martin, and R. Horman. Automatically classifying database workloads. pages 622–624. Conference on Information and Knowledge Management (CIKM02), 2002.

[18] S. Elnaffar. A methodology for auto-recognizing dbms workloads. Conference of the Centre for Advanced Studies on Collaborative Research (CASCON02), 2002.
[19] G. Weikum, A. M¨onkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. pages 20–31, Hong Kong, China, August 2002. 28th International Conference on Very Large Data Bases (VLDB 2002).
[20] S. Lightstone, G. M. Lohman, and D. C. Zilio. Toward autonomic computing with db2 universal database. SIGMOD Record, 31:55–61, September 2002.
[21] H. Heiss and R.Wagner. Adaptive load control in transaction processing systems. pages 47–54, Barcelona, Catalonia, Spain, September 1991. 17th International Conference on Very Large Data Bases (VLDB 1991).
[22] A. Adya, B. Liskov, and P. O'Neil. Generalized Isolation Level Definitions. In Proceedings of the IEEE International Conference on Data Engineering, March 2000.
[23] Atul Adya, Barbara Liskov and Patrick O'Neil. Towards an Isolation Level Standard. SIGMOD.
[24] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha. Making Snapshot Isolation Serializable. ACM Transactions on Database Systems (TODS), 2005.
[25] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Generalized Snapshot Isolation and a Prefix-Consistent Implementation. Technical Reports in Computer and Communication Sciences (IC/2004/21), 2004. EPFL, Lausanne, Switzerland.
[26] http://www.cs.umb.edu/~isotest (Visitado em 05/07/2006)
[27] Milanés A. Y., Lifschitz S. e Salles M. A. V. Estado da Arte em Auto-sintonia de Sistemas de Bancos de Dados Relacionais. 2004.
[28] Delaney K. Inside Microsoft SQL Server 2000. 2001. Microsoft Press.
[29] Spenik M. Microsoft SQL Server 2000 DBA Survival Guide. 2001. SAMS.
[30] Shasha D. e Bonnet P. Database Tuning: Principles, Experiments, and Troubleshooting Techniques. 2002. Morgan Kaufmann.
[31] Dunham J. Database Performance Tuning Handbook. 1998. McGraw-Hill.
[32] Pelzer T. e Gulutzan P. Performance Tuning SQL. 2002. Addison Wesley.
[33] Whalen E., Garcia M., DeLuca S e Thompson D. Microsoft® SQL Server 2000™ Performance Tuning Technical Reference. 2001. Microsoft Press.