Fábio Renato de Almeida

Gerenciamento de transação e mecanismo de serialização baseado em Snapshot

São José do Rio Preto, SP – Brasil

Fevereiro de 2014

Fábio Renato de Almeida

Gerenciamento de transação e mecanismo de serialização baseado em Snapshot

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, área de concentração em Computação Aplicada, junto ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", câmpus de São José do Rio Preto.

Orientador: Prof. Dr. Carlos Roberto Valêncio

Programa de Pós-Graduação em Ciência da Computação
Departamento de Ciências de Computação e Estatística
Universidade Estadual Paulista "Júlio de Mesquita Filho"

São José do Rio Preto, SP – Brasil
Fevereiro de 2014

Almeida, Fábio Renato de.
Gerenciamento de transação e mecanismo de serialização baseado em Snapshot / Fábio Renato de Almeida. – São José do Rio Preto, 2014
126 p. : il., gráfs., tabs.

Orientador: Carlos Roberto Valêncio
Dissertação (mestrado) – Universidade Estadual Paulista "Júlio de Mesquita Filho", Instituto de Biociências, Letras e Ciências Exatas

1. Computação. 2. Sistema de transação (Sistemas de computação) 3. Engenharia de software. 4. Banco de dados – Gerência. 5. Armazenamento de dados. I. Valêncio, Carlos Roberto. II. Universidade Estadual Paulista "Júlio de Mesquita Filho". Instituto de Biociências, Letras e Ciências Exatas. III. Título.

CDU – 518.721

Ficha catalográfica elaborada pela Biblioteca do IBILCE
UNESP – Câmpus de São José do Rio Preto

Fábio Renato de Almeida

Gerenciamento de transação e mecanismo de serialização baseado em Snapshot

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, área de concentração em Computação Aplicada, junto ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", câmpus de São José do Rio Preto.

Prof. Dr. Carlos Roberto Valêncio
UNESP – São José do Rio Preto
(Orientador)

Prof.ª Dr.ª Elaine Parros Machado de Sousa
USP – São Carlos

Prof.ª Dr.ª Rogéria Cristiane Gratão de Souza
UNESP – São José do Rio Preto

São José do Rio Preto, SP – Brasil
Fevereiro de 2014

Pelos incontáveis momentos de alegria e carinho, dedico este trabalho às minhas duas meninas: Meg Maria de Almeida *e* Pandora de Almeida.
Foi uma dádiva estar com vocês.

Agradecimentos

Lembro-me de minha primeira entrevista para entrar no programa de Mestrado com o professor Valêncio como se fosse ontem: Ele me deu uma bronca! Claro, foi uma bronca gentil, mas foi uma bronca... ☺

Sempre tive curiosidade em saber como funcionam os gerenciadores de bancos de dados e, no momento em que conheci o Trabalho de Doutorado do professor Valêncio, percebi que era minha chance de matar essa curiosidade. Foi uma aventura e tanto, eu sofri, me desesperei, mas também aprendi muito e tive momentos incríveis de êxtase a cada trecho de código C++ que era compreendido.

Em meio a essa jornada conheci muitas pessoas inteligentes, tive professores fantásticos, cursei uma disciplina em Bauru e, mais uma vez graças ao professor Valêncio, tive a oportunidade de apresentar um trabalho na China.

Por todos esses motivos sou grato ao professor Valêncio, mas principalmente digo OBRIGADO por ter me aceitado como aluno: foi uma honra poder me aventurar nas linhas de código do `NuGeM`.

Agradeço também a todo o pessoal do Grupo de Banco de Dados, que por inúmeras vezes me auxiliaram.

Se cheguei até aqui foi porque meus pais nunca deixaram de me apoiar, e devo muito a eles. Agradeço ao meu irmão por acreditar na minha capacidade, algumas vezes até mais do que eu mesmo, e agradeço ao meu amigo Luciano Comar por me auxiliar no caminho árduo que todo ser humano deve trilhar durante seu primeiro contato com o banco de dados Oracle.

E por último, mas não menos importante, não poderia deixar de agradecer aos meus outros dois amigos, Marcus Vinícius Bongeovani e Marcus Vinícius Scarpelli, que por inúmeras vezes tiveram que ouvir sobre meus avanços e dificuldades pelo caminho e, pasmem... continuaram sendo meus amigos.

"Whether you think you can or
whether you think you can't,
you're right."

Henry Ford

Resumo

Dentre os diversos níveis de isolamento sob os quais uma transação pode executar, SNAPSHOT se destaca pelo fato de lidar com uma visão isolada da base de dados. Uma transação sob o isolamento SNAPSHOT nunca bloqueia e nunca é bloqueada quando solicita uma operação de leitura, permitindo portanto uma maior concorrência quando a mesma é comparada a uma execução sob um isolamento baseado em bloqueios. Entretanto, SNAPSHOT não é imune a todos os problemas decorrentes da concorrência e, portanto, não oferece garantia de serialização. Duas estratégias são comumente empregadas para se obter tal garantia. Na primeira delas o próprio SNAPSHOT é utilizado, mas uma alteração estratégica na aplicação e na base de dados, ou até mesmo a inclusão de um componente de software extra, são empregados como auxiliares para se obter apenas históricos serializáveis. Outra estratégia, explorada nos últimos anos, tem sido a construção de algoritmos fundamentados no protocolo de SNAPSHOT, mas adaptados de modo a impedir as anomalias decorrentes do mesmo e, portanto, garantir serialização.

A primeira estratégia traz como vantagem o fato de se aproveitar os benefícios de SNAPSHOT, principalmente no que diz respeito ao monitoramento apenas dos elementos que são escritos pela transação. Contudo, parte da responsabilidade em se lidar com problemas de concorrência é transferida do Sistema Gerenciador de Banco de Dados (SGBD) para a aplicação. Por sua vez, a segunda estratégia deixa apenas o SGBD como responsável pelo controle de concorrência, mas os algoritmos até então apresentados nesta categoria tem exigido também o monitoramento dos elementos lidos.

Neste trabalho é desenvolvida uma técnica onde os benefícios de SNAPSHOT são mantidos e a garantia de serialização é obtida sem a necessidade de adaptação do código da aplicação ou da introdução de uma camada de software extra. A técnica proposta é implementada em um protótipo de SGBD que possui características temporais e que foi construído para demonstrar a aplicabilidade da técnica em gerenciadores que adotam o modelo orientado a objetos. Contudo, a estratégia proposta é independente de modelo de dados e pode ser aplicada também a gerenciadores relacionais ou a sistemas de armazenamento na nuvem.

Palavras-chaves: Gerenciamento de transação. Serialização. Snapshot. Estrutura de armazenamento. Bancos de dados não convencionais.

Abstract

Among the various isolation levels under which a transaction can execute, SNAPSHOT stands out because of its capacity to work on an isolated view of the database. A transaction under the SNAPSHOT isolation never blocks and is never blocked when requesting a read operation, thus allowing a higher level of concurrency when it is compared to an execution under a lock-based isolation. However, SNAPSHOT is not immune to all the problems that arise from the competition, and therefore no serialization warranty exists. Two strategies are commonly employed to obtain such assurance. In the first one SNAPSHOT itself is used, but a strategic change in the application and database, or even the addition of an extra software component, are employed as assistants to get only serializable histories. Another strategy, explored in recent years, has been the coding of algorithms based on the SNAPSHOT protocol, but adapted to prevent the anomalies arising from it, and therefore ensure serialization.

The first strategy has the advantage of exploring the benefits of SNAPSHOT, especially with regard to monitoring only the elements that are written by the transaction. However, part of the responsibility for dealing with competition issues is transferred from the Database Management System (DBMS) to the application. In turn, the second strategy leaves only the DBMS as responsible for concurrency control, but the algorithms presented so far in this category also require the monitoring of the elements that the transaction reads.

In this work we developed a technique where the benefits of SNAPSHOT use are retained and serialization warranty is achieved without the need for adaptation of application code or the addition of an extra software layer. The proposed technique is implemented in a prototype of a DBMS that has temporal features and has been built to demonstrate the applicability of the technique in systems that employ the object-oriented model. However, the proposed strategy is independent of the adopted data model and can also be applied to relational databases or cloud storage systems.

Key-words: Transaction management. Serialization. Snapshot. Storage structure. Non-conventional databases.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

2PL Two-Phase Locking

ACID Atomicidade, Consistência, Isolamento e Durabilidade

ANSI American National Standards Institute

LRU Least Recently Used

MVCC MultiVersion Concurrency Control

NuGeM NÚcleo GErenciador de dados Multimídia

ODMG Object Data Management Group

OID Object IDentifier

PL Portable-Level

PL-SI Portable-Level Snapshot Isolation

RAID Redundant Array of Independent Disks

S2PL Strict Two-Phase Locking

SGBD Sistema Gerenciador de Banco de Dados

SQL Structured Query Language

SSD Solid State Drive

TPC Transaction processing Performance Council

UML Unified Modeling Language

Sumário

Introdução

Por volta dos anos 60 os computadores foram dotados da capacidade de armazenar e gerenciar grandes quantidades de dados com o advento de meios de armazenamento persistente capazes de manipulá-los de forma direta e eficiente. Os gerenciadores de bancos de dados surgiram nessa mesma época, decorrentes dessa capacidade e da necessidade de estruturar os dados e oferecer rotinas padronizadas de acesso. Um Sistema Gerenciador de Banco de Dados (SGBD) é um programa, ou conjunto de programas, cujo propósito é controlar todos os aspectos que cercam um banco de dados, tais como sua estrutura, a leitura e escrita dos dados, os mecanismos de recuperação diante de falhas de hardware ou software, o controle de concorrência, etc (NASSU; SETZER, 1999).

Voltando-se à infraestrutura responsável pelo controle de concorrência em um SGBD e considerando apenas transações que não ferem ou violam intencionalmente as regras impostas sobre os dados, uma única transação em atividade é capaz de levar a base de dados de um estado consistente a outro sempre. Entretanto, dependendo do nível de isolamento sob o qual uma transação se encontra, a execução intercalada das ações disparadas por duas ou mais transações pode violar algumas das restrições sobre os dados ou fazer com que uma transação concorrente tenha uma visão, ainda que momentânea, inconsistente da base de dados. Portanto, a garantia de consistência está relacionada não apenas ao bom comportamento da transação mas também ao nível de isolamento sob o qual a mesma executa.

Motivação

Para que a garantia de consistência seja obtida, pode-se utilizar um nível de isolamento que impeça toda e qualquer ação conflitante. Tradicionalmente o nível de isolamento Serializable adota essa estratégia e, portanto, apenas históricos serializáveis são permitidos. Dessa forma, o resultado da execução intercalada das ações de transações concorrentes é o mesmo que o de uma execução serial. Todavia, o grau de concorrência oferecido acaba sendo menor em razão de um número maior de ações que são impedidas. O isolamento Snapshot é capaz de suportar uma maior concorrência, mas seu protocolo não oferece garantia de um histórico de execução transacional serializável e, portanto, a base de dados pode ser levada a um estado inconsistente.

Pelo fato de lidar com múltiplas versões dos elementos na base de dados, o isolamento Snapshot é capaz de oferecer não apenas um bom nível de concorrência mas

também um grau de consistência próximo ao obtido pelo isolamento SERIALIZABLE. Por esse motivo, algumas propostas tem surgido na literatura de modo a se obter garantia de consistência dos dados com base no isolamento SNAPSHOT, as quais podem ser agrupadas em duas estratégias. Na primeira delas o próprio SNAPSHOT é utilizado, mas uma alteração estratégica na aplicação e na base de dados (JORWEKAR et al., 2007; FEKETE et al., 2005), ou até mesmo a inclusão de um componente de software extra (ALOMARI; FEKETE; ROHM, 2009; ALOMARI, 2008), são empregados como auxiliares para se obter apenas históricos serializáveis. Outra estratégia, explorada nos últimos anos, tem sido a construção de algoritmos fundamentados no protocolo de SNAPSHOT, mas adaptados de modo a impedir as anomalias decorrentes do mesmo e, portanto, garantir serialização (YABANDEH; FERRO, 2012; REVILAK; O'NEIL; O'NEIL, 2011; REVILAK, 2011; CAHILL, 2009; CAHILL; RöHM; FEKETE, 2009; CAHILL; RöHM; FEKETE, 2008).

A primeira estratégia traz como vantagem o fato de se aproveitar os benefícios de SNAPSHOT, principalmente no que diz respeito ao monitoramento apenas dos elementos que são escritos pela transação. Contudo, parte da responsabilidade em se lidar com problemas de concorrência é transferida do SGBD para a aplicação. Por sua vez, a segunda estratégia deixa apenas o SGBD como responsável pelo controle de concorrência, mas os algoritmos até então apresentados nesta categoria tem exigido também o monitoramento dos elementos lidos.

Objetivos

O objetivo deste trabalho foi conceber um mecanismo que ofereça garantia de consistência aproveitando o melhor das duas estratégias, ou seja, impondo sobrecarga de gerenciamento apenas sobre os elementos que são escritos pela transação e, ao mesmo tempo, deixando o SGBD como o único responsável pelo controle de concorrência. Dessa forma, foi desenvolvida uma técnica onde os benefícios de SNAPSHOT são mantidos e a garantia de serialização é obtida sem a necessidade de adaptação do código da aplicação ou da introdução de uma camada de software extra. O mecanismo adotado, denominado *daemon*, foi implementado em um SGBD de codinome Draco, o qual possui características temporais e foi construído especificamente para demonstrar a aplicabilidade da técnica em gerenciadores que adotam o modelo orientado a objetos. Contudo, a estratégia proposta é independente de modelo de dados e pode ser aplicada também a gerenciadores relacionais ou a sistemas de armazenamento na nuvem.

Resultados e contribuições

O mecanismo desenvolvido para o gerenciamento transacional mostrou ser possível obter garantia de consistência dos dados não apenas no gerenciador de dados complexos

construído mas também em gerenciadores relacionais como SQL Server, Firebird e Oracle. O novo nível de isolamento proposto, denominado neste trabalho de Daemon Snapshot, é capaz de garantir um histórico de execução transacional serializável no SQL Server com uma taxa de erro de serialização menor que a obtida pelo próprio isolamento Serializable, o qual é nativo do SQL Server. Para os gerenciadores Firebird e Oracle, que não oferecem um isolamento que garante consistência dos dados, Daemon Snapshot se mostra uma opção não apenas viável em termos de concorrência obtida como também a única existente capaz de impor sobrecarga de gerenciamento somente sobre os elementos escritos.

Como fruto deste trabalho foram publicados três artigos (VALêNCIO et al., 2013; ALMEIDA et al., 2012; VALêNCIO et al., 2011) e dois relatórios técnicos (VALêNCIO; ALMEIDA, 2013a; VALêNCIO; ALMEIDA, 2013b). Além disso, foi gerado código C++ para a nova versão do gerenciador de dados complexos `NuGeM` e para o gerenciador de codinome Draco, construído apenas para demonstrar a aplicabilidade da técnica de *daemons* sobre gerenciadores não convencionais. Outros dois artigos sobre o trabalho também foram escritos mas ainda encontram-se em processo de submissão e aceite. De modo a suportar os testes, foram construídos também dois sistemas de *benchmark*, um implementado em C++ e outro em C#.

Organização do trabalho

Os capítulos desta dissertação estão organizados da seguinte forma:

- Capítulo 1 — Fundamentação teórica. O conceito de transação e as especificações de isolamento que foram propostas no decorrer dos anos são descritos no Capítulo 1.

- Capítulo 2 — Snapshot. No Capítulo 2 o isolamento Snapshot é apresentado juntamente com as principais técnicas encontradas na literatura para se obter garantia de serialização por meio deste.

- Capítulo 3 — Daemons e Draco. Um mecanismo de serialização também fundamentado em Snapshot é o resultado deste trabalho. As características da técnica proposta são descritas no Capítulo 3 juntamente com um protótipo de SGBD que a implementa.

- Capítulo 4 — Testes e resultados. A infraestrutura proposta para o SGBD no Capítulo 3 é testada no Capítulo 4 de modo a levantar questões relacionadas a desempenho e garantia de serialização. Os resultados dos testes são comparados com o uso de Snapshot em sua forma original. O sistema de *benchmark* adotado

é então adaptado e aplicado aos gerenciadores convencionais SQL Server, Firebird e Oracle de modo a comparar o grau de consistência e concorrência obtidos pelos níveis de isolamento existentes diante da técnica desenvolvida neste trabalho.

1 Fundamentação teórica

A garantia de consistência dos dados pode ser facilmente obtida se um SGBD simplesmente executar as transações em série. Todavia tal abordagem é ineficiente pois não oferece a possibilidade de concorrência. Uma vez que o processamento intercalado de transações seja permitido, torna-se imprescindível a utilização de mecanismos que isolem de maneira adequada suas ações de modo que conflitos sejam evitados e o resultado final obtido seja o mesmo que o de uma execução serial. Não obstante o isolamento ideal seja aquele que ofereça total garantia de consistência, muitas vezes este impõe uma menor concorrência em virtude de um número maior de ações conflitantes que devem ser impedidas. Por esse motivo, níveis de isolamento não tão rígidos também são oferecidos pelos SGBDs para que as aplicações possam, em pontos estratégicos, abdicar parcial ou totalmente da garantia de consistência em prol de uma maior eficiência.

Neste capítulo o conceito de transação é apresentado juntamente com as especificações de isolamento que foram propostas no decorrer dos anos. São cobertas as definições originalmente introduzidas no trabalho "Graus de Consistência" (GRAY et al., 1976), o padrão de isolamento adotado pela norma ANSI (ANSI, 1992), sua revisão (BERENSON et al., 1995) e, finalmente, a redefinição (ADYA; LISKOV; O'NEIL, 2000; ADYA, 1999) das especificações ANSI que tornam os níveis de isolamento portáveis para qualquer tipo de tecnologia transacional: bloqueios, otimista, multiversão, etc. O capítulo termina com um breve resumo das principais características de cada uma das especificações.

1.1 Transação

Sob o ponto de vista de uma aplicação, uma transação consiste numa ferramenta que delega ao SGBD a responsabilidade de lidar com problemas de concorrência, execução parcial e falhas no sistema, permitindo que a aplicação trate a transação como uma unidade de execução isolada e sem falhas (LIU; ÖZSU, 2009). Sob o ponto de vista do próprio SGBD, uma transação é um grupo constituído por uma sequência de ações atômicas cujo propósito é levar a base de dados de um estado consistente a outro. Se uma das ações falhar, então a transação inteira é revertida de modo a recuperar o estado anterior consistente da base de dados. A transação é, portanto, uma unidade atômica maior (BERENSON et al., 1995; GRAY et al., 1976).

Desta forma, um sistema transacional confiável deve apresentar as seguintes propriedades (YABANDEH; FERRO, 2012; CAHILL; RöHM; FEKETE, 2009; LIU; ÖZSU, 2009;

GRAY; REUTER, 1993), conhecidas na literatura como propriedades ACID[1]:

- **Atomicidade** — Dentro do contexto de uma transação, as ações estão sujeitas à condição tudo-ou-nada, ou seja, não existe a possibilidade de efetivação parcial de suas ações. Assim sendo, a transação em si comporta-se como uma unidade atômica e indivisível.

- **Consistência** — As restrições impostas sobre os dados devem ser garantidas ao término da transação. Essa garantia está sujeita ao nível de isolamento sob o qual a transação executa e também requer que a própria aplicação coopere de modo a não solicitar ações que violem as regras de negócio pertinentes ao banco de dados em questão.

- **Isolamento** — Idealmente uma transação deve se comportar como se fosse a única em atividade, consumindo e gerando apenas dados consistentes. O nível de isolamento escolhido para uma transação em particular define o comportamento do sistema na presença de transações concorrentes e este impacta diretamente na consistência dos dados e no volume de transações concorrentes suportadas (*throughput*).

- **Durabilidade** — Uma vez que a transação tenha encerrado com sucesso (*commit*), o resultado de suas ações deve sobreviver a falhas de hardware ou software.

Considerando-se apenas transações bem comportadas, ou seja, que não ferem ou violam intencionalmente as regras impostas sobre os dados, uma única transação em atividade é capaz de levar a base de dados de um estado consistente a outro sempre. Entretanto, dependendo do nível de isolamento sob o qual uma transação se encontra, a execução intercalada das ações disparadas por duas ou mais transações pode violar algumas das restrições impostas sobre os dados ou fazer com que uma transação concorrente tenha uma visão, ainda que momentânea, inconsistente da base de dados. Neste caso diz-se que algumas das ações são conflitantes.

Na literatura (YABANDEH; FERRO, 2012; ADYA; LISKOV; O'NEIL, 2000; BERENSON et al., 1995) é comum a modelagem da execução intercalada de transações por meio de um histórico contendo a requisição linear de suas ações, tais como a leitura e escrita de elementos específicos na base de dados: um objeto, uma linha em uma tabela, uma página ou até mesmo uma tabela inteira. Um conflito ocorre quando duas ações em um histórico pertencem a transações concorrentes, agem sobre um mesmo elemento e pelo menos uma dessas ações é uma operação de escrita. Ações conflitantes também podem ocorrer sobre

[1] ACID — Atomicity, Consistency, Isolation, Durability; O termo foi originalmente concebido por Haërder e Reuter em 1983. Embora pouco comum, o acrônimo CAPS (Consistency, Atomicity, Persistency, Serializability) também pode ser utilizado (GRAY; REUTER, 1993).

um conjunto de elementos, os quais são cobertos por um predicado[2] (BERENSON et al., 1995).

Se o resultado obtido de uma execução intercalada for o mesmo que o de uma execução serial das transações envolvidas, diz-se que o histórico resultante da execução intercalada é serializável (YABANDEH; FERRO, 2012; BERENSON et al., 1995). Um histórico não serializável apresenta, portanto, ações conflitantes. Contudo, o fato de existirem ações conflitantes em um histórico não implica necessariamente que o mesmo não seja serializável. Assim sendo, um dos desafios enfrentados por pesquisadores na área de banco de dados é conceber mecanismos para se identificar ações conflitantes que levam a ocorrência de históricos não serializáveis.

1.2 Suporte transacional

Em sistemas onde o suporte transacional é oferecido com base em uma única versão dos elementos, cada transação tem acesso ao mesmo item de dado e, portanto, está sujeita a uma série de problemas de concorrência. A vantagem desse modelo é que uma única instância dos dados é mantida.

Em uma implementação baseada em multiversão, os dados são inseridos e atualizados por intermédio da criação de novos elementos, ou seja, sem que haja a sobrescrita de dados antigos. Neste caso o sistema deve registrar o momento em que cada transação inicia e encerra de modo que seja possível identificar qual versão de um determinado elemento é visível a uma transação em particular, tornando a mesma imune à maioria dos problemas relacionados à concorrência. Todavia, esse tipo de suporte transacional requer a existência de mais de uma instância dos dados (YABANDEH; FERRO, 2012).

Em ambos os casos a identificação de conflitos pode ser feita utilizando-se uma abordagem pessimista ou otimista. Na abordagem pessimista, bloqueios compartilhados e exclusivos são obtidos sobre os elementos à medida que estes são requisitados pelas transações. Se uma transação requisita um bloqueio exclusivo sobre um elemento que já se encontra bloqueado por outra transação, o sistema deve abortar uma das transações ou congelar a transação solicitante até que o bloqueio seja liberado. Na abordagem otimista, também conhecida como livre de bloqueios, nenhum bloqueio é imposto sobre os elementos durante o processamento das transações. Neste caso os conflitos devem ser identificados no momento em que a transação inicia o processo de encerramento (fase de *commit*) e, caso haja algum, a transação deve abortar (YABANDEH; FERRO, 2012). O processo de validação consiste basicamente em verificar se a transação pode ser serializada

[2] Um predicado é uma condição booleana que age sobre as entidades colecionadoras dos elementos na base de dados de modo que um subconjunto destes seja selecionado para leitura e/ou escrita (ADYA; LISKOV; O'NEIL, 2000).

comparando-se os elementos que leu e escreveu com aqueles que foram lidos e escritos por outras transações que já encerraram (ADYA, 1999; KUNG; ROBINSON, 1981).

De modo intuitivo, a abordagem pessimista parte do pressuposto de que algo pode dar errado e, portanto, bloqueia os elementos assim que os mesmos são acessados. Analogamente, a abordagem otimista assume que nenhum conflito irá ocorrer e, portanto, deixa a verificação desses apenas para a etapa de encerramento.

Muitos SGBDs adotam um suporte transacional baseado em bloqueios juntamente com o protocolo de "Bloqueio de Duas Fases" (Two-Phase Locking — 2PL) de modo a garantir históricos serializáveis. O protocolo 2PL impõe as transações uma fase de crescimento e uma de encolhimento. Durante a fase de crescimento bloqueios são adquiridos e as ações são executadas. A fase de encolhimento inicia-se quando um bloqueio é liberado e, a partir deste ponto, nenhum outro bloqueio pode ser adquirido, ou seja, a transação para de requisitar novos bloqueios tão logo inicia a liberação dos já existentes (BERENSON et al., 1995; BERNSTEIN; GOODMAN, 1981). Implementações baseadas em bloqueios podem oferecer o recurso de serialização a um alto custo de gerenciamento e baixa concorrência (YABANDEH; FERRO, 2012).

Um sistema transacional capaz de garantir serialização deve permitir que as transações executem sob um nível de isolamento que assegure que suas ações não entrem em conflito com as ações de transações concorrentes (YABANDEH; FERRO, 2012). Entretanto, níveis de isolamento não tão rígidos também são úteis quando a garantia de consistência pode, em alguns momentos, ser sacrificada em prol de uma maior concorrência (BERENSON et al., 1995).

1.3 Graus de consistência

A forma mais simples de se garantir a consistência é simplesmente evitando-se que ações conflitantes ocorram por meio da aquisição de bloqueios sobre os elementos na base de dados à medida que estes vão sendo requisitados pelas transações.

Um bloqueio compartilhado é obtido sobre um elemento quando uma transação deseja lê-lo. Neste caso, outras transações também podem ler o elemento, mas nenhuma pode alterá-lo. Por sua vez, um bloqueio exclusivo é obtido sobre um elemento quando uma transação deseja escrevê-lo, cabendo apenas a transação detentora do bloqueio o direito de escrever e também ler o elemento (GRAY et al., 1976).

O mecanismo de bloqueios ajuda a garantir a consistência, mas reduz a concorrência. Por esta razão, em geral diversos níveis de isolamento são oferecidos de modo que cada transação possa ser executada sob a garantia de consistência necessária e o desempenho desejado (ADYA; LISKOV; O'NEIL, 2000; GRAY et al., 1976).

Originalmente o conceito de níveis de isolamento foi introduzido por GRAY et al., 1976 sob a denominação de "Graus de Consistência", onde quatro graus de consistência foram definidos: 0 – 3. Conforme observa-se na Figura 1, o grau 1 apoia-se sobre o grau 0 de modo que a garantia de isolamento pertinente ao grau 0 também esteja presente no grau 1. De modo análogo, o grau 2 possui sua própria garantia mais as garantias dos graus 0 e 1. Por sua vez, transações que executam sob o grau de consistência 3 possuem todas as garantias de isolamento.

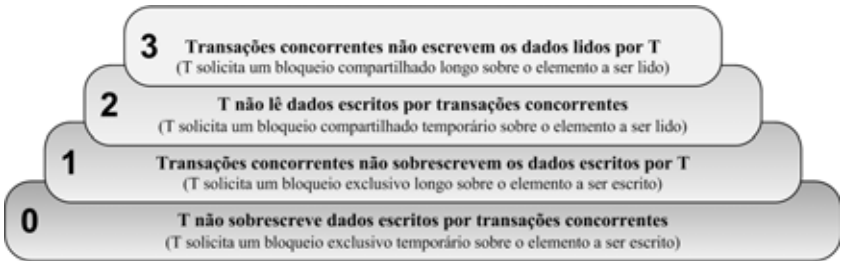

Figura 1 – Graus de consistência e suas respectivas garantias de isolamento. T representa uma transação em andamento (GRAY et al., 1976).

Transações sob o grau de consistência 0 são irreversíveis em virtude dos elementos escritos serem instalados imediatamente na base de dados antes mesmo do encerramento da transação. A partir do grau 1, os elementos escritos são instalados na base de dados apenas se a transação que os escreveu encerrar com uma operação de *commit*. Dessa forma, os SGBDs devem garantir que as transações executem pelo menos sob o grau de consistência 1 para que o estado anterior da base de dados possa sempre ser recuperado (GRAY et al., 1976). Sob o grau de consistência 2, as transações não tem acesso aos elementos escritos por transações concorrentes ainda em andamento. Finalmente, o grau 3 torna uma transação também imune a alterações concorrentes sobre elementos já lidos.

Cada um dos graus de consistência requer que a transação adquira um bloqueio sobre o elemento antes de executar uma ação sobre este. Um bloqueio temporário é um bloqueio adquirido apenas pelo tempo suficiente para que a ação seja executada, ou seja, a leitura ou escrita do elemento. Por sua vez, um bloqueio longo é mantido por todo o tempo de vida da transação, ou seja, até que a mesma encerre.

Os níveis de isolamento propostos por GRAY et al. são definidos levando-se em conta um suporte transacional baseado em bloqueios. Por esse motivo, as especificações não podem ser utilizadas por implementações que adotam outro tipo de tecnologia, tais como abordagens otimistas ou baseadas em multiversão.

1.4 Norma ANSI

As especificações ANSI SQL-92 (ANSI, 1992) consideram algumas anomalias resultantes da execução intercalada de transações e identificam os fenômenos que dão ori-

gem a essas anomalias. O monitoramento desses fenômenos pode ser utilizado como um mecanismo para se evitar ações conflitantes e impedir a ocorrência de históricos não serializáveis.

Embora originalmente a norma ANSI tenha sido concebida de modo que outras tecnologias possam ser adotadas como suporte transacional, além das baseadas em bloqueio, suas especificações são ambíguas e a norma é incompleta (BERENSON et al., 1995). Por esse motivo, os fenômenos descritos nas próximas seções são apresentados considerando-se a revisão proposta para a norma.

1.4.1 Fenômenos

Um fenômeno pode ser entendido como a manifestação de um acesso concorrente sobre um elemento ou conjunto de elementos na base de dados e que pode dar origem a uma anomalia, como uma visão inconsistente dos dados ou até mesmo a violação de algumas restrições sobres estes (BERENSON et al., 1995). Para cada um dos fenômenos descritos nas seções seguintes é especificado o histórico H responsável pelo surgimento da anomalia, o fragmento F deste histórico utilizado na prevenção do fenômeno que a origina e um exemplo real de ocorrência do mesmo.

A prevenção com base nos fragmentos indicados restringe algumas vezes até mesmo históricos serializáveis, entretanto tal abordagem é necessária para se garantir que a anomalia decorrente de cada fenômeno de fato não ocorra (BERENSON et al., 1995), como originalmente previsto pelas especificações ANSI (ANSI, 1992). A notação[3] a seguir é utilizada para se descrever as ações presentes em um histórico:

- $w_i(x)$ — Indica a escrita de um elemento x pela transação T_i. O elemento x pode ser alvo, neste caso, de uma operação de inserção, atualização ou até mesmo exclusão.
- $w_i(x,v)$ — O mesmo que $w_i(x)$, porém o valor v escrito é especificado.
- $w_i(x:P)$ — O mesmo que $w_i(x)$, porém o elemento x sendo escrito satisfaz a condição imposta pelo predicado P.
- $r_i(x)$ — Indica a leitura de um elemento x pela transação T_i.
- $r_i(x,v)$ — O mesmo que $r_i(x)$, porém o valor v lido é especificado.
- $r_i(P)$ — Indica que a transação T_i faz a leitura de um conjunto de elementos que satisfazem a um predicado P.
- c_i — Indica um *commit* realizado pela transação T_i.
- a_i — Indica um *abort* realizado pela transação T_i.

[3]Esta é uma adaptação da notação utilizada por BERENSON et al., 1995.

1.4.1.1 Leitura Suja

O fenômeno Leitura Suja ocorre quando uma transação T_1 escreve um elemento e uma outra transação T_2 lê esse elemento antes mesmo que a transação T_1 finalize. Se por ventura T_1 finalizar com uma operação de *abort*, T_2 estará de posse de um valor que não foi salvo e de fato nunca existiu na base de dados. O histórico H_{LS} que desencadeia tal anomalia e o fragmento F_{LS} responsável pela prevenção do fenômeno Leitura Suja são dados a seguir.

H_{LS}: $w_1(x)...r_2(x)...\{a_1...c_2$ ou $c_2...a_1\}$
F_{LS}: $w_i(x)...r_j(x)$ $(i \neq j)$

Como exemplo, considere dois elementos distintos x e y na base de dados, os quais representam duas contas bancárias com valores $x = 50$ e $y = 50$. Considere agora duas transações T_1 e T_2 com as seguintes características:

– T_1 transfere um montante de \$40 da conta x para a conta y.
– T_2 consulta o saldo das duas contas.

Isoladamente as duas transações são naturalmente serializáveis e, portanto, produzem resultados consistentes. Entretanto, considere o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_1(x,50)$ $w_1(x,10)$ $r_2(x,10)$ $r_2(y,50)$ c_2 $r_1(y,50)$ $w_1(y,90)$ c_1

Embora a transação T_1 transfira corretamente o montante de \$40 da conta x para a conta y, a transação T_2 obtém saldos inconsistentes: $x = 10$ e $y = 50$, onde o balanço total deveria ser \$100. O histórico H é, portanto, não serializável. Neste exemplo o fenômeno Leitura Suja é identificável por meio do fragmento $w_1(x,10)$ $r_2(x,10)$.

1.4.1.2 Leitura não-Repetível

O fenômeno Leitura não-Repetível, também conhecido como Leitura Difusa, ocorre quando uma transação T_1 lê um elemento e uma outra transação T_2 modifica ou exclui esse elemento. Se a transação T_2 efetuar uma operação de *commit* e a transação T_1 tentar ler o elemento novamente, então T_1 obterá um valor diferente ou descobrirá que o elemento não existe mais. O histórico H_{LnR} que desencadeia tal anomalia e o fragmento F_{LnR} responsável pela prevenção do fenômeno Leitura não-Repetível são dados a seguir.

H_{LnR}: $r_1(x)...w_2(x)...c_2...r_1(x)...c_1$
F_{LnR}: $r_i(x)...w_j(x)$ $(i \neq j)$

Considere o mesmo exemplo de antes, onde dois elementos distintos x e y na base de dados representam duas contas bancárias com valores $x = 50$ e $y = 50$. Desta vez, entretanto, as transações T_1 e T_2 trocam de função:

– T_1 consulta o saldo das duas contas.

– T_2 transfere um montante de \$40 da conta x para a conta y.

Assim como antes, isoladamente as duas transações são naturalmente serializáveis e, portanto, produzem resultados consistentes. Considere agora o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_1(x,50)$ $r_2(x,50)$ $w_2(x,10)$ $r_2(y,50)$ $w_2(y,90)$ c_2 $r_1(y,90)$ c_1

Embora a transação T_2 transfira corretamente o montante de \$40 da conta x para a conta y, a transação T_1 obtém saldos inconsistentes: $x = 50$ e $y = 90$, onde o balanço total deveria ser \$100. O histórico H é, portanto, não serializável. Neste exemplo o fenômeno Leitura não-Repetível é identificável por meio do fragmento $r_1(x,50)$ $w_2(x,10)$.

1.4.1.3 Fantasma

O fenômeno Fantasma ocorre quando uma transação T_1 lê um conjunto de elementos que satisfazem a um predicado e uma outra transação T_2 atualiza a base dados de modo que novos elementos sejam cobertos pelo mesmo predicado. Se a transação T_2 efetuar uma operação de *commit* e a transação T_1 repetir a consulta, então T_1 obterá o conjunto de elementos anterior acrescido de novos elementos decorrentes das ações impostas por T_2. O histórico H_F que desencadeia tal anomalia e o fragmento F_F responsável pela prevenção do fenômeno Fantasma são dados a seguir.

H_F: $r_1(P)...w_2(x{:}P)...c_2...r_1(P)...c_1$
F_F: $r_i(P)...w_j(x{:}P)$ $(i \neq j)$

Vale ressaltar que o elemento x em questão não precisa necessariamente ser fruto de uma operação de inserção. Uma atualização ou mesmo exclusão que desencadeie o aparecimento de um novo elemento no conjunto obtido inicialmente por T_1 é o suficiente para que o fenômeno se manifeste.

Como exemplo, considere duas transações T_1 e T_2 com as seguintes características:

– T_1 obtém uma lista de funcionários ativos.
– T_2 insere um novo funcionário ativo x e atualiza o total de funcionários ativos y.

Isoladamente as duas transações são naturalmente serializáveis e, portanto, produzem resultados consistentes. Entretanto, considere o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_1(P)$ $w_2(x{:}P)$ $r_2(y,9)$ $w_2(y,10)$ c_2 $r_1(y,10)$ c_1

O valor lido em y pela transação T_1, que representa o total de funcionários ativos, é inconsistente com o número destes na lista originalmente obtida pela transação. O histórico H é, portanto, não serializável. Neste exemplo o fenômeno Fantasma é identificável por meio do fragmento $r_1(P)$ $w_2(x{:}P)$.

1.4.1.4 Escrita Suja

O fenômeno Escrita Suja ocorre quando uma transação T_1 escreve um elemento e uma outra transação T_2 escreve esse mesmo elemento antes de T_1 finalizar. Se quaisquer das duas transações encerrar com uma operação de *abort*, não ficará claro para o sistema de recuperação do banco de dados qual valor deve permanecer. O histórico H_{ES} que desencadeia tal anomalia e o fragmento F_{ES} responsável pela prevenção do fenômeno Escrita Suja são dados a seguir.

H_{ES}: $w_1(x)...w_2(x)...\{a_1$ ou $a_2\}$
F_{ES}: $w_i(x)...w_j(x)$ $(i \neq j)$

O fenômeno Escrita Suja pode violar restrições impostas sobre mais de um elemento. Como exemplo, considere dois elementos distintos x e y na base de dados, sobre os quais existe a seguinte restrição: $x = y$. Considere agora duas transações T_1 e T_2 com as seguintes características:

– T_1 modifica os valores de x e y de tal modo que $x = 1$ e $y = 1$.
– T_2 modifica os valores de x e y de tal modo que $x = 2$ e $y = 2$.

Isoladamente as duas transações não violam a restrição imposta sobre os elementos x e y e naturalmente são serializáveis. Entretanto, considere o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $w_1(x,1)$ $w_2(x,2)$ $w_2(y,2)$ c_2 $w_1(y,1)$ c_1

O histórico H não é serializável pois o resultado intercalado das ações produz no final $x = 2$ e $y = 1$, violando portando a restrição imposta sobre os elementos x e y. Neste exemplo o fenômeno Escrita Suja é identificável por meio do fragmento $w_1(x,1)$ $w_2(x,2)$.

1.4.1.5 Atualização Perdida

O fenômeno Atualização Perdida ocorre quando duas transações T_1 e T_2 atualizam um mesmo elemento com base em um único valor obtido inicialmente, ou seja, após a aquisição do valor, T_2 faz sua atualização finalizando com um *commit* e em seguida T_1 faz sua atualização também finalizando com um *commit*. Neste caso, mesmo a transação T_2 tendo encerrado, sua atualização sobre o elemento em questão é perdida. O histórico H_{AP} que desencadeia tal anomalia e o fragmento F_{AP} responsável pela prevenção do fenômeno Atualização Perdida são dados a seguir.

H_{AP}: $r_1(x)...r_2(x)...w_2(x)...c_2...w_1(x)...c_1$
F_{AP}: $r_i(x)...w_j(x)$ $(i \neq j)$
$F_{AP} = F_{LnR}$

Como exemplo, considere um elemento $x = 100$ na base de dados e duas transações T_1 e T_2 com as seguintes características:

– T_1 efetua um incremento de 10 em x.
– T_2 efetua um incremento de 20 em x.

Isoladamente as duas transações são naturalmente serializáveis e, portanto, produzem resultados consistentes. Entretanto, considere o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_1(x,100)$ $r_2(x,100)$ $w_2(x,120)$ c_2 $w_1(x,110)$ c_1

O histórico H não é serializável pois o resultado intercalado das operações produz no final $x = 110$ ao invés de $x = 130$, ou seja, a atualização feita por T_2 foi perdida mesmo T_2 tendo efetuado um *commit*. Neste exemplo o fenômeno Atualização Perdida é consequência direta do fenômeno Leitura não-Repetível, identificável por meio do fragmento $r_1(x,100)$ $w_2(x,120)$.

1.4.1.6 Leitura Oblíqua

O fenômeno Leitura Oblíqua ocorre quando dois elementos na base de dados estão sob alguma restrição e uma transação T_1 efetua a leitura desses elementos enquanto uma outra transação T_2 os atualiza. Se T_1 lê um dos elementos e logo em seguida T_2 atualiza os dois elementos finalizando com uma operação de *commit*, então T_1 poderá ter uma visão inconsistente da base de dados no momento em que ler o outro elemento. O histórico H_{LO} que desencadeia tal anomalia e o fragmento F_{LO} responsável pela prevenção do fenômeno Leitura Oblíqua são dados a seguir.

H_{LO}: $r_1(x)...w_2(x)...w_2(y)...c_2...r_1(y)...c_1$
F_{LO}: $r_i(x)...w_j(x)$ $(i \neq j)$
$F_{LO} = F_{LnR}$

Como exemplo, considere dois elementos distintos x e y na base de dados, sobre os quais existe a seguinte restrição: $x > y$. Inicialmente tem-se $x = 1$ e $y = 0$. Considere agora duas transações T_1 e T_2 com as seguintes características:

– T_1 consulta os valores de x e y.
– T_2 modifica os valores de x e y de modo que $x = 2$ e $y = 1$.

Isoladamente as duas transações não violam a restrição imposta sobre os elementos x e y e naturalmente são serializáveis. Entretanto, considere o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_1(x,1)$ $w_2(x,2)$ $w_2(y,1)$ c_2 $r_1(y,1)$ c_1

Embora o resultado intercalado das operações leve a base de dados a um estado consistente, onde $x = 2$ e $y = 1$, a transação T_1 se depara com dados inconsistentes, obtendo $x = 1$ e $y = 1$, o que não condiz com a restrição imposta sobre os elementos x e y. Portanto, o histórico H não é serializável. Neste exemplo o fenômeno Leitura Oblíqua

também é consequência direta do fenômeno Leitura não-Repetível, identificável por meio do fragmento $r_1(x,1)$ $w_2(x,2)$.

1.4.1.7 Escrita Oblíqua

O fenômeno Escrita Oblíqua ocorre quando dois elementos na base de dados estão sob alguma restrição e uma transação T_1 atualiza um desses elementos enquanto uma transação T_2 atualiza o outro. Neste caso a base de dados poderá ser levada a um estado inconsistente. O histórico H_{EO} que desencadeia tal anomalia e o fragmento F_{EO} responsável pela prevenção do fenômeno Escrita Oblíqua são dados a seguir.

H_{EO}: $r_1(x)...r_2(y)...w_1(y)...w_2(x)...\{c_1...c_2$ ou $c_2...c_1\}$
F_{EO}: $r_i(x)...w_j(x)$ $(i \neq j)$
$F_{EO} = F_{LnR}$

Como exemplo, considere dois elementos distintos x e y na base de dados, os quais representam duas contas bancárias com valores $x = 50$ e $y = 50$. Desta vez há uma restrição sobre estas duas contas de modo que $x + y > 0$, ou seja, uma das contas pode ficar com saldo negativo desde que a outra conta tenha saldo suficiente para cobrir o débito. Considere agora duas transações T_1 e T_2 com as seguintes características:

– T_1 faz uma retirada de \$60 da conta x.
– T_2 faz uma retirada de \$60 da conta y.

Isoladamente apenas uma das transações seria finalizada com sucesso, o que levaria o saldo das contas a ser $x = -10$ e $y = 50$, caso T_1 executasse primeiro, ou $x = 50$ e $y = -10$, caso T_2 executasse primeiro. Em quaisquer dos casos a restrição $x + y > 0$ seria mantida e portanto a base de dados permaneceria em um estado consistente. Entretanto, considere o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_1(x,50)$ $r_1(y,50)$ $r_2(x,50)$ $r_2(y,50)$ $w_1(x,-10)$ c_1 $w_2(y,-10)$ c_2

A transação T_1 verifica que a retirada solicitada de \$60 é maior do que o saldo da conta x, mas ainda assim concede a requisição pois a conta y pode cobrir o restante. No momento em que T_1 finaliza com um *commit*, a base de dados ainda encontra-se em um estado consistente: $x + y > 0$. A transação T_2 paralelamente faz o mesmo, porém a retirada é feita da conta y. Finalmente, quando T_2 finaliza com um *commit*, o saldo das duas contas passa a ser $x = -10$ e $y = -10$, violando portanto a restrição $x + y > 0$ e levando a base de dados a um estado inconsistente. O histórico H é, portanto, não serializável. Neste exemplo o fenômeno Escrita Oblíqua também é consequência direta do fenômeno Leitura não-Repetível, identificável por meio do fragmento $r_2(x,50)$ $w_1(x,-10)$.

1.4.2 Níveis de isolamento ANSI

Com base apenas nos fenômenos Leitura Suja, Leitura não-Repetível e Fantasma, as especificações ANSI originais (ANSI, 1992) estabelecem quatro níveis de isolamento:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

Cada um dos níveis é definido considerando-se os fenômenos que podem ou não se manifestar. Esses níveis de isolamento e os respectivos fenômenos permitidos são apresentados na Tabela 1, onde os conceitos originalmente reconhecidos pela norma ANSI (ANSI, 1992) estão destacados em fundo cinza e os demais são oriundos da revisão proposta (BERENSON et al., 1995) e de um comparativo com o trabalho "Graus de Consistência" (GRAY et al., 1976).

Tabela 1 – Níveis de isolamento ANSI e fenômenos possíveis.

		LS	LnR	F	ES	AP	LO	EO
Grau 0		✓	✓	✓	✓	✓	✓	✓
Grau 1	Read Uncommitted	✓	✓	✓		✓	✓	✓
Grau 2	Read Committed		✓	✓		✓	✓	✓
Grau 3	Repeatable Read			✓				
	Serializable							

LS = Leitura Suja; LnR = Leitura não-Repetível; F = Fantasma; ES = Escrita Suja;
AP = Atualização Perdida; LO = Leitura Oblíqua; EO = Escrita Oblíqua

Sob o nível de isolamento Read Uncommitted, uma transação é imune apenas ao fenômeno Escrita Suja. Por sua vez, uma transação sob o nível de isolamento Read Committed é imune aos fenômenos Leitura Suja e Escrita Suja. Transações sob o nível de isolamento Repeatable Read estão sujeitas apenas ao fenômeno Fantasma. Serializable é o nível mais forte, onde nenhum fenômeno pode se manifestar e, portanto, a serialização é garantida.

Os níveis de isolamento Read Uncommitted, Read Committed e Repeatable Read, correspondem aos graus de consistência 1, 2 e 3, respectivamente. Essa correspondência só é possível mediante a adoção da revisão proposta para a norma ANSI, pois conforme é explicado na próxima seção, a revisão torna as especificações ANSI voltadas a um suporte transacional baseado em bloqueios. Ao contrário do exposto em BERENSON et al. 1995, o grau 3 não foi atribuído como equivalente ao isolamento Serializable neste trabalho em virtude de sua definição original em "Graus de Consistência" (GRAY et al., 1976) não impedir o fenômeno Fantasma.

Nota-se na Tabela 1 que as especificações ANSI não definem um nível de isolamento compatível com o grau de consistência 0. Desta forma, qualquer sistema transacional que adote a norma ANSI deve oferecer apenas níveis de isolamento sob os quais as transações sejam reversíveis.

1.4.3 Considerações

Cada fenômeno pode levar a uma anomalia na base de dados e, portanto, deve ser evitado. Neste contexto, anomalia deve ser entendido como um histórico resultante não serializável devido à ocorrência de um ou mais conflitos. Os históricos que desencadeiam cada uma das anomalias descritas e os fragmentos responsáveis pela prevenção dos fenômenos que dão origem a essas anomalias estão resumidos na Tabela 2. Observa-se na tabela que, se uma transação é imune ao fenômeno Leitura não-Repetível por conta do rastreamento do padrão $r_i(x)...w_j(x)$, então a mesma será imune aos fenômenos Atualização Perdida, Leitura Oblíqua e Escrita Oblíqua.

Tabela 2 – Históricos que desencadeiam a ocorrência de anomalias e seus respectivos fragmentos utilizados na prevenção dos fenômenos que as originam.

Fenômeno	Histórico que desencadeia a anomalia	Prevenção ($i \neq j$)
Leitura Suja	$w_1(x)...r_2(x)...\{a_1...c_2$ ou $c_2...a_1\}$	$w_i(x)...r_j(x)$
Leitura não-Repetível	$r_1(x)...w_2(x)...c_2...r_1(x)...c_1$	$r_i(x)...w_j(x)$
Fantasma	$r_1(P)...w_2(x{:}P)...c_2...r_1(P)...c_1$	$r_i(P)...w_j(x{:}P)$
Escrita Suja	$w_1(x)...w_2(x)...\{a_1$ ou $a_2\}$	$w_i(x)...w_j(x)$
Atualização Perdida	$r_1(x)...r_2(x)...w_2(x)...c_2...w_1(x)...c_1$	$r_i(x)...w_j(x)$
Leitura Oblíqua	$r_1(x)...w_2(x)...w_2(y)...c_2...r_1(y)...c_1$	$r_i(x)...w_j(x)$
Escrita Oblíqua	$r_1(x)...r_2(y)...w_1(y)...w_2(x)...\{c_1...c_2$ ou $c_2...c_1\}$	$r_i(x)...w_j(x)$

Ainda que a ocorrência de um fenômeno não implique, necessariamente, no surgimento de uma anomalia, a prevenção desses com base nos fragmentos indicados na Tabela 2 possibilita a construção de um sistema transacional capaz de garantir serialização (BERENSON et al., 1995). Entretanto, tal abordagem é agressiva no sentido de que alguns possíveis históricos serializáveis também são impedidos e, consequentemente, há uma redução na carga de transações concorrentes suportadas pelo sistema (YABANDEH; FERRO, 2012).

Contudo, os fragmentos de prevenção sugeridos pela revisão proposta para a norma ANSI tornam suas especificações fortemente direcionadas à utilização de bloqueios. Impedir a ocorrência de padrões do tipo $w_i(x)...r_j(x)$ ou $w_i(x)...w_j(x)$ é, de fato, o mesmo que inserir um bloqueio exclusivo sobre o elemento x para que uma transação concorrente não possa lê-lo ou escrevê-lo. De modo análogo, impedir a ocorrência de padrões do tipo $r_i(x)...w_j(x)$ ou $r_i(P)...w_j(x{:}P)$ é o mesmo que inserir um bloqueio compartilhado sobre o elemento x para que uma transação concorrente consiga lê-lo, mas não escrevê-lo. Por esta razão a revisão corrige a norma, mas fere seu objetivo principal de tornar as especifi-

cações independentes da tecnologia utilizada como suporte transacional (ADYA; LISKOV; O'NEIL, 2000).

Em geral, consistência e concorrência são dois extremos em um sistema transacional baseado em bloqueios. Enquanto o isolamento mais fraco (READ UNCOMMITTED) permite a manifestação de um número maior de fenômenos, o mesmo impede menos históricos que o isolamento mais forte (SERIALIZABLE), onde nenhum fenômeno pode se manifestar. Desta forma, conforme esboçado na Figura 2, quanto mais forte é o nível de isolamento, maior é a consistência, porém menor é a concorrência (YABANDEH; FERRO, 2012; ADYA; LISKOV; O'NEIL, 2000).

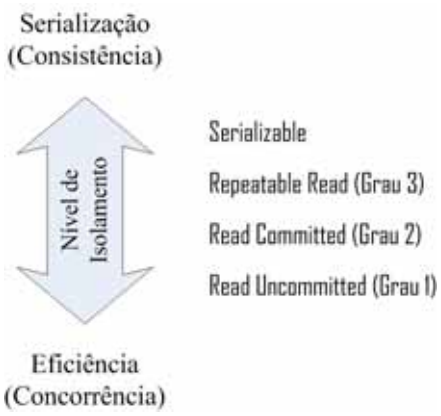

Figura 2 – Níveis de isolamento e suas relações com consistência e concorrência.

Transações que executam sob um nível de isolamento mais fraco podem se deparar com uma visão inconsistente dos dados (BERENSON et al., 1995). Se tal visão é utilizada direta ou indiretamente na atualização da base de dados, então a mesma também pode ser levada a um estado inconsistente. Considerando-se este aspecto, o nível de isolamento adequado é aquele capaz de permitir alta concorrência dentro de um contexto totalmente serializável. Conforme observa-se na Figura 2, este parece um objetivo impossível de se atingir em sistemas onde o suporte transacional é baseado em bloqueios (YABANDEH; FERRO, 2012).

1.5 Especificações portáveis

Embora a norma ANSI revisada seja completa e correta, seus níveis de isolamento possuem especificações preventivas, altamente restritivas e direcionadas a sistemas onde o suporte transacional é baseado em bloqueios. Por essa razão, de modo a se obter um padrão na definição de níveis de isolamento que seja menos restritivo e também portável para outros tipos de tecnologia transacional, um novo conjunto de especificações foi formulado com base em restrições sobre históricos e grafos de serialização (ADYA; LISKOV; O'NEIL, 2000; ADYA, 1999). Ao contrário das estratégias anteriores, onde alguns

históricos serializáveis são impedidos preventivamente de modo a se garantir a consistência imposta por cada nível de isolamento, as novas especificações permitem um número maior destes até mesmo sob o nível de isolamento mais forte.

Por possibilitar a adoção por sistemas onde o suporte transacional é baseado em multiversão, as ações em um histórico devem ser modeladas considerando-se a possibilidade de existência de múltiplas versões dos elementos na base de dados, exigindo, portanto, uma notação diferente da utilizada anteriormente. A notação a seguir é sugerida por ADYA; LISKOV; O'NEIL, 2000:

- $\mathrm{w}_i(x_i)$ — Indica a escrita de um elemento x pela transação T_i. O elemento x pode ser alvo de uma operação de inserção, atualização ou até mesmo exclusão. Uma nova versão de um elemento na base de dados é criada sempre que este é alvo de uma operação de escrita. A notação x_i representa a última versão do elemento x criada por T_i.

- $\mathrm{w}_i(x_{i.n})$ — O mesmo que $\mathrm{w}_i(x_i)$, porém indica a n-ésima vez que o elemento x é escrito dentro do contexto de execução da transação T_i.

- $\mathrm{w}_i(x_i,\mathrm{v})$ — Idem a $\mathrm{w}_i(x_i)$, porém o valor v escrito é especificado.

- $\mathrm{r}_j(x_i)$ — Indica a leitura da transação T_j de uma versão do elemento x criada pela transação T_i.

- $\mathrm{r}_j(x_{i.n})$ — O mesmo que $\mathrm{r}_j(x_i)$, porém indica a leitura da n-ésima escrita de x feita pela transação T_i.

- $\mathrm{r}_j(x_i,\mathrm{v})$ — Idem a $\mathrm{r}_j(x_i)$, porém o valor v lido é especificado.

- $\mathrm{r}_i(\mathrm{P{:}Vset(P)})$ $\mathrm{r}_i(x_j)$ $\mathrm{r}_i(y_k)$ ... — Indica a leitura de um conjunto de elementos por uma transação T_i com base em um predicado P. De modo a avaliar a condição imposta pelo predicado, o sistema deve selecionar a versão apropriada de cada elemento participante da requisição para somente depois verificar se tal elemento atende ou não ao predicado. O conjunto dessas versões é denotado por Vset(P). No exemplo em questão, os elementos x_j e y_k são as versões em Vset(P) que satisfazem a condição imposta por P, ou seja, $\{x_j, y_k\} \subseteq \mathrm{Vset(P)}$.

- $\mathrm{r}_i(\mathrm{P{:}Vset(P)})$ $\mathrm{w}_i(x_i)$ $\mathrm{w}_i(y_{i,\mathrm{dead}})$ ... — Indica a escrita de um conjunto de elementos por uma transação T_i com base em um predicado P. No exemplo, o elemento x_i é fruto de uma operação de inserção ou atualização e o elemento y_i representa a exclusão de y. Quando uma transação remove um elemento, uma versão especial deste é criada de modo a representar sua morte: *dead*[4].

[4] Algumas vezes denominada *tombstone* (lápide).

- c_i — Indica um *commit* realizado pela transação T_i.
- a_i — Indica um *abort* realizado pela transação T_i.

1.5.1 Conflitos e dependências

Ao invés de considerar apenas transações concorrentes, o conjunto de especificações portáveis proposto utiliza uma abordagem na qual identificam-se os conflitos diretos envolvendo as últimas transações que agiram sobre uma determinada versão de um elemento e encerraram com sucesso. Três tipos de conflitos são considerados: write-read (wr), read-write (rw) e write-write (ww). Cada um dos conflitos gera uma dependência entre duas transações, onde uma das transações é responsável pela escrita do elemento e a outra é responsável por sua leitura ou sobrescrita.

Uma dependência de leitura ocorre quando há um conflito do tipo wr, ou seja, uma transação lê um elemento que foi escrito por outra transação. Desta forma, uma transação T_j possui uma dependência de leitura em relação a uma transação T_i ($T_i \xrightarrow{wr} T_j$) se T_i instalar[5] na base de dados um elemento x_i e T_j ler este elemento x_i ou possuir uma leitura baseada em predicado que seja afetada pela instalação de x_i.

Quando uma transação sobrescreve um elemento que foi lido por outra transação, diz-se que há um conflito do tipo rw ou uma antidependência. Desta forma, uma transação T_j possui uma antidependência em relação a uma transação T_i ($T_i \xrightarrow{rw} T_j$) se T_j instalar uma nova versão de um elemento que tenha sido anteriormente lido por T_i ou que afete uma leitura baseada em predicado feita por T_i.

Uma dependência de escrita ocorre quando há um conflito do tipo ww, ou seja, uma transação sobrescreve um elemento que foi escrito por outra transação. Desta forma, uma transação T_j possui uma dependência de escrita em relação a uma transação T_i ($T_i \xrightarrow{ww} T_j$) se T_i instalar na base de dados um elemento x_i e T_j instalar a versão seguinte deste elemento x_j. De acordo com ADYA; LISKOV; O'NEIL, 2000, não existe dependência de escrita baseada em predicado em virtude de tais ações serem modeladas como uma leitura envolvendo um predicado seguida de operações de escrita sobre elementos individuais.

1.5.2 Grafo de serialização direta

Considerando-se as dependências possíveis entre duas transações, ADYA; LISKOV; O'NEIL, 2000 definem um "Grafo de Serialização Direta" como sendo um grafo onde os nós correspondem a transações em um histórico que fizeram *commit* e que contém arestas direcionadas representando os conflitos diretos entre essas transações, ou seja, as dependências estabelecidas entre as mesmas. Enquanto o histórico registra as ações de todas as

[5] Neste contexto, instalar refere-se ao ato de inserir um elemento definitivamente na base de dados, ou seja, o elemento é escrito e a transação que o escreveu é encerrada com uma operação de *commit*.

transações, inclusive as que ainda estão em execução, o grafo de serialização direta registra apenas as dependências diretas entre as transações que encerraram com sucesso. Como exemplo, considere três transações T_1, T_2, T_3 e o histórico H a seguir como resultado da execução intercalada de suas ações.

H: $w_1(x_1)$ $w_1(y_1)$ $w_1(z_1)$ $w_3(x_3)$ c_1 $r_2(x_1)$ $w_2(y_2)$ c_2 $r_3(y_2)$ $w_3(z_3)$ c_3

O histórico é serializável na ordem T_1, T_2, T_3. O grafo de serialização direta resultante do histórico H é exibido na Figura 3 e as dependências estabelecidas entre as transações são descritas a seguir:

- T_1: $w_1(x_1)$ $w_1(y_1)$ $w_1(z_1)$ c_1 — A transação T_1 escreve os elementos x, y e z gerando uma versão de cada (x_1, y_1 e z_1). Como não há registro de processamento anterior, nenhuma dependência é estabelecida.

- T_2: $r_2(x_1)$ $w_2(y_2)$ c_2 — A transação T_2 inicia sua execução lendo a versão de um elemento escrita por T_1 (x_1). Neste caso T_2 passa a ter uma dependência de leitura em relação a T_1 ($T_1 \xrightarrow{wr} T_2$). Em seguida T_2 escreve uma nova versão do elemento y (y_2), a qual substitui a versão y_1 escrita por T_1 e faz com que T_2 adquira uma dependência de escrita em relação a T_1 ($T_1 \xrightarrow{ww} T_2$).

- T_3: $w_3(x_3)$ $r_3(y_2)$ $w_3(z_3)$ c_3 — A transação T_3 atualiza o elemento x gerando uma nova versão deste (x_3), a qual substitui a versão anterior escrita por T_1 (x_1) e faz com que T_3 adquira uma dependência de escrita em relação a T_1 ($T_1 \xrightarrow{ww} T_3$). Entretanto, como T_2 havia feito a leitura da versão x_1, T_3 adquiri uma antidependência em relação a T_2 ($T_2 \xrightarrow{rw} T_3$). Em seguida T_3 lê uma versão do elemento y escrita por T_2 (y_2), fazendo com que T_3 também tenha uma dependência de leitura em relação a T_2 ($T_2 \xrightarrow{wr} T_3$). Finalmente, a transação T_3 atualiza o elemento z gerando uma nova versão deste (z_3). Nesse momento T_3 adquiriria outra dependência de escrita em relação a T_1 ($T_1 \xrightarrow{ww} T_3$) devido a versão z_3 substituir a versão anterior z_1 escrita por T_1, mas tal dependência não é adicionada pois o grafo de serialização direta registra no máximo um de cada tipo de dependência possível entre transações.

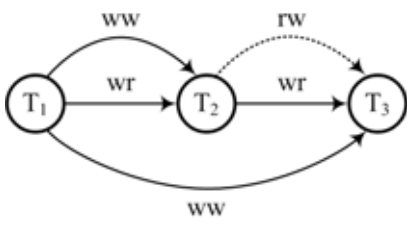

Figura 3 – Grafo de serialização direta resultante da execução intercalada das transações T_1, T_2 e T_3.

Conforme pode-se observar na Figura 3, a ausência de um ciclo no grafo prova que o histórico que deu origem a este é serializável na ordem T_1, T_2, T_3. Cada uma das dependências existentes modela o fluxo de dados inter-transacional.

1.5.3 Fenômenos

Assim como na norma ANSI, os níveis de isolamento portáveis são definidos considerando-se os fenômenos que podem ou não se manifestar. Entretanto, esses fenômenos não são os mesmos levantados pelas especificações ANSI, embora em muitos deles haja uma certa similaridade semântica.

Os fenômenos descritos a seguir (ADYA; LISKOV; O'NEIL, 2000; ADYA, 1999) apresentam propriedades que os identificam ou por meio de um fragmento do histórico ou com base em determinados padrões de ciclos que ocorrem no grafo de serialização direta decorrente deste mesmo histórico.

1.5.3.1 Ciclo de Escrita

O fenômeno Ciclo de Escrita ocorre em um histórico H se o grafo de serialização direta deste histórico tiver um ciclo direcionado contendo apenas dependências de escrita. Este fenômeno é representado na Figura 4.

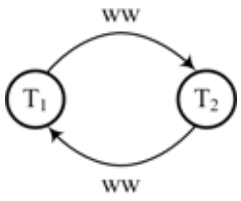

Figura 4 – Manifestação do fenômeno Ciclo de Escrita.

Intuitivamente, uma transação T_2 pode sobrescrever um elemento escrito por uma transação T_1 ainda em andamento apenas se a transação T_1 não fizer o mesmo em relação a transação T_2. Desta forma, ao contrário do fenômeno Escrita Suja nas especificações ANSI, o fenômeno Ciclo de Escrita permite que transações concorrentes modifiquem os mesmos elementos desde que não os façam em ordem oposta.

A especificação do fenômeno Ciclo de Escrita possibilita a implementação de sistemas transacionais otimistas ou multiversão. Uma implementação baseada em bloqueios naturalmente não permitirá a ocorrência do fenômeno em virtude de apenas uma transação por vez ter o direito de escrever um determinado elemento, direito este adquirido por intermédio da aquisição de um bloqueio exclusivo sobre o mesmo.

1.5.3.2 Leitura Abortada

O fenômeno Leitura Abortada ocorre quando uma transação T_1 escreve uma versão de um elemento x e uma outra transação T_2 lê essa versão antes mesmo que T_1 finalize. Se por ventura T_1 finalizar com uma operação de *abort*, T_2 estará de posse de um valor que não foi salvo e de fato nunca existiu na base de dados. O histórico H_{LA} que desencadeia tal anomalia e o fragmento F_{LA} responsável pela identificação do fenômeno Leitura Abortada são dados a seguir.

H_{LA}: $w_1(x_{1.n})...r_2(x_{1.n})...a_1$
F_{LA}: $w_i(x_{i.n})...r_j(x_{i.n})...a_i$ $(i \neq j)$

A identificação deve ser feita de modo que se a transação T_i abortar, T_j também deve abortar. Além disso, uma operação de *commit* solicitada por T_j deve aguardar até que T_i efetue seu próprio *commit* com sucesso.

1.5.3.3 Leitura Intermediária

O fenômeno Leitura Intermediária ocorre quando uma transação T_1 escreve uma versão de um elemento x, uma transação T_2 lê essa versão e T_1, mais uma vez, atualiza o elemento x gerando outra versão. Se por ventura T_2 finalizar com uma operação de *commit*, T_2 terá considerado uma versão de x que de fato não corresponde a modificação final feita por T_1. O histórico H_{LI} que desencadeia tal anomalia e o fragmento F_{LI} responsável pela identificação do fenômeno Leitura Intermediária são dados a seguir.

H_{LI}: $w_1(x_{1.m})...r_2(x_{1.m})...w_1(x_{1.n})...c_2$
F_{LI}: $w_i(x_{i.m})...r_j(x_{i.m})...w_i(x_{i.n})$ $(i \neq j)$

A identificação do fenômeno Leitura Intermediária garante que uma transação possa fazer *commit* somente se tiver acessado as últimas versões dos elementos escritos por outras transações.

1.5.3.4 Fluxo de Informação Circular

O fenômeno Fluxo de Informação Circular ocorre em um histórico H se o grafo de serialização direta deste histórico tiver um ciclo direcionado contendo apenas dependências de leitura ou escrita. Intuitivamente, a ocorrência do fenômeno indica que duas transações T_1 e T_2 se afetam mutuamente. Este fenômeno é representado na Figura 5.

Juntos, os fenômenos Leitura Abortada, Leitura Intermediária e Fluxo de Informação Circular constituem a essência de prevenção do fenômeno Leitura Suja segundo a norma ANSI, porém suas especificações permitem a leitura de elementos escritos por transações concorrentes ainda em andamento e, portanto, são menos restritivos e possibilitam sua adoção por outros tipos de tecnologia transacional além das baseadas em bloqueios.

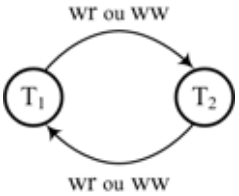

Figura 5 – Manifestação do fenômeno Fluxo de Informação Circular.

1.5.3.5 Ciclo com Antidependência de Item

O fenômeno Ciclo com Antidependência de Item ocorre em um histórico H se o grafo de serialização direta deste histórico tiver um ciclo direcionado contendo pelo menos uma antidependência que não envolva a ação de predicados, ou seja, uma antidependência baseada apenas em elementos individuais (itens de dados). Este fenômeno é representado na Figura 6.

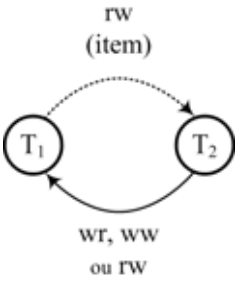

Figura 6 – Manifestação do fenômeno Ciclo com Antidependência de Item.

Ao contrário do fenômeno Leitura não-Repetível da norma ANSI, a especificação proposta para o fenômeno Ciclo com Antidependência de Item permite que uma transação T_2 modifique um elemento mesmo este tendo sido lido anteriormente por uma transação T_1 ainda em andamento, desde que T_1 não possua qualquer tipo de dependência em relação a T_2. Mais uma vez, a definição possibilita a adoção de outro tipo de tecnologia transacional além da baseada em bloqueios.

1.5.3.6 Ciclo com Antidependência

O fenômeno Ciclo com Antidependência ocorre em um histórico H se o grafo de serialização direta deste histórico tiver um ciclo direcionado contendo pelo menos uma antidependência, seja esta baseada ou não em predicados. Este fenômeno é representado na Figura 7.

Além de apresentar propriedades semelhantes ao fenômeno Ciclo com Antidependência de Item, o fenômeno Ciclo com Antidependência é capaz de lidar com o aparecimento de elementos fantasmas em virtude de sua especificação considerar a ação de predicados.

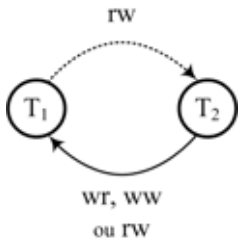

Figura 7 – Manifestação do fenômeno Ciclo com Antidependência.

1.5.4 Níveis de isolamento portáveis

Com base nos fenômenos descritos, os seguintes níveis de isolamento portáveis (PL — Portable Level) são estabelecidos como representantes diretos dos isolamentos ANSI (ADYA; LISKOV; O'NEIL, 2000; ADYA, 1999):

- PL–1
- PL–2
- PL–2.99
- PL–3

Assim como nas especificações ANSI, cada um dos níveis é definido considerando-se os fenômenos que podem ou não se manifestar. Esses níveis de isolamento e os respectivos fenômenos permitidos são apresentados na Tabela 3.

Tabela 3 – Níveis de isolamento portáveis e fenômenos possíveis.

			CE	LA	LI	FIC	CADi	CAD
Grau 0			✓	✓	✓	✓	✓	✓
Grau 1	Read Uncommitted	PL-1		✓	✓	✓	✓	✓
Grau 2	Read Committed	PL-2					✓	✓
Grau 3	Repeatable Read	PL-2.99						✓
	Serializable	PL-3						

CE = Ciclo de Escrita; LA = Leitura Abortada; LI = Leitura Intermediária;
FIC = Fluxo de Informação Circular; CADi = Ciclo com Antidependência de Item;
CAD = Ciclo com Antidependência

Sob o nível de isolamento PL–1, uma transação é imune apenas ao fenômeno Ciclo de Escrita. Por sua vez, uma transação sob o nível de isolamento PL–2 é imune aos fenômenos Ciclo de Escrita, Leitura Abortada, Leitura Intermediária e Fluxo de Informação Circular. Transações sob o nível de isolamento PL–2.99 estão sujeitas apenas ao fenômeno Ciclo com Antidependência. PL–3 é o nível mais forte, onde nenhum fenômeno pode se manifestar e, portanto, a serialização é garantida.

Conforme nota-se na Tabela 3, os níveis de isolamento PL–1, PL–2, PL–2.99 e PL–3 são representantes diretos dos isolamentos Read Uncommitted, Read Committed, Repeatable Read e Serializable.

1.5.5 Considerações

Diferentemente dos fenômenos reconhecidos pelas especificações ANSI, os quais são utilizados como prevenção de suas anomalias resultantes, os fenômenos reconhecidos pelas especificações portáveis representam, de fato, a ocorrência real de uma anomalia, sendo portanto utilizados não como prevenção, mas como identificação certa de um problema de concorrência. Esses fenômenos são identificáveis ou por meio de fragmentos do próprio histórico ou pela ocorrência de determinados tipos de ciclos no grafo de serialização direta resultante do histórico. Os fenômenos e seus respectivos mecanismos de identificação estão resumidos na Tabela 4.

Tabela 4 – Fenômenos possíveis de acordo com as especificações portáveis e seus respectivos mecanismos de identificação.

Fenômeno	Mecanismo de identificação ($i \neq j$ e $m \neq n$)
Ciclo de Escrita	Ciclo com dependências ww
Leitura Abortada	$\mathrm{w}_i(x_{i.n})...\mathrm{r}_j(x_{i.n})...\mathrm{a}_i$
Leitura Intermediária	$\mathrm{w}_i(x_{i.m})...\mathrm{r}_j(x_{i.m})...\mathrm{w}_i(x_{i.n})$
Fluxo de Informação Circular	Ciclo com dependências wr ou ww
Ciclo com Antidependência de Item	Ciclo com pelo menos uma antidependência (item-rw)
Ciclo com Antidependência	Ciclo com pelo menos uma antidependência (rw)

1.6 Considerações finais

A proposta original para níveis de isolamento (GRAY et al., 1976) introduziu quatro graus de consistência, onde o objetivo foi dar a possibilidade de escolha entre a concorrência desejada e a consistência necessária. Entretanto, as especificações foram direcionadas exclusivamente a implementações baseadas em bloqueios.

De modo a oferecer um padrão de níveis de isolamento que fosse independente da tecnologia adotada como suporte transacional, a norma ANSI (ANSI, 1992) identifica um conjunto de anomalias resultantes da execução intercalada de transações e define os níveis de isolamento com base em fenômenos que dão origem a essas anomalias. Cada um dos níveis é especificado pela norma ANSI considerando-se os fenômenos que podem ou não se manifestar.

BERENSON et al., 1995 mostram que as especificações ANSI são ambíguas e incompletas, propondo correções à norma. Entretanto, a revisão torna as especificações ANSI fortemente direcionadas a implementações baseadas em bloqueios, violando portanto o objetivo principal da norma de ser independente da tecnologia adotada como suporte

transacional. Considerando-se esta revisão, as abordagens até então apresentadas são puramente preventivas e altamente restritivas, pois garantem a consistência desejada em cada nível de isolamento ao custo de impedir, preventivamente, ações que também poderiam levar a históricos serializáveis.

Finalmente, em ADYA; LISKOV; O'NEIL, 2000 novas especificações são propostas como um padrão na definição de níveis de isolamento, as quais, além de menos restritivas, podem ser aplicadas a implementações baseadas em bloqueios, otimista ou multiversão. A abordagem faz uso de restrições sobre históricos e grafos de serialização, possibilitando que todo histórico serializável seja permitido até mesmo sobre o nível de isolamento mais forte. Embora as especificações sejam completas, corretas e independentes da tecnologia adotada pelo suporte transacional, seu custo de implementação pode ser altamente proibitivo em virtude da sobrecarga associada à identificação de determinados padrões de ciclos nos grafos de serialização (YABANDEH; FERRO, 2012).

As especificações propostas para os níveis de isolamento estão resumidas na Tabela 5 considerando-se todos os fenômenos vistos, onde a possibilidade de ocorrência ou não destes é discriminada de duas formas: o fenômeno ocorre em quaisquer dos níveis propostos ou apenas no isolamento portável em questão. Na tabela, os conceitos com fundo cinza são originalmente reconhecidos pela norma ANSI (ANSI, 1992) e os demais são oriundos da revisão proposta (BERENSON et al., 1995) e de um comparativo com os trabalhos "Graus de Consistência" (GRAY et al., 1976) e "Especificações Portáveis" (ADYA; LISKOV; O'NEIL, 2000).

Tabela 5 – Níveis de isolamento e fenômenos possíveis.

			Fenômenos identificados pelas especificações ANSI revisadas							Fenômenos identificados pelas especificações portáveis					
			LS	LnR	F	ES	AP	LO	EO	CE	LA	LI	FIC	CADi	CAD
Grau 0			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Grau 1	Read Uncommitted	PL-1	✓	✓	✓	PL	✓	✓	✓		✓	✓	✓	✓	✓
Grau 2	Read Committed	PL-2	PL	✓	✓	PL	✓	✓	✓					✓	✓
Grau 3	Repeatable Read	PL-2.99	PL		✓	PL									✓
	Serializable	PL-3	PL			PL									

LS = Leitura Suja; LnR = Leitura não-Repetível; F = Fantasma; ES = Escrita Suja;
AP = Atualização Perdida; LO = Leitura Oblíqua; EO = Escrita Oblíqua

CE = Ciclo de Escrita; LA = Leitura Abortada; LI = Leitura Intermediária; FIC = Fluxo de Informação Circular;
CADi = Ciclo com Antidependência de Item; CAD = Ciclo com Antidependência

✓ = O fenômeno ocorre em todos os níveis de isolamento
PL = O fenômeno ocorre apenas no nível de isolamento portável em questão

Nota-se na Tabela 5 que todos os níveis portáveis estão sujeitos aos fenômenos ANSI Leitura Suja e Escrita Suja, fruto da capacidade que é dada a transações que executam segundo as especificações propostas para esses níveis de suportar a leitura e escrita de elementos ainda sob o domínio de outras transações, tornando-os menos restritivos que

os isolamentos ANSI correspondentes.

Outros níveis de isolamento comumente encontrados em bancos de dados não são considerados pela norma ANSI, tais como Cursor Stability e Snapshot (BERENSON et al., 1995).

Cursor Stability estende o nível de isolamento Read Committed impedindo que o fenômeno Atualização Perdida se manifeste quando da manipulação dos elementos na base de dados por meio de cursores (BERENSON et al., 1995). Para isso é mantido um bloqueio sobre o elemento atualmente em uso (*cursor*) de modo que o dado possa ser lido e atualizado apenas pela transação que detém esse bloqueio. Tão logo o cursor seja movido, o bloqueio é liberado e o elemento pode ser lido e modificado por outra transação, a não ser que o mesmo tenha sido alterado, e neste caso o bloqueio será removido somente após a transação que o alterou efetuar uma operação de *commit* ou *abort*. É importante notar que o fenômeno Atualização Perdida é prevenido pelo isolamento Cursor Stability somente quando se utiliza cursores. Em alguns gerenciadores o isolamento Read Committed é, de fato, uma implementação do isolamento Cursor Stability (BERENSON et al., 1995).

Snapshot é um caso especial pelo fato de ser o único nível de isolamento que requer um suporte transacional capaz de manter múltiplas versões dos elementos na base de dados, o que o habilita a oferecer alta concorrência em um ambiente próximo a serializável (YABANDEH; FERRO, 2012; FEKETE et al., 2005; BERENSON et al., 1995).

2 Snapshot

Neste capítulo o isolamento SNAPSHOT é apresentado formalmente segundo as especificações ANSI revisadas e as especificações portáveis. SNAPSHOT também é comparado aos níveis de isolamento vistos no capítulo anterior considerando-se os fenômenos que podem ou não se manifestar e as anomalias decorrentes de seu uso. Em seguida são descritas algumas das principais técnicas encontradas na literatura para se obter garantia de serialização com SNAPSHOT. Basicamente as técnicas se enquadram em duas categorias:

1. Serialização com SNAPSHOT — O isolamento SNAPSHOT padrão é utilizado mas requer uma alteração estratégica no código da aplicação e na base de dados ou na inclusão de um componente de software externo ao SGBD capaz de gerenciar e impedir as anomalias que, de outra forma, ocorreriam sob SNAPSHOT.

2. Serialização com algoritmos fundamentados em SNAPSHOT — Novos níveis de isolamento são propostos com base nos fundamentos do isolamento SNAPSHOT, sendo que as abordagens empregadas possuem características peculiares quanto a sobrecarga imposta para se obter garantia de serialização.

2.1 Características do isolamento Snapshot

SNAPSHOT é um nível de isolamento originalmente não previsto pelas especificações ANSI, 1992 e que exige um suporte transacional capaz de manter múltiplas versões dos elementos na base de dados, conhecido como MVCC — Multiversion Concurrency Control[1] (YABANDEH; FERRO, 2012; BERENSON et al., 1995). Fundamentalmente, o nível de isolamento SNAPSHOT adiciona apenas a detecção de conflitos do tipo write-write a esses sistemas.

Em sistemas onde o isolamento SNAPSHOT é oferecido, as transações recebem identificadores de tempo no instante em que iniciam e encerram, conhecidos como *start-timestamp* e *commit-timestamp*. Em geral esses identificadores são atribuídos por um serviço centralizado de modo a garantir a ordem das transações e a unicidade de cada instante (YABANDEH; FERRO, 2012). A notação[2] a seguir é utilizada para se descrever o mecanismo de gerenciamento do isolamento SNAPSHOT:

[1] MVCC: Controle de Concorrência Multiversão
[2] Esta é uma adaptação da notação utilizada por YABANDEH; FERRO, 2012.

- t_{s_i} — Indica o *start-timestamp* da transação T_i.
- t_{c_i} — Indica o *commit-timestamp* da transação T_i.
- $[t_{s_i}, t_{c_i}]$ — Intervalo de tempo no qual a transação T_i iniciou (t_{s_i}) e encerrou (t_{c_i}), ou seja, seu tempo de vida.

Nas seções seguintes o isolamento SNAPSHOT é descrito segundo as especificações ANSI revisadas (BERENSON et al., 1995) e as especificações portáveis (ADYA, 1999).

2.1.1 Snapshot segundo as especificações ANSI revisadas

Uma transação T_i sob o nível de isolamento SNAPSHOT tem acesso somente aos dados em estado consistente na base de dados no momento de seu início, ou seja, somente aqueles cujas versões possuem instante de *commit* δ menor que o instante de início da transação T_i ($\delta < t_{s_i}$). Esse instantâneo (*snapshot*) da base de dados é totalmente imune as operações disparadas por transações concorrentes e comporta-se como se a mesma estivesse sob a ação de uma única transação. Para a transação SNAPSHOT, os elementos que figuram na base de dados são aqueles cujas versões foram instaladas antes de seu início e aqueles que a própria transação escrever (YABANDEH; FERRO, 2012; FEKETE et al., 2005; BERENSON et al., 1995).

Duas transações T_i e T_j estarão em conflito se apresentarem sobreposição espacial e temporal. A sobreposição espacial caracteriza-se quando as transações escrevem um mesmo elemento, ou seja, apresentam um conflito do tipo write-write. Por sua vez, a sobreposição temporal ocorre se T_i e T_j forem concorrentes, ou seja, $t_{s_i} < t_{c_j}$ e $t_{s_j} < t_{c_i}$ (YABANDEH; FERRO, 2012). Desta forma, uma transação T_i pode encerrar somente se nenhuma outra transação com instante de *commit* δ no intervalo de execução de T_i ($t_{s_i} < \delta < t_{c_i}$) escreveu um elemento também escrito por T_i. Essa estratégia, *First-Committer-Wins* (Primeira a encerrar vence), impede a manifestação dos fenômenos Escrita Suja e Atualização Perdida (BERENSON et al., 1995). Alternativamente, o sistema transacional pode adotar a estratégia *First-Updater-Wins* (Primeira a atualizar vence). Neste caso, se T_i e T_j são transações concorrentes e T_i escreve um elemento x, então T_i adquiri um bloqueio exclusivo sobre este. Se posteriormente a transação T_j tenta escrever o mesmo elemento enquanto a transação T_i ainda está em andamento, T_j aborta imediatamente ou é colocada em estado de espera até que o bloqueio sobre x seja liberado. Caso T_i finalize com um *commit*, então T_j deverá abortar; T_j terá permissão para prosseguir apenas se T_i liberar o bloqueio fazendo um *abort*. Se por ventura T_j tentar escrever o elemento x após o encerramento de T_i, T_j deverá abortar imediatamente mesmo não havendo mais um bloqueio sobre este (FEKETE et al., 2005).

Na Figura 8, T_2 e T_3 apresentam sobreposição espacial e temporal. A sobreposição espacial ocorre devido às duas transações escreverem o mesmo elemento x. A sobreposição temporal ocorre porque as transações são concorrentes. Neste caso, independentemente do uso da estratégia *First-Committer-Wins* ou *First-Updater-Wins*, uma das duas transações deve abortar.

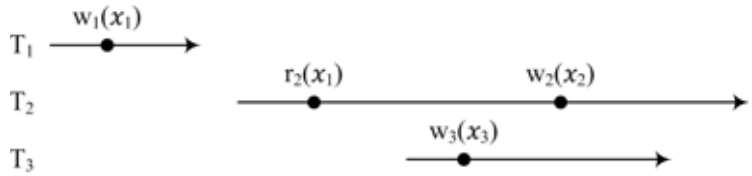

Figura 8 – Sobreposição espacial e temporal envolvendo as transações T_2 e T_3.

Graças ao mecanismo de versionamento dos dados, os fenômenos Leitura Suja, Leitura não-Repetível, Fantasma e Leitura Oblíqua são naturalmente prevenidos pelo isolamento SNAPSHOT, visto que as atualizações provocadas por transações concorrentes de fato geram novos elementos na base de dados e estes não são visíveis a uma transação SNAPSHOT em andamento (YABANDEH; FERRO, 2012; BERENSON et al., 1995).

Contudo, o simples monitoramento de conflitos do tipo write-write não é suficiente para se garantir serialização a transações que executam sob o isolamento SNAPSHOT (YABANDEH; FERRO, 2012). Como exemplo, considere dois elementos distintos x e y na base de dados representando duas contas bancárias com valores $x = 50$ e $y = 50$, e sobre os quais existe a restrição $x + y > 0$, ou seja, uma das contas pode ficar com saldo negativo desde que a outra conta tenha saldo suficiente para cobrir o débito. Considere agora duas transações T_1 e T_2 com as seguintes características:

– T_1 faz uma retirada de \$60 da conta x.
– T_2 faz uma retirada de \$60 da conta y.

Isoladamente apenas uma das transações seria finalizada com sucesso, o que levaria o saldo das contas a ser $x = -10$ e $y = 50$, caso T_1 executasse primeiro, ou $x = 50$ e $y = -10$, caso T_2 executasse primeiro. Em quaisquer dos casos a restrição $x + y > 0$ seria mantida e portanto a base de dados permaneceria em um estado consistente. Considere, entretanto, o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_1(x_0,50)$ $r_2(y_0,50)$ $r_1(y_0,50)$ $w_1(x_1,-10)$ c_1 $r_2(x_0,50)$ $w_2(y_2,-10)$ c_2

Inicialmente a transação T_1 verifica o saldo da conta x e a transação T_2 verifica o saldo da conta y. T_1 identifica que a conta x não possui saldo suficiente para cobrir a retirada e então inicia a verificação do saldo da conta y. A transação T_1 confirma a possibilidade de retirada e efetua o saque de \$60 da conta x, encerrando sua execução logo em seguida. No momento em que T_1 finaliza com um *commit*, a base de dados ainda

encontra-se em um estado consistente: $x + y > 0$. Paralelamente, T_2 identifica que a conta y não possui saldo suficiente para cobrir a retirada e então inicia a verificação do saldo da conta x. Note neste ponto que mesmo T_1 já tendo encerrado, T_2 tem acesso somente a uma versão anterior ao dado atualizado em x por T_1. A transação T_2 confirma, erroneamente, a possibilidade de retirada e efetua o saque de \$60 da conta y, encerrando sua execução logo em seguida. Após o encerramento de T_2 o saldo das duas contas passa a ser $x = -10$ e $y = -10$, violando portanto a restrição $x + y > 0$ e levando a base de dados a um estado inconsistente. O histórico H é, portanto, não serializável.

No exemplo dado, nota-se que transações sob o nível de isolamento SNAPSHOT não são imunes ao fenômeno Escrita Oblíqua, o qual ocorreu por não ter havido sobreposição espacial: T_1 escreveu x, porém T_2 escreveu y. Outra anomalia decorrente do isolamento SNAPSHOT, identificada por FEKETE; O'NEIL; O'NEIL, 2004 e denominada "Anomalia de transação somente leitura", ocorre quando uma transação somente leitura participa concorrentemente com duas transações que atualizam, cada uma, elementos distintos na base de dados. Tal anomalia, entretanto, também é consequência direta do fenômeno Escrita Oblíqua em virtude dos elementos alvos de atualização estarem sob uma restrição. SNAPSHOT é situado na Tabela 6 e comparado com as especificações ANSI revisadas, onde os conceitos com fundo cinza são originalmente reconhecidos pela norma ANSI (ANSI, 1992) e os demais são oriundos da revisão proposta (BERENSON et al., 1995) e de um comparativo com o trabalho "Graus de Consistência" (GRAY et al., 1976).

Tabela 6 – SNAPSHOT, níveis de isolamento ANSI e fenômenos possíveis.

		LS	LnR	F	ES	AP	LO	EO
Grau 0		✓	✓	✓	✓	✓	✓	✓
Grau 1	Read Uncommitted	✓	✓	✓		✓	✓	✓
Grau 2	Read Committed		✓	✓		✓	✓	✓
Grau 3	Repeatable Read			✓				
	Snapshot							✓
	Serializable							

LS = Leitura Suja; LnR = Leitura não-Repetível; F = Fantasma; ES = Escrita Suja;
AP = Atualização Perdida; LO = Leitura Oblíqua; EO = Escrita Oblíqua

2.1.2 Snapshot segundo as especificações portáveis

Uma definição mais formal para o isolamento SNAPSHOT é introduzida por ADYA, 1999 — Em uma transação T_i sob o nível de isolamento SNAPSHOT, as operações de leitura são processadas considerando-se o instante de seu início (t_{s_i}), ou seja, se $r_i(x_j)$ ocorre, então:

1. T_i inicia após o encerramento da transação responsável por escrever x_j ($t_{c_j} < t_{s_i}$).

2. Se uma versão x_k existe, além de x_j, então T_i inicia antes do encerramento de T_k ($t_{s_i} < t_{c_k}$) ou T_i inicia após ($t_{c_k} < t_{s_i}$) mas a versão x_k é anterior a x_j, ou seja, x_j foi instalado na base de dados após a instalação de x_k.

Além disso, se T_i e T_j são transações concorrentes, então ambas não podem modificar um mesmo elemento, ou seja, as operações de escrita $w_i(x_i)$ e $w_j(x_j)$ podem ocorrer somente se $t_{c_i} < t_{s_j}$ ou $t_{c_j} < t_{s_i}$. Da mesma forma que antes, tal propriedade é obtida utilizando-se a estratégia *First-Committer-Wins* ou *First-Updater-Wins*.

A especificação portável para o isolamento SNAPSHOT considera um grafo de serialização direta onde um novo tipo de dependência é traçado: dependência de início. Uma transação T_j possui uma dependência de início em relação a uma transação T_i ($T_i \xrightarrow{s} T_j$) se $t_{c_i} < t_{s_j}$, ou seja, T_j inicia após o encerramento de T_i. Desta forma, conclui-se que uma dependência de início sempre existe entre transações não concorrentes, porém a mesma será útil apenas no tratamento de transações SNAPSHOT.

Além da dependência de início, dois novos fenômenos são identificados e utilizados na especificação portável do isolamento SNAPSHOT:

- Interferência
- Efeito Perdido

O fenômeno Interferência ocorre em um histórico H se o grafo de serialização direta deste histórico tiver uma transação T_2 com uma dependência de leitura ou escrita em relação a uma transação T_1 e a mesma transação T_2 não possuir uma dependência de início em relação a transação T_1. Intuitivamente, a ocorrência do fenômeno Interferência indica que transações concorrentes escreveram um mesmo elemento ou as ações tomadas por uma interferem na outra. Este fenômeno é representado na Figura 9.

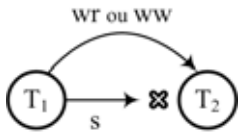

Figura 9 – Manifestação do fenômeno Interferência.

O fenômeno Efeito Perdido ocorre em um histórico H se o grafo de serialização direta deste histórico tiver um ciclo direcionado com exatamente uma dependência de início e uma antidependência. Intuitivamente, a ocorrência do fenômeno Efeito Perdido indica que uma transação T_2 não obteve acesso a todos os efeitos produzidos por uma transação T_1 que encerrou antes do início de T_2, estabelecendo a antidependência como se T_1 tivesse escrito um elemento após sua leitura por T_2. Este fenômeno é representado na Figura 10.

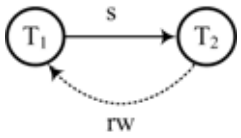

Figura 10 – Manifestação do fenômeno Efeito Perdido.

Snapshot é, portanto, especificado como sendo o nível de isolamento PL-SI (*Portable-Level Snapshot Isolation*) onde apenas os fenômenos Ciclo com Antidependência de Item e Ciclo com Antidependência podem se manifestar. Ao contrário da especificação ANSI revisada, uma implementação alternativa de Snapshot pode possibilitar a leitura de elementos escritos por transações concorrentes ainda em andamento.

O isolamento Snapshot é situado na Tabela 7 e comparado com os fenômenos característicos de outros níveis portáveis.

Tabela 7 – Snapshot, níveis de isolamento portáveis e fenômenos possíveis.

			CE	LA	LI	FIC	CADi	CAD	I	EP
Grau 0			✓	✓	✓	✓	✓	✓	✓	
Grau 1	Read Uncommitted	PL-1		✓	✓	✓	✓	✓	✓	
Grau 2	Read Committed	PL-2					✓	✓	✓	
Grau 3	Repeatable Read	PL-2.99						✓	✓	
	Snapshot	PL-SI					✓	✓		
	Serializable	PL-3							PL	

CE = Ciclo de Escrita; LA = Leitura Abortada; LI = Leitura Intermediária;
FIC = Fluxo de Informação Circular; CADi = Ciclo com Antidependência de Item;
CAD = Ciclo com Antidependência; I = Interferência; EP = Efeito Perdido

✓ = O fenômeno ocorre em todos os níveis de isolamento
PL = O fenômeno ocorre apenas no nível de isolamento portável em questão

Nota-se que o fenômeno Interferência não implica, necessariamente, em um histórico não serializável, razão pela qual o mesmo pode se manifestar em transações sob o isolamento PL-3. Entretanto, a ausência deste no isolamento Snapshot condiciona o mesmo a impedir alguns históricos serializáveis. Com relação ao fenômeno Efeito Perdido, observa-se que este não deve se manifestar em nenhum dos níveis de isolamento, ou seja, os efeitos provocados por uma transação que encerrou devem ser visíveis a todas as transações que iniciam na sequência, não importa qual seja o nível de isolamento.

2.1.3 Considerações

Pelo fato de lidar com uma visão isolada da base de dados, uma transação sob o isolamento Snapshot nunca bloqueia e nunca é bloqueada quando solicita uma operação de leitura, evidentemente considerando-se que tal visão possa ser mantida pelo menos pelo tempo de vida da própria transação. Além disso, como o monitoramento ocorre apenas para conflitos do tipo write-write, transações somente leitura executam

sem qualquer sobrecarga de bloqueio mesmo em sistemas onde o suporte transacional faz uso desses (YABANDEH; FERRO, 2012; BERENSON et al., 1995).

Em uma implementação livre de bloqueios, a fase de encerramento da transação deve receber um conjunto com os identificadores de todos os elementos que escreveu. Neste caso o servidor deve ter informações suficientes para que seja capaz de verificar sobreposições espaciais e temporais (YABANDEH; FERRO, 2012).

Em sistemas que oferecem o recurso de isolamento SNAPSHOT é possível também iniciar uma transação com um *timestamp* propositalmente antigo de modo a permitir uma "viagem no tempo", tomando-se uma perspectiva histórica da base de dados (BERENSON et al., 1995).

A revisão proposta para a norma ANSI (BERENSON et al., 1995), embora correta, define o isolamento SNAPSHOT de forma pouco formal. ADYA, 1999 define o isolamento SNAPSHOT formalmente segundo restrições sobre o histórico e determinados padrões em um grafo de serialização resultante deste. Assim como nas especificações ANSI revisadas, SNAPSHOT é fundamentado na identificação de conflitos do tipo write-write, contudo as especificações portáveis permitem uma implementação alternativa onde uma transação SNAPSHOT pode ler elementos escritos por transações ainda em andamento.

As especificações propostas estão resumidas na Tabela 8 considerando-se todos os níveis de isolamento e fenômenos vistos, onde a possibilidade de ocorrência ou não destes é discriminada de duas formas: o fenômeno ocorre em quaisquer dos níveis propostos ou apenas no isolamento portável em questão. Na tabela, os conceitos com fundo cinza são originalmente reconhecidos pela norma ANSI (ANSI, 1992) e os demais são oriundos da revisão proposta (BERENSON et al., 1995) e de um comparativo com os trabalhos "Graus de Consistência" (GRAY et al., 1976) e "Especificações Portáveis" (ADYA; LISKOV; O'NEIL, 2000; ADYA, 1999).

2.2 Tornando Snapshot serializável

Independentemente do nível de isolamento em uso, qualquer tipo de anomalia pode ser detectada por meio de grafos de serialização de testes. Basicamente, a ideia consiste em receber a requisição de uma operação e traçar uma possível dependência em um grafo de serialização antes mesmo de executar a ação solicitada. Se um ciclo for encontrado, então uma das transações participantes deste deve abortar. Tal estratégia requer a existência de uma estrutura de dados que represente o grafo, onde cada nó corresponde a uma transação que encerrou ou que ainda está em andamento. Arestas são adicionadas ao grafo à medida que as dependências são identificadas e um conjunto dos elementos lidos e escritos por cada transação também é mantido. Embora a estratégia possibilite a construção de um sistema transacional que impeça apenas execuções não

Tabela 8 – Todos os níveis de isolamento e fenômenos possíveis.

			Fenômenos identificados pelas especificações ANSI revisadas							Fenômenos identificados pelas especificações portáveis							
			LS	LnR	F	ES	AP	LO	EO	CE	LA	LI	FIC	CADi	CAD	I	EP
Grau 0			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Grau 1	Read Uncommitted	PL-1	✓	✓	✓	PL	✓	✓	✓		✓	✓	✓	✓	✓	✓	
Grau 2	Read Committed	PL-2	PL	✓	✓	PL	✓	✓	✓					✓	✓	✓	
Grau 3	Repeatable Read	PL-2.99	PL		✓	PL									✓	✓	
	Snapshot	PL-SI	PL						✓					✓	✓		
	Serializable	PL-3	PL			PL										PL	

LS = Leitura Suja; LnR = Leitura não-Repetível; F = Fantasma; ES = Escrita Suja;
AP = Atualização Perdida; LO = Leitura Oblíqua; EO = Escrita Oblíqua

CE = Ciclo de Escrita; LA = Leitura Abortada; LI = Leitura Intermediária; FIC = Fluxo de Informação Circular;
CADi = Ciclo com Antidependência de Item; CAD = Ciclo com Antidependência; I = Interferência; EP = Efeito Perdido

✓ = O fenômeno ocorre em todos os níveis de isolamento
PL = O fenômeno ocorre apenas no nível de isolamento portável em questão

serializáveis, a mesma não é prática do ponto de vista de implementação por possuir alta sobrecarga de gerenciamento (YABANDEH; FERRO, 2012; CAHILL, 2009).

Nas seções seguintes são descritas algumas técnicas para se obter garantia de serialização com base no isolamento SNAPSHOT e com uma sobrecarga de gerenciamento aceitável. As duas primeiras técnicas apresentadas utilizam o próprio isolamento SNAPSHOT e, portanto, algum tipo de estratégia adicional deve ser utilizada para se obter garantia de consistência. Por sua vez, as três últimas técnicas representam propostas de protocolos transacionais fundamentados no isolamento SNAPSHOT mas capazes de impedir a ocorrência do fenômeno Escrita Oblíqua, garantindo portanto apenas históricos serializáveis.

2.2.1 Análise estática das tarefas transacionais

Uma maior eficiência no processamento ou uma garantia maior de concorrência é a clássica justificativa para utilização de um nível de isolamento mais fraco. Se uma tarefa transacional não produz dados inconsistentes sob um isolamento que não garante serialização, ou se pelo menos a inconsistência obtida não é crítica para a aplicação, então a transação pode perfeitamente executar sob esse nível de isolamento. FEKETE et al., 2005 apresentam um conjunto de ferramentas teóricas que auxiliam na obtenção de garantia de serialização a transações que executam sob o isolamento SNAPSHOT. A técnica utilizada consiste em inicialmente fazer uma análise estática das transações de uma aplicação de modo a se identificar cada possível execução intercalada não serializável. Basicamente a análise é fundamentada na possibilidade de ocorrência de ciclos em um grafo de serialização direta (ADYA, 1999) resultante de um histórico em potencial envolvendo todas as transações da aplicação.

Enquanto ADYA, 1999 caracteriza que toda transação SNAPSHOT não serializável

participa de um ciclo com pelo menos uma antidependência, FEKETE et al., 2005 demonstram que o ciclo, de fato, é constituído por uma estrutura de risco contendo duas antidependências consecutivas envolvendo transações concorrentes, conforme observa-se na Figura 11. Tal estrutura está, portanto, presente em toda execução não serializável de transações SNAPSHOT e, uma vez identificada, duas estratégias podem ser utilizadas para se obter garantia de serialização:

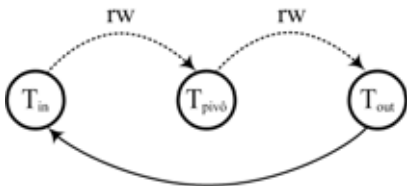

Figura 11 – Estrutura de risco envolvendo transações SNAPSHOT, onde T_{in} e T_{out} podem ser a mesma transação. Cada uma das antidependências delimita uma região de vulnerabilidade. T_{out} é a primeira transação na cadeia a encerrar.

1. Materializar restrições impostas sobre dois ou mais elementos na base de dados em um elemento distinto de modo que sua atualização seja necessária e o protocolo de SNAPSHOT possa bloquear ações conflitantes.

2. Promover elementos que são apenas lidos por uma transação a sofrerem também a ação de uma escrita de maneira a impedir que o fenômeno Escrita Oblíqua se manifeste quando esses elementos participam de uma restrição.

Embora a técnica não exija nenhum tipo de alteração no núcleo do SGBD, a possibilidade de ocorrência de uma estrutura de risco deve levar a uma modificação na lógica da aplicação de modo que o protocolo de SNAPSHOT possa identificar o conflito e a serialização seja garantida. Desta forma, a consistência é obtida transferindo-se parte da responsabilidade em lidar com problemas de concorrência do SGBD para a aplicação.

A teoria apresentada por FEKETE et al., 2005 é expandida por JORWEKAR et al., 2007 de modo que a detecção e correção de anomalias seja feita de forma prática. Assim como na estratégia anterior, um conjunto de tarefas transacionais é mecanicamente identificado e modificado de modo que nenhuma anomalia decorrente do isolamento SNAPSHOT possa ocorrer. Entretanto, técnicas são apresentadas para se identificar falsos positivos, ou seja, quais dessas tarefas previamente selecionadas, de fato, não geram anomalias. Além disso, a técnica proposta pode ser automatizada operando sobre as descrições de cada tarefa transacional, as quais são obtidas diretamente do código fonte da aplicação ou do rastreamento de instruções SQL.

Alternativamente, a transação pivô ($T_{pivô}$ na Figura 11) pode ser configurada para executar sob o protocolo 2PL, o que permite obter garantia de serialização sem a necessidade de se modificar as tarefas transacionais (ALOMARI et al., 2008b). Tal estratégia,

entretanto, pode resultar numa perda significativa de desempenho quando comparada à possibilidade de se aplicar a teoria proposta por FEKETE et al., 2005.

Desta forma a serialização sob o isolamento SNAPSHOT pode ser obtida, mas exige a modificação do código da aplicação de modo a introduzir atualizações artificiais ou bloqueios explícitos sobre cada uma das tarefas transacionais que são identificadas como possíveis geradoras de conflitos. Portanto, o processo de investigação de cada tarefa transacional (FEKETE et al., 2005), ainda que em sua forma automatizada (JORWEKAR et al., 2007), permite obter históricos serializáveis sob o isolamento SNAPSHOT mas apresenta as seguintes desvantagens:

- Quando a aplicação é alterada, todas as tarefas precisam novamente passar pelo processo de análise de modo a se identificar a possibilidade de ocorrência de novos conflitos.

- Transações *ad-hoc* não são suportadas em virtude dessas introduzirem, possivelmente, estruturas de risco em relação as tarefas transacionais já analisadas.

2.2.2 Gerenciador de bloqueios externo

As técnicas de materialização e promoção conduzem a operações de escrita adicionais. Quando essas operações são atribuídas a transações somente leitura, não apenas o desempenho das mesmas é afetado mas também a concorrência suportada pelo sistema (*throughput*) em virtude da possibilidade dessas transações serem abortadas caso o isolamento SNAPSHOT detecte um conflito do tipo write-write.

A técnica proposta por ALOMARI, 2008 e ALOMARI; FEKETE; ROHM, 2009 para se obter garantia de serialização consiste na introdução de um mecanismo de bloqueios fora dos domínios do sistema transacional do SGBD. Esse componente de software adicional é ligado a aplicação e denomina-se "Gerenciador de Bloqueios Externo". Basicamente, uma transação solicita um bloqueio exclusivo sobre uma entidade especial cujos parâmetros representam uma das regiões de vulnerabilidade de uma estrutura de risco identificada com base na teoria apresentada por FEKETE et al., 2005. Somente as transações envolvidas na região de vulnerabilidade escolhida é que requisitam bloqueios.

O gerenciador de bloqueios externo difere significativamente de um gerenciador de bloqueios tradicional nos seguintes aspectos:

- Transações que não estão envolvidas nas regiões de vulnerabilidade selecionadas não requisitam bloqueios.

- Somente bloqueios exclusivos são atribuídos.

- Poucas entidades de bloqueio são exigidas por uma transação independentemente da carga de acesso imposta pela mesma.
- Todos os bloqueios são adquiridos antes do início da transação e liberados logo após seu encerramento.
- As entidades de bloqueio são requisitadas em uma ordem específica, o que permite ao algoritmo impedir a ocorrência de *deadlocks*.

Diferentemente da técnica de materialização e promoção, o gerenciador de bloqueios externo oferece desempenho comparável a SNAPSHOT não importando quais tarefas transacionais foram selecionadas para modificação após a análise estática feita sobre a aplicação. Dessa forma, tem-se um *throughput* próximo a SNAPSHOT ainda que todas as transações envolvidas em estruturas de risco passem a requisitar bloqueios.

2.2.3 Serializable Snapshot

Considerando-se as desvantagens apresentadas pela técnica de investigação e adaptação de código, e na busca por um nível de isolamento que possibilite alta concorrência e ofereça garantia de serialização sem qualquer exigência de mudança no código da aplicação, um novo mecanismo de isolamento fundamentado nos princípios de SNAPSHOT é proposto: SERIALIZABLE SNAPSHOT (CAHILL; RöHM; FEKETE, 2009; CAHILL, 2009; CAHILL; RöHM; FEKETE, 2008). O algoritmo por trás deste mecanismo possui as seguintes características:

- As anomalias resultantes de uma execução intercalada de transações SNAPSHOT são detectadas e impedidas dinamicamente sem qualquer exigência de modificação no código da aplicação, garantindo-se, portanto, apenas históricos serializáveis.
- As propriedades características do isolamento SNAPSHOT são mantidas, ou seja, operações de leitura nunca são bloqueadas e nunca bloqueiam operações de escrita.
- Oferece eficiência (*throughput*) comparável a SNAPSHOT e superior a algoritmos que utilizam o protocolo Two-Phase Locking (2PL).
- Facilmente implementável em sistemas que já suportam o isolamento SNAPSHOT.

SERIALIZABLE SNAPSHOT é fundamentado na teoria apresentada por ADYA, 1999 e FEKETE et al., 2005. O algoritmo é capaz de detectar em tempo de execução determinados padrões de conflitos que ocorrem em toda execução não serializável de transações sob o isolamento SNAPSHOT. Basicamente, SERIALIZABLE SNAPSHOT monitora antidependências consecutivas envolvendo transações concorrentes e, quando tal situação é

encontrada, uma das transações é abortada. Nenhum rastreamento de ciclos em um grafo de serialização é exigido pelo algoritmo e são considerados apenas conflitos envolvendo duas transações, exigindo portanto pouca sobrecarga de gerenciamento durante o processo de detecção.

Pelo fato de não identificar se as transações envolvidas participam de um ciclo, SERIALIZABLE SNAPSHOT detecta uma execução não serializável em potencial, agindo portanto de forma conservadora pois algumas transações serializáveis também podem ser abortadas desnecessariamente (falsos positivos).

Em sua forma fundamental, para toda transação T o algoritmo mantém dois parâmetros booleanos: *in* e *out*. O primeiro indica se T possui uma antidependência em relação a uma transação concorrente, enquanto o segundo indica se uma transação concorrente possui uma antidependência em relação a T. Assim sendo, uma transação não serializável em potencial é caracterizada quando os dois parâmetros são verdadeiros. Entretanto, isso não implica necessariamente que a transação T deva ser a vítima escolhida para abortar.

SERIALIZABLE SNAPSHOT identifica a ocorrência de uma antidependência envolvendo duas transações concorrentes T_1 e T_2 em dois momentos:

1. Quanto T_2 escreve um elemento gerando uma nova versão deste e T_1, posteriormente, lê a versão anterior do elemento. No instante em que T_1 efetua a leitura, o algoritmo tem conhecimento de uma versão mais recente do elemento lido e, neste caso, marca os parâmetros booleanos T_1.out e T_2.in como verdadeiros, detectando portanto a existência de uma antidependência. Neste momento o algoritmo estabelece condições suficientes para saber apenas que T_2 possui uma antidependência e uma transação antidepende de T_1.

2. Quando T_1 lê um elemento e T_2, posteriormente, o escreve gerando uma nova versão deste. No instante em que T_2 efetua a escrita, o algoritmo tem conhecimento de toda leitura anterior feita do elemento devido ao subsistema de controle de concorrência registrar um bloqueio de leitura especial (*Snapshot Read Lock*) para cada transação SNAPSHOT que tenha lido o elemento. Neste caso, os parâmetros booleanos T.out e T_2.in são marcados como verdadeiros para toda transação T que tenha adquirido anteriormente o bloqueio de leitura. Assim como antes, o algoritmo apenas define de que lado cada uma das transações participa dos relacionamentos de antidependência.

Em sua forma otimizada, o algoritmo utiliza referências (ponteiros) a outras transações no lugar dos parâmetros booleanos. Desta forma, se uma antidependência é identificada entre duas transações T_1 e T_2 ($T_1 \xrightarrow{\text{rw}} T_2$), então os parâmetros pertinentes a cada transação são configurados do seguinte modo: T_1.out = T_2 e T_2.in = T_1. Tal

estratégia possibilita ao algoritmo identificar situações onde duas antidependências consecutivas são permitidas desde que a última transação na cadeia não seja a primeira a encerrar, reduzindo portanto o número de falsos positivos. Entretanto, se T_2 adquirir uma antidependência em relação a outra transação T_α ($T_\alpha \xrightarrow{rw} T_2$), então o parâmetro T_2.in é configurado com a própria referência a T_2 ($T_2.in = T_2$), o que permite representar a condição de existência da antidependência, porém sem a especificidade da transação origem. De maneira análoga, o mesmo procedimento é realizado para o parâmetro *out* da transação T_1 caso duas ou mais transações concorrentes possuam antidependências em relação a T_1. Em ambos os casos o algoritmo é remetido a sua forma fundamental baseada em parâmetros booleanos e, portanto, não é possível tomar decisões com base na primeira transação na cadeia a encerrar.

As informações de estado de uma transação T devem ser mantidas, mesmo após seu encerramento, pelo menos enquanto houver uma transação em execução que tenha sido concorrente a T. Essa condição é verdadeira para o algoritmo em sua forma fundamental ou otimizada.

2.2.4 Precisely Serializable Snapshot

Assim como em CAHILL, 2009, o trabalho apresentado por REVILAK, 2011 e REVILAK; O'NEIL; O'NEIL, 2011 baseia-se em FEKETE et al., 2005, porém a abordagem não é voltada a estruturas de risco, mas sim a detecção de ciclos em um grafo de dependências, onde a transação é abortada durante sua fase de encerramento caso a mesma introduza um ciclo no grafo. O algoritmo proposto, denominado Precisely Serializable Snapshot, permite uma redução de até 50% no número de transações abortadas quando comparado ao isolamento Serializable Snapshot proposto por CAHILL, 2009, resultando em um ganho aproximado de 7–19% na concorrência suportada (*throughput*).

Como a técnica baseia-se na ocorrência de ciclos de dependências, as ações tomadas por uma transação e as dependências adquiridas devem ser mantidas, mesmo após seu encerramento, pelo menos enquanto houver a possibilidade da transação se envolver em um ciclo de dependência futuro. Neste caso a transação continua a existir, mas não mais em sua forma ativa, passando a denominar-se transação zumbi. Transações zumbi são, portanto, transações que encerraram mas que ainda tem potencial para fazer parte de um ciclo de dependência futuro.

O algoritmo Precisely Serializable Snapshot é fundamentado em dois componentes centrais que atuam de forma cooperativa:

1. Tabela de bloqueios — Possui uma estrutura de dados derivada do gerenciador de bloqueios apresentado por GRAY; REUTER, 1993. Sua função é manter o registro

de todas as ações executadas pelas transações, identificando dependências entre as mesmas e informando essas dependências ao segundo componente.

2. Grafo de teste de ciclos — Responsável por garantir que uma transação possa encerrar somente se a mesma não gerar ciclos de dependências. O componente também identifica quando uma transação que já encerrou, mas cujos registros de ações continuam sendo mantidos (transação zumbi), pode finalmente deixar de existir em virtude de não haver mais a possibilidade desta de participar de um ciclo de dependência futuro.

A estrutura de dados por trás da tabela de bloqueios é exemplificada na Figura 12 para o histórico H a seguir.

H: $r_1(v)$ $r_2(x)$ $r_2(y)$ c_2 $r_3(z)$ $w_3(y)$ c_3 $r_1(x)$ $w_1(z)$ c_1

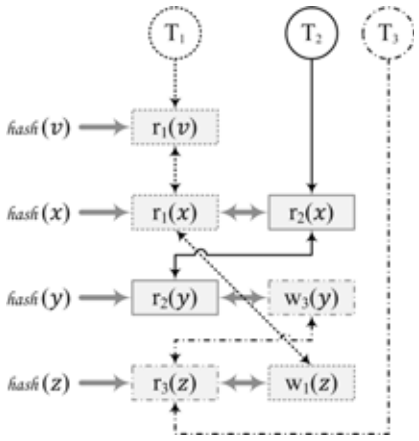

Figura 12 – Estrutura da tabela de bloqueios utilizada pelo algoritmo Precisely Serializable Snapshot.

A tabela de bloqueios é fundamentalmente constituída por um lista vertical e uma horizontal. A lista vertical mantém a relação das ações executadas por uma transação em particular, onde as operações de leitura são agrupadas no início e as operações de escrita são agrupadas no fim da lista. A lista horizontal mantém a relação das ações executadas sobre cada elemento na base de dados, onde as operações são ordenadas com base no *timestamp* de início das transações. Este é o motivo pelo qual a operação $r_1(x)$ foi inserida antes da operação $r_2(x)$ na Figura 12, mesmo $r_1(x)$ tendo sido executada após $r_2(x)$ no histórico H.

Uma dependência pode ser facilmente identificada analisando-se as operações presentes em cada lista horizontal. Conforme pode-se observar na Figura 12:

- Uma antidependência existe por conta do acesso ao elemento y envolvendo as transações T_2 e T_3 ($T_2 \xrightarrow{rw} T_3$).

- Uma antidependência existe por conta do acesso ao elemento z envolvendo as transações T_3 e T_1 ($T_3 \xrightarrow{rw} T_1$).

A tabela de bloqueios mantém registros de ações de transações ativas e transações zumbis. Como cada item na tabela deve pertencer, simultaneamente, a duas listas duplamente encadeadas, estes devem conter os seguintes dados:

- Uma referência a transação que deu origem à ação.
- O elemento alvo na base de dados.
- O tipo de operação executada (*read* ou *write*).
- As informações de encadeamento pertinentes a lista vertical.
- As informações de encadeamento pertinentes a lista horizontal.

É apenas durante a fase de encerramento de uma transação que o grafo de teste de ciclos é acionado. Basicamente o encerramento é dividido em duas etapas:

1. Identificação de dependências.
2. Checagem de existência de ciclos.

A identificação de dependências é feita varrendo-se a lista vertical vinculada a transação na tabela de bloqueios. Para cada elemento lido e escrito pela transação, a lista horizontal na qual este se encontra é consultada de modo a identificar conflitos e adicionar as respectivas dependências transacionais em um grafo de teste de ciclos. Diferentemente de um grafo de serialização direta, o tipo de dependência estabelecido entre duas transações não é discriminado, mas apenas o fato de tal dependência existir. Além disso, são considerados apenas conflitos envolvendo transações zumbi e a transação sendo encerrada. Cada nó do grafo de teste de ciclos contém, além da identificação da própria transação, dois campos principais:

- Dependentes: Lista contendo todas as transações que dependem da transação em questão.
- Número de dependências: Informa a quantidade de dependências que a transação atualmente possui.

Uma vez terminada a etapa de identificação de dependências, a checagem de existência de ciclos tem início. Nesta etapa o algoritmo consulta o grafo de teste de ciclos

de modo a identificar se a transação em encerramento introduziu ciclos no grafo. Este processo é feito utilizando-se as informações contidas nos campos descritos anteriormente: a lista de dependentes é utilizada pelo algoritmo para se testar ciclos, enquanto o número de dependências auxilia na identificação de transações que podem deixar de ser zumbis. Uma transação zumbi pode finalmente ser eliminada quando não possuir mais dependências (Número de dependências = 0) e seu *timestamp* de encerramento for menor que o *timestamp* de início da transação mais antiga dentre as que ainda estão ativas.

O teste de ciclo é realizado utilizando-se o algoritmo de "Busca em Profundidade" (CORMEN et al., 2009), o qual percorre apenas a cadeia vinculada a transação sendo encerrada em virtude do ciclo, caso exista, obrigatoriamente conter a transação. Isto é possível pois é garantido que o grafo em seu estado anterior não possui ciclos, o que permite a identificação do mesmo sem a sobrecarga associada a uma busca por toda sua estrutura.

2.2.5 Write-Snapshot

Segundo YABANDEH; FERRO, 2012, o mecanismo de identificação de conflitos do tipo write-write utilizado em SNAPSHOT, além de permitir alguns históricos não serializáveis, desnecessariamente reduz a concorrência impedindo outros que os são. Essa é uma característica pertinente ao nível de isolamento independentemente de sua implementação adotar as especificações ANSI ou portáveis. Como exemplo, considere o histórico H a seguir como resultado da execução intercalada de duas transações SNAPSHOT.

H: $r_1(x_0)$ $w_2(x_2)$ $w_1(x_1)$ c_1 c_2

As transações T_1 e T_2 apresentam sobreposição espacial e temporal, ou seja, escrevem um mesmo elemento e são concorrentes. Por essa razão, T_1 e T_2 estão em conflito segundo o isolamento SNAPSHOT e, portanto, o histórico H não é permitido. De fato, T_2 efetua uma operação de escrita no elemento x sem mesmo antes efetuar sua leitura (*blind-write*), o que torna o histórico H serializável na ordem T_1, T_2. Assim sendo, os autores questionam a eficácia do método utilizado em SNAPSHOT, destacando este não ser suficiente e também não necessário, e propõem uma abordagem baseada na detecção de conflitos do tipo read-write.

Ao contrário do mecanismo write-write, onde apenas a fase de leitura de uma transação SNAPSHOT é imune a transações concorrentes, a detecção de conflitos do tipo read-write também possibilita o isolamento da fase de escrita da transação, razão pela qual a técnica é denominada WRITE-SNAPSHOT.

Da mesma forma que no isolamento SNAPSHOT, duas transações T_i e T_j estarão em conflito sob o nível de isolamento WRITE-SNAPSHOT se houver sobreposição espacial e temporal. Entretanto, esses conceitos são ligeiramente diferentes para WRITE-

SNAPSHOT:

- Sobreposição espacial — T_i lê um elemento e T_j escreve este mesmo elemento.
- Sobreposição temporal — T_j encerra durante o tempo de vida de T_i ($t_{s_i} < t_{c_j} < t_{c_i}$).

Resumidamente, uma transação só pode encerrar se os elementos que leu não foram escritos por uma transação concorrente que já encerrou.

Diferentemente de SNAPSHOT, em uma implementação WRITE-SNAPSHOT livre de bloqueios, a fase de *commit* da transação deve carregar dois conjuntos, um contendo os identificadores dos elementos lidos e outro dos elementos escritos. Porém, da mesma forma que em SNAPSHOT, o servidor deve ser capaz de verificar sobreposições espaciais e temporais no momento em que a transação inicia o encerramento.

WRITE-SNAPSHOT permite a execução de alguns históricos que seriam proibidos sob o isolamento SNAPSHOT. Contudo, alguns históricos serializáveis em SNAPSHOT também são impedidos sob o isolamento WRITE-SNAPSHOT. Considere o histórico H a seguir.

H: $r_1(x_0)$ $r_2(y_0)$ $w_2(x_2)$ $w_1(z_1)$ c_2 c_1

A transação T_2 atualiza o elemento x com base no elemento y. T1, por sua vez, atualiza o elemento z com base em x. Ainda que o histórico H seja serializável sob o isolamento SNAPSHOT, T_1 seguido de T_2, este não é permitido sob WRITE-SNAPSHOT. Embora as vezes não necessário, o mecanismo de detecção de conflitos utilizado por WRITE-SNAPSHOT é suficiente para se garantir apenas históricos serializáveis. Entretanto, não há uma vantagem clara de WRITE-SNAPSHOT sobre SNAPSHOT quanto ao nível de concorrência suportado, pois este depende do padrão de acesso a dados imposto pelas aplicações.

2.2.6 Considerações

Praticamente todos os SGBDs de produção que oferecem serialização real o fazem via protocolo S2PL (*Strict Two-Phase Locking*), adquirindo bloqueios sobre todos os elementos lidos e escritos e mantendo esses bloqueios até o término da transação. De modo a evitar o fenômeno Fantasma, os bloqueios devem ser de predicado e comumente são implementados usando-se intervalos sobre índices (PORTS; GRITTNER, 2012).

As técnicas apresentadas neste capítulo para se obter garantia de serialização com SNAPSHOT vão desde o aproveitamento de seu próprio mecanismo de controle até a implementação de outros algoritmos cujos princípios são fundamentados em SNAPSHOT. As principais características de cada uma das estratégias propostas são discriminadas e comparadas com o isolamento SNAPSHOT na Tabela 9.

Serializable Snapshot foi implementado em Berkeley DB (CAHILL; RöHM; FEKETE, 2008) e MySQL/InnoDB (REVILAK; O'NEIL; O'NEIL, 2011; CAHILL, 2009). Uma versão estendida do algoritmo foi implementada em PostgreSQL (PORTS; GRITTNER, 2012), a qual possibilita obter um desempenho comparável a Snapshot (cerca de 7% inferior) e significativamente superior a algoritmos baseados no protocolo 2PL diante de cargas de trabalho de leitura intensiva.

Tabela 9 – Técnicas de serialização baseadas em Snapshot.

	Snapshot	Análise estática	Gerenciador de bloqueios externo	Serializable Snapshot	Precisely Serializable Snapshot	Write Snapshot
Aspectos positivos						
Oferece garantia de serialização		✓	✓	✓	✓	✓
Suporta transações *ad-hoc*	✓			✓	✓	✓
Transações somente leitura nunca são abortadas	✓	✓	✓	✓	✓	✓
Aspectos negativos						
Bloqueia algumas transações serializáveis (falsos positivos)	×	×	×	×		×
Sobrecarga imposta sobre elementos lidos (read-set)				×	×	×
Sobrecarga imposta sobre elementos escritos (write-set)	×	×	×	×	×	×
Sobrecarga associada a detecção de ciclos					×	
Requer alteração no SGBD				×	×	×
Requer alteração no código da aplicação		×	×			
Requer alteração na base de dados		×				
Mantém dados sobre transações encerradas por um tempo				×	×	
Outras características						
Estratégia de monitoramento	ww	ww	ww	rw	ciclo	rw

A implementação de Serializable Snapshot em PostgreSQL apresentou uma taxa de falha de serialização abaixo de 1%, o que talvez não justifique o uso do algoritmo Precisely Serializable Snapshot diante da sobrecarga de gerenciamento que este impõe em virtude de utilizar uma abordagem baseada na detecção de ciclos (PORTS; GRITTNER, 2012).

Por sua vez, Write-Snapshot foi implementado no HBase de modo a comparar a sobrecarga imposta pelo monitoramento de conflitos read-write, utilizada por Write-Snapshot, e a sobrecarga associada ao monitoramento de conflitos write-write feito por Snapshot. Além disso, mensurou-se o *throughput* obtido por ambos algoritmos. Os resultados mostram que Write-Snapshot e Snapshot possuem desempenho comparável, sendo que Write-Snapshot apresenta uma taxa aproximada de 2% superior em relação ao número de transações abortadas em virtude da garantia de serialização que o mesmo oferece (YABANDEH; FERRO, 2012).

2.3 Considerações finais

Conforme pode-se observar, os algoritmos de controle de concorrência não levam em consideração a computação realizada sobre os elementos na base de dados. Ao invés,

toda decisão é tomada apenas com base nos elementos que são lidos e escritos.

A vantagem de se executar uma transação sob um nível de isolamento que não garante serialização está relacionada ao fato desse nível impor uma menor sobrecarga no gerenciamento e permitir uma maior concorrência. Se o SGBD for capaz de oferecer um isolamento que proporcione apenas históricos serializáveis, mas com características de gerenciamento e concorrência comparáveis a um nível menos rígido, então não há razão para se executar uma transação sob uma menor garantia de consistência (ADYA, 1999). Por outro lado, se o mecanismo de controle de concorrência impor uma sobrecarga de gerenciamento muito alta, então este corre o risco de consumir mais recursos do que realmente salva (GRAY; REUTER, 1993).

Em sua forma original SNAPSHOT é capaz de oferecer um bom desempenho, contudo o mesmo não garante um histórico serializável e, portanto, pode levar a base de dados a um estado inconsistente. Sob o isolamento SNAPSHOT as transações frequentemente atingem um desempenho próximo do obtido por transações que executam sob READ COMMITTED, com a vantagem de que SNAPSHOT é imune a um número maior de anomalias (ALOMARI, 2008). Além disso, seu algoritmo é mais eficiente que algoritmos baseados no protocolo 2PL, especialmente em tarefas transacionais de leitura intensiva (CAHILL; RÖHM; FEKETE, 2009; CAHILL, 2009).

SNAPSHOT é o nível mais alto de consistência disponível em Oracle e até recentemente em PostgreSQL (PORTS; GRITTNER, 2012). De fato, seu algoritmo é utilizado mesmo quando SERIALIZABLE é requisitado e, portanto, as aplicações estão sujeitas a eventuais inconsistências nos dados. Microsoft SQL Server, MySQL/InnoDB e Berkeley DB suportam transações SNAPSHOT e SERIALIZABLE simultaneamente, sendo que SERIALIZABLE é implementando com base no protocolo 2PL (Two-Phase Locking). Oracle não possui tal funcionalidade e PostgreSQL suporta transações SNAPSHOT e SERIALIZABLE com base apenas em um sistema transacional MVCC (PORTS; GRITTNER, 2012; JORWEKAR et al., 2007).

Ao contrário das estratégias até então apresentadas, a técnica proposta neste trabalho oferece garantia de consistência sem a necessidade de adaptação do código da aplicação e mantendo a sobrecarga de gerenciamento apenas sobre os elementos que a transação escreve. De modo a demonstrar sua aplicabilidade, foi construído um protótipo de SGBD com características temporais e voltado ao gerenciamento de dados complexos segundo o modelo orientado a objetos. Os testes realizados sobre o protótipo comparam a eficácia e eficiência da técnica proposta frente ao isolamento SNAPSHOT em sua forma natural. Embora o protótipo tenha características voltadas ao modelo orientado a objetos, a técnica é independente de modelo de dados e pode ser aplicada a gerenciadores relacionais e até mesmo a sistemas de armazenamento na nuvem.

3 Daemons e Draco

Uma das principais virtudes do algoritmo Snapshot é o fato deste exigir apenas o monitoramento dos elementos que são escritos por uma transação. Assim sendo, transações somente leitura executam sem qualquer sobrecarga de gerenciamento. Contudo, Snapshot não é capaz de garantir apenas históricos serializáveis. Os algoritmos Serializable Snapshot, Precisely Serializable Snapshot e Write-Snapshot oferecem tal garantia, mas exigem o monitoramento dos elementos lidos e escritos. Por sua vez, as técnicas de análise estática e bloqueios externos permitem tirar proveito da eficiência de Snapshot e obter garantia de serialização transferindo-se parte da responsabilidade em lidar com problemas de concorrência do SGBD para a aplicação.

A ideia proposta neste trabalho também faz uso do isolamento Snapshot, contudo o mecanismo de garantia de serialização fica apenas sob a responsabilidade do SGBD, o que permite oferecer às aplicações um ambiente dinâmico quanto ao suporte transacional. Para atingir esse objetivo, é sugerida uma alteração não estrutural no esquema da base de dados, a qual é responsável pela definição de entidades especiais que estabelecem caminhos de dependências entre aqueles elementos que estão sob alguma restrição. Essas entidades, denominadas *daemons*[1], manifestam operações de escrita virtuais sobre os elementos de um caminho, as quais em sua essência não ocorrem fisicamente mas são suficientes para que o protocolo de Snapshot possa detectar conflitos.

Neste capítulo o conceito de *daemon* é apresentado juntamente com o protótipo de um sistema gerenciador de dados complexos que o implementa de modo a oferecer garantia de serialização a transações que executam sob o isolamento Snapshot.

3.1 Daemons

Três exemplos são descritos nesta seção para se introduzir o conceito de *daemons*. Para cada um dos exemplos são apresentados diagramas UML de classes e objetos, os quais representam, respectivamente, o esquema da base de dados e um possível estado da mesma. Os diagramas UML seguem as especificações da versão 2.0 (OMG, 2013; LARMAN, 2004) e apenas os atributos e relacionamentos relevantes para a compreensão de *daemons* são considerados.

[1] *Daemon*: processo que executa automaticamente sob a guarda do sistema (WHEELER, 2013). O termo *daemon* (demônio) vem da mitologia grega, onde os *daemons* são espíritos guardiões (TECHNOLOGICA, 2013).

Sob a ótica das especificações ANSI revisadas, transações SNAPSHOT não são imunes ao fenômeno Escrita Oblíqua. Os exemplos apresentados mostram como este fenômeno pode ser evitado sob determinadas condições estruturais da base de dados utilizando-se *daemons*. Enquanto o primeiro exemplo descreve um caso básico de ocorrência do fenômeno, o segundo ressalta um problema descrito como "Anomalia de transação somente leitura sob Snapshot" e identificado por FEKETE; O'NEIL; O'NEIL, 2004. O último exemplo apresenta outra vertente de ocorrência do fenômeno Escrita Oblíqua, considerado uma variante da manifestação do fenômeno Fantasma por BERENSON et al., 1995. Entretanto, todos são consequência direta do fenômeno Escrita Oblíqua e, portanto, identificáveis por meio de *daemons*.

3.1.1 Exemplo 1 – O fenômeno Escrita Oblíqua em sua essência

Considere a representação parcial de um esquema exibido na Figura 13, onde duas classes são definidas: Pessoa e Conta. Segundo o esquema, o atributo conta de um objeto da classe Pessoa pode conter ou não uma referência (ponteiro) a um objeto da classe Conta de modo a estabelecer um vínculo entre os mesmos, ou seja, a conta bancária da pessoa. De modo análogo, o atributo titular de um objeto da classe Conta deve estabelecer a relação inversa referenciando o objeto da classe Pessoa detentora da conta em questão. Por sua vez, o atributo conjuge de um objeto da classe Pessoa pode referenciar ou não um outro objeto, também da classe Pessoa, que possui relacionamento conjugal com a pessoa que o referencia e, quando isso ocorre, a relação inversa também deve ser estabelecida por intermédio do mesmo atributo no objeto referenciado. O diagrama também deixa explícito o fato de que objetos da classe Conta possuem o atributo saldo, o qual é responsável por armazenar o saldo atual da conta.

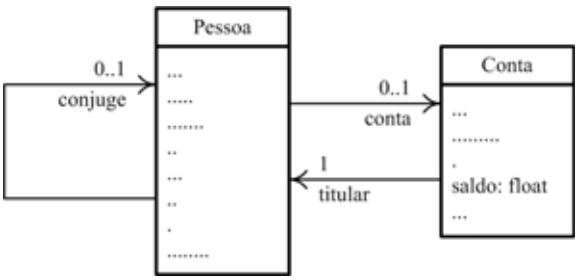

Figura 13 – Diagrama de classes UML: Representação esquemática das classes Pessoa e Conta e os relacionamentos existentes entre as mesmas.

Considere agora a situação hipotética representada na Figura 14 por meio de um diagrama de objetos UML, onde são exibidas duas instâncias da classe Pessoa (objetos P1 e P2) e duas instâncias da classe Conta (objetos C1 e C2). Os objetos P1 e P2 representam duas pessoas casadas e, portanto, o atributo conjuge de P1 referencia P2 e o atributo conjuge de P2 referencia P1. No exemplo dado os cônjuges possuem contas distintas e, assim sendo, o atributo conta de P1 referencia uma instância da classe Conta (objeto C1)

enquanto o atributo conta de P2 referencia uma outra instância da classe Conta (objeto C2). Evidentemente, o atributo titular de cada uma das contas faz referência ao objeto Pessoa titular da conta em questão.

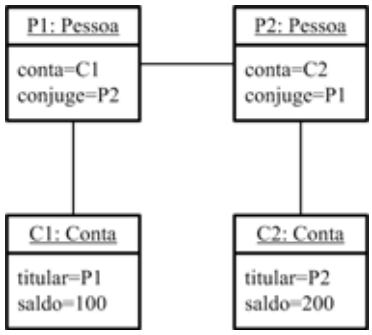

Figura 14 – Diagrama de objetos UML: Instâncias das classes Pessoa e Conta e os relacionamentos entre as mesmas.

A seguinte regra de negócio também se aplica aos objetos representados na Figura 14: "Um valor pode ser debitado da conta de uma das pessoas, mesmo não havendo saldo suficiente, desde que o saldo disponível na conta de seu cônjuge possa cobrir o débito restante."

Pode-se, inicialmente, tentar introduzir a seguinte restrição durante a definição da classe Conta:

check (saldo + titular.conjuge.conta.saldo >= 0)

Vale lembrar, entretanto, que tal abordagem não funciona diante de uma transação SNAPSHOT em virtude do valor considerado na checagem do saldo da conta do respectivo cônjuge ser baseado no estado em que se encontrava a base de dados no instante em que iniciou a transação que está solicitando o débito. Portanto, mesmo com a introdução de uma possível cláusula *check*, o fenômeno Escrita Oblíqua pode se manifestar se uma transação concorrente fizer uma alteração na conta vinculada ao cônjuge da pessoa em questão. Deve-se ainda considerar o fato de que pode não haver um cônjuge e, se houver, pode ainda não haver uma conta vinculada ao mesmo.

De modo a impedir o fenômeno Escrita Oblíqua e garantir apenas históricos serializáveis mesmo sob o isolamento SNAPSHOT, sugere-se um mecanismo semelhante a uma cláusula *check* mas que, ao invés de efetuar uma verificação, não permita que os objetos participantes de um relacionamento de restrição (C1 e C2 no exemplo dado) sofram a influência de transações concorrentes. Esse objetivo é atingido neste trabalho por meio de entidades especiais, denominadas *daemons*, as quais são responsáveis por manifestar operações de escrita virtuais não apenas sobre os objetos diretamente envolvidos em uma restrição mas sobre toda a cadeia de objetos encontrados desde a fonte da modificação até o destino, ou seja, o objeto alvo da restrição.

Na Figura 15 é exibida a linguagem que define o esquema representado na Figura 13 juntamente com a cláusula proposta para definição de *daemons*. Com exceção da cláusula *daemon*, a linguagem apresentada para o esquema segue o padrão ODMG 3.0 (BERLER et al., 2000).

```
class Pessoa
{
    .
    .
    .
    attribute Conta conta;
    attribute Pessoa conjuge;
    .
    .
    .
};

class Conta
{
    .
    .
    .
    attribute Pessoa titular;
    attribute float saldo;
    .
    .
    .
    daemon titular.conjuge.conta;
};
```

Figura 15 – Linguagem de definição de dados para as classes Pessoa e Conta segundo os padrões ODMG 3.0 e a cláusula proposta para definição de *daemons*.

A introdução da cláusula *daemon* na definição do esquema deve impor a seguinte regra ao SGBD: "Toda vez que um objeto da classe Conta é escrito, o titular da conta, seu cônjuge (se houver) e a conta deste cônjuge (se houver) também devem ser escritos."

A regra em questão seria equivalente a se aplicar a técnica de promoção sobre cada um dos objetos participantes do caminho de dependências especificado pela cláusula. Entretanto, *daemons* devem operar de forma automática e sob a guarda do SGBD. Além disso, uma operação de escrita realizada por um *daemon* deve ser virtual, ou seja, a escrita de fato não deve ocorrer fisicamente, mas apenas conceitualmente e o suficiente para que o protocolo de Snapshot possa prevenir a ocorrência do fenômeno Escrita Oblíqua e, portanto, garantir apenas históricos serializáveis.

3.1.2 Exemplo 2 – Anomalia de transação somente leitura sob Snapshot

Um caso menos comum de manifestação do fenômeno Escrita Oblíqua é exemplificado e tratado nesta seção: "Anomalia de transação somente leitura sob Snapshot", identificado por FEKETE; O'NEIL; O'NEIL, 2004.

Considere a representação parcial de um esquema exibido na Figura 16, onde duas classes são definidas: Pessoa e Conta. Segundo o esquema, o atributo conta_corrente de um objeto da classe Pessoa pode conter ou não uma referência a um objeto da classe Conta de modo a estabelecer um vínculo entre os mesmos, ou seja, a conta corrente da pessoa.

De modo análogo, o atributo conta_poupanca estabelece um vínculo entre a pessoa e sua conta poupança. Por sua vez, o atributo titular de um objeto da classe Conta deve estabelecer a relação inversa referenciando o objeto da classe Pessoa detentora da conta em questão. Assim como no exemplo anterior, o diagrama também deixa explícito o fato de que objetos da classe Conta possuem o atributo saldo, o qual é responsável por armazenar o saldo atual da conta.

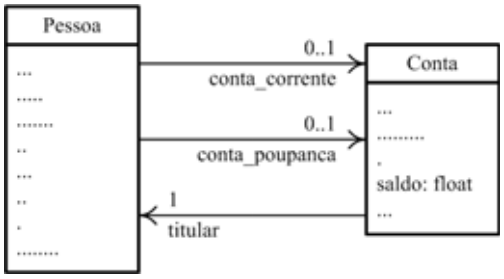

Figura 16 – Diagrama de classes UML: Representação esquemática das classes Pessoa e Conta e os relacionamentos existentes entre as mesmas considerando-se dois tipos de conta.

Considere agora a situação hipotética representada na Figura 17 por meio de um diagrama de objetos UML, onde uma única instância da classe Pessoa (objeto P) está relacionada a duas instâncias da classe Conta (objetos X e Y). No exemplo em questão o atributo conta_corrente de P referencia uma instância da classe Conta (objeto X) enquanto o atributo conta_poupanca de P referencia a outra instância da classe Conta (objeto Y). Evidentemente, o atributo titular de cada uma das contas faz referência ao objeto Pessoa titular da conta em questão.

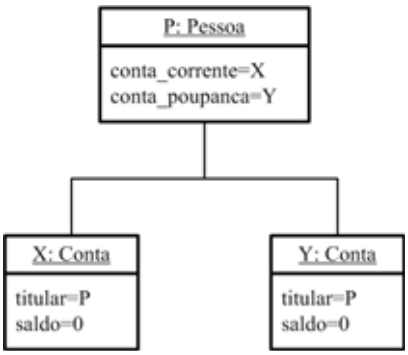

Figura 17 – Diagrama de objetos UML: Instâncias das classes Pessoa e Conta e os relacionamentos entre as mesmas considerando-se dois tipos de conta.

A seguinte regra de negócio se aplica aos objetos representados na Figura 17: “Um valor pode ser debitado de qualquer conta sem ônus, mesmo não havendo saldo suficiente, desde que o saldo disponível na outra conta possa cobrir o débito restante. O débito também é concedido caso o saldo somado das duas contas não possa cobrir a retirada, porém um ônus de $1 é adicionado à conta sendo movimentada.”

Considere agora três transações T_1, T_2 e T_3 com as seguintes características:

– T_1 faz um depósito de \$20 na conta poupança (objeto Y).
– T_2 faz uma retirada de \$10 da conta corrente (objeto X).
– T_3 é uma transação somente leitura que consulta o saldo das duas contas.

Se T_1 é serializado antes de T_2, ou seja, se o depósito é feito antes da retirada, então o resultado final deve ser X = -10 e Y = 20. Caso T_2 seja serializado antes de T_1, ou seja, a retirada é feita antes do depósito, então o resultado final será X = -11 e Y = 20. Em teoria a transação T_3 deve sempre produzir um resultado consistente sob o isolamento SNAPSHOT, principalmente considerando-se o fato da mesma ser uma transação somente leitura. Entretanto, considere o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_2(X_0, 0)$ $r_2(Y_0, 0)$ $r_1(Y_0, 0)$ $w_1(Y_1, 20)$ c_1 $r_3(X_0, 0)$ $r_3(Y_1, 20)$ c_3 $w_2(X_2, -11)$ c_2

Enquanto o resultado final obtido é X = -11 e Y = 20, a transação T_3 verifica que em um determinado instante o saldo da conta corrente é \$0 e o da conta poupança é \$20. Segundo FEKETE; O'NEIL; O'NEIL, 2004 esses dados são inconsistentes considerando-se o resultado final obtido pois a transação T_3 confirma que o depósito foi realizado antes da retirada e, neste caso, nenhum ônus deve ser aplicado e o saldo final da conta corrente deveria ser \$-10 ao invés de \$-11. Tal anomalia, entretanto, também é consequência direta do fenômeno Escrita Oblíqua. O que ocorre nesta situação é o fato das duas contas também estarem sob uma restrição, ou seja, a aplicação ou não do ônus depende do montante sendo debitado e do saldo atual das duas contas.

Na Figura 18 é exibida a linguagem de definição de dados para o esquema apresentado na Figura 16. A introdução de dois *daemons* para objetos da classe Conta garante que a anomalia em questão não se manifeste.

```
class Pessoa
{
    .
    .
    .
    attribute Conta conta_corrente;
    attribute Conta conta_poupanca;
    .
    .
    .
};

class Conta
{
    .
    .
    .
    attribute Pessoa titular;
    attribute float saldo;
    .
    .
    .
    daemon titular.conta_corrente;
    daemon titular.conta_poupanca;
};
```

Figura 18 – Linguagem de definição de dados para as classes Pessoa e Conta onde dois *daemons* são introduzidos para objetos do tipo Conta.

A introdução dos *daemons* na definição do esquema deve impor a seguinte regra ao SGBD: "Toda vez que um objeto da classe Conta é escrito, o titular da conta, sua conta corrente (se houver) e sua conta poupança (se houver) também devem ser escritos."

Neste caso, como o objeto em questão pode representar tanto uma conta corrente quanto uma conta poupança, um dos *daemons* certamente desencadeará uma operação de escrita virtual sobre o próprio objeto, cabendo ao SGBD simplesmente ignorar tal situação.

3.1.3 Exemplo 3 – Variante do fenômeno Fantasma

Finalmente, o último exemplo considera uma variante da manifestação do fenômeno Fantasma segundo BERENSON et al., 1995.

Considere a representação parcial de um esquema exibido na Figura 19, onde duas classes são definidas: Funcionario e Projeto. Segundo o esquema, o atributo projetos de um objeto da classe Funcionario representa a lista de projetos atualmente alocados ao funcionário em questão, podendo o mesmo estar à frente de nenhum projeto. Por sua vez, o atributo responsavel de um objeto da classe Projeto deve estabelecer a relação inversa referenciando o objeto da classe Funcionario responsável pelo projeto. Os objetos da classe Projeto também possuem o atributo horas, o qual especifica o número de horas diárias que o funcionário responsável pelo projeto deve dedicar.

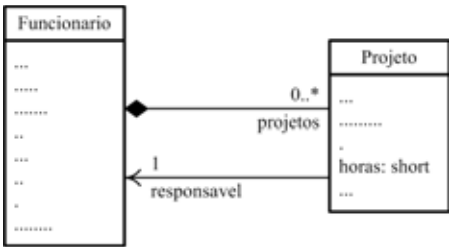

Figura 19 – Diagrama de classes UML: Representação esquemática das classes Funcionario e Projeto e os relacionamentos existentes entre as mesmas.

Considere agora a situação hipotética representada na Figura 20 por meio de um diagrama de objetos UML, onde uma única instância da classe Funcionario (objeto F) está relacionada a três instâncias da classe Projeto (objetos P1, P2 e P3). No exemplo em questão os projetos P1, P2 e P3 estão alocados ao funcionário F por meio do atributo projetos em F e da relação inversa obtida pelo atributo responsavel em cada um dos projetos.

A seguinte regra de negócio se aplica aos objetos representados na Figura 20: "Um projeto pode ser alocado a um funcionário desde que o total de horas considerando-se todos os projetos sob a responsabilidade do mesmo não ultrapasse oito."

Considere agora duas transações T_1 e T_2 com as seguintes características:

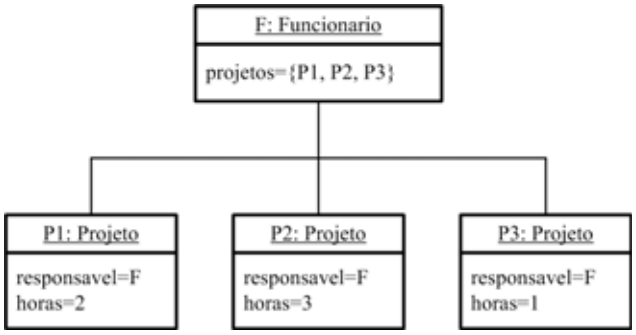

Figura 20 – Diagrama de objetos UML: Instâncias das classes Funcionario e Projeto e os relacionamentos entre as mesmas.

– T_1 verifica que o funcionário ainda possui duas horas disponíveis e aloca um projeto P4 de uma hora ao mesmo.
– T_2 faz a mesma verificação e conclui que um projeto P5 de duas horas pode ser alocado ao funcionário.

Isoladamente apenas uma das transações seria concluída com sucesso. Entretanto, considere o histórico H a seguir como resultado da execução intercalada dessas transações.

H: $r_1(P1_0, 2)$ $r_1(P2_0, 3)$ $r_1(P3_0, 1)$ $r_2(P1_0, 2)$ $r_2(P2_0, 3)$ $r_2(P3_0, 1)$
$w_1(P4_1, 1)$ c_1 $w_2(P5_2, 2)$ c_2

O isolamento SNAPSHOT não é capaz de identificar a anomalia em virtude das duas transações efetuarem a escrita sobre elementos distintos. Embora descrito por BERENSON et al., 1995 como uma variante da manifestação do fenômeno Fantasma, este também é um caso típico de ocorrência do fenômeno Escrita Oblíqua. Na Figura 21 é exibida a linguagem de definição de dados para o esquema apresentado na Figura 19. A introdução de um único *daemon* para objetos da classe Projeto garante que a anomalia em questão não se manifeste.

```
class Funcionario
{
    .
    .
    .
    attribute set<Projeto> projetos;
    .
    .
    .
};

class Projeto
{
    .
    .
    .
    attribute Funcionario responsavel;
    attribute short horas;
    .
    .
    .
    daemon responsavel;
};
```

Figura 21 – Linguagem de definição de dados para as classes Funcionario e Projeto.

A introdução do *daemon* na definição do esquema deve impor a seguinte regra ao SGBD: "Toda vez que um objeto da classe Projeto é escrito, seu responsável também é escrito."

A princípio pode-se pensar em definir um *daemon* com um caminho de dependências que atinja o objeto responsável pelo armazenamento da lista de projetos: **daemon responsavel.projetos**. Para que isso seja possível, Set<> deve ser utilizado no lugar de set<> no esquema em virtude do padrão ODMG 3.0 especificar que o primeiro representa, por si só, um objeto independente. Entretanto, a aplicação de tal estratégia não impediria que o próprio objeto responsavel sofresse uma operação de escrita virtual, visto que tais operações são disparadas para todos os objetos presentes em um caminho de dependências especificado pelo *daemon*. Portanto, a forma como foi apresentado o *daemon* na Figura 21 é suficiente para se resolver o problema.

3.1.4 Considerações

O conceito de *daemons* parte da premissa de que algum tipo de relacionamento existe entre aqueles elementos que estão sob alguma restrição e que é possível, a partir de um desses elementos, chegar ao outro por meio de um caminho de dependências. Operações de escrita virtuais ocorrem sobre todos os elementos encontrados pelo caminho mas não desencadeiam o processamento de *daemons* possivelmente ligados aos mesmos — diz-se neste caso que os *daemons* não são despertados por operações de escritas virtuais. Paradoxalmente, *daemons* agem como anjos da guarda no intuito de oferecer um ambiente transacional baseado em SNAPSHOT com garantia de serialização.

Um *daemon* é, portanto, uma entidade especial que age sob a guarda do SGBD sempre que o objeto onde o mesmo foi definido é escrito. Trata-se de uma restrição parcialmente declarada por meio de um caminho de dependências onde os elementos do caminho são virtualmente escritos, cabendo à aplicação impor a regra de negócio e ao SGBD apenas garantir a serialização utilizando SNAPSHOT.

A seguinte sintaxe é sugerida para a cláusula *daemon*: **daemon** $\mathbf{o_1}$**[.**$\mathbf{o_2}$**]**..., onde o_1, o_2, etc. representam atributos que referenciam outros objetos e que são alcançáveis a partir do objeto para o qual o *daemon* existe.

As seções seguintes descrevem um protótipo de SGBD (codinome Draco[2]) que implementa *daemons* e cuja infraestrutura possui características temporais e é voltada ao gerenciamento de dados complexos.

[2]Draco (Dragão): Criatura do filme Coração de Dragão, 1996. Draco foi o primeiro personagem totalmente computadorizado que conversava com humanos em filmes (AGUIAR, 2011). O codinome Draco para o projeto do SGBD foi escolhido arbitrariamente e não possui qualquer relação com siglas.

3.2 Draco

Nesta seção é apresentada a infraestrutura de um SGBD capaz de oferecer garantia de serialização com base em SNAPSHOT e *daemons*. A infraestrutura proposta, denominada Draco, possui características temporais e é voltada ao gerenciamento de dados complexos, os quais são constituídos por um ou mais objetos.

Draco foi construído em C++ com o propósito de demonstrar a aplicabilidade do uso de *daemons* sobre o modelo orientado a objetos, contudo a técnica de *daemons* também pode ser aplicada a gerenciadores convencionais e a sistemas de armazenamento na nuvem. Uma base de dados sob o domínio de Draco é gerenciada por intermédio de três arquivos (Figura 22):

1. Arquivo de OIDs (.oid) — Responsável por armazenar dados vinculados aos identificadores dos objetos (Object IDentifiers).
2. Arquivo de objetos (.o) — Responsável por armazenar os objetos.
3. Arquivo de controle — Responsável por manter informações que estabelecem a configuração da base de dados ou o estado atual da mesma.

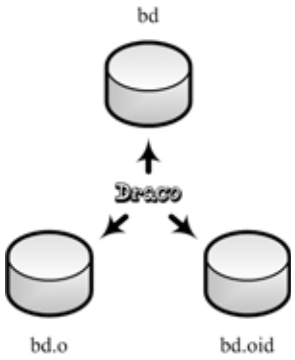

Figura 22 – Arquitetura de armazenamento de uma base de dados Draco.

A arquitetura proposta permite que os arquivos sejam armazenados em dispositivos diferentes, incluindo a própria memória principal, o que permite avaliar a sobrecarga imposta sobre cada um durante o gerenciamento da base de dados ou até mesmo reduzi-la alocando os arquivos em dispositivos mais adequados de acordo com as características e exigências de processamento, tais como unidades RAID ou SSD (Solid State Drive).

3.2.1 Arquivo de OIDs

Um OID (Object IDentifier — Identificador de Objeto) é um identificador único atribuído a cada objeto na base de dados, o qual é utilizado para se acessar objetos e estabelecer relacionamentos entre os mesmos. O arquivo de OIDs (.oid) é responsável

pelo armazenamento dos dados vinculados a esses identificadores, tais como a localização física dos objetos e a transação responsável pela última escrita dos mesmos.

Direta ou indiretamente todo OID deve levar à localização de seu objeto, seja por meio de uma abordagem física ou lógica. Enquanto a abordagem física utiliza o próprio endereço do objeto como seu OID, a abordagem lógica faz uso de estruturas de dados na memória principal de modo a mapear um valor lógico de OID para a localização física do objeto, permitindo portanto que os objetos possam ser movidos. Diversas estratégias de gerenciamento de OIDs foram propostas: endereço físico, tabelas *hash*, árvores B, tabela de indireção (ALMEIDA et al., 2012; VALêNCIO et al., 2011; FERRIZZI, 2010; VALêNCIO, 2000). A estratégia adotada neste trabalho para o gerenciamento de OIDs tira proveito das vantagens oferecidas pelas abordagens física e lógica por meio de um mapa de OIDs físico e duas estruturas de cache na memória principal.

A estrutura de armazenamento do mapa de OIDs é exibida na Figura 23. O mapa é formado por um bloco de controle principal seguido por um ou mais fragmentos. Cada fragmento, por sua vez, possui seu próprio bloco de controle e diversos blocos de OIDs (Bloco 0, Bloco 1, etc.). Todos os blocos são constituídos por duas páginas, as quais são denominadas P-0 e P-1 e agem de forma cooperativa de modo a levar a base de dados sempre de um estado consistente a outro.

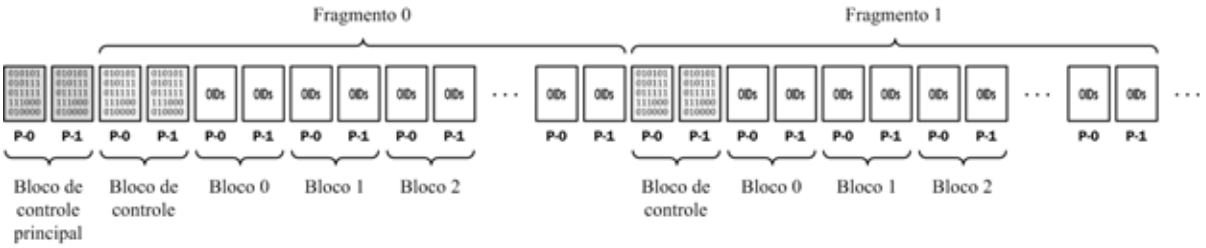

Figura 23 – Estrutura de armazenamento do mapa de OIDs (arquivo .oid).

Os blocos de controle armazenam dados binários por meio de páginas binárias. A estrutura de uma página binária é exibida na Figura 24.

Figura 24 – Página binária P-0 ou P-1 do bloco de controle principal ou do bloco de controle de um dos fragmentos do mapa de OIDs.

Uma página binária de um bloco de controle de um dos fragmentos é responsável por indicar qual das páginas, P-0 ou P-1, de cada um dos blocos de OIDs (Bloco 0, Bloco1, etc.) do respectivo fragmento contém os dados que levam ao estado atual da base de dados. O primeiro bit da página binária é responsável pelo gerenciamento do Bloco 0, o segundo bit é responsável pelo gerenciamento do Bloco 1 e assim sucessivamente. Desta forma conclui-se que o número de blocos de OIDs por fragmento é exatamente o número máximo de bits na página binária. Quando o valor de um bit é zero, diz-se que a página ativa no respectivo bloco de OIDs é a P-0, do contrário a página ativa é a P-1. Assim sendo, se o valor do i-ésimo bit na página binária do bloco de controle de um fragmento é um, então a página P-1 do Bloco i deste fragmento é a página ativa e, portanto, contém os OIDs que levam ao estado atual da base de dados. É importante notar que o próprio bloco de controle do fragmento também está sujeito à mesma regra, ou seja, em um determinado momento apenas uma das duas páginas do bloco é considerada como ativa, cabendo ao bloco de controle principal gerenciar tal informação.

Uma página binária do bloco de controle principal é responsável por indicar qual das páginas, P-0 ou P-1, do bloco de controle de cada um dos fragmentos atualmente alocados contém os dados que levam ao estado atual da base de dados. O primeiro bit da página binária é responsável pelo gerenciamento do bloco de controle do Fragmento 0, o segundo bit é responsável pelo gerenciamento do bloco de controle do Fragmento 1 e assim sucessivamente. Conclui-se portanto que o número máximo de fragmentos no mapa é exatamente o número máximo de bits na página binária. Quando o valor de um bit é zero, diz-se que a página ativa no respectivo bloco de controle é a P-0, do contrário a página ativa é a P-1. Assim sendo, se o valor do i-ésimo bit na página binária do bloco de controle principal é um, então a página P-1 do bloco de controle do Fragmento i contém a relação das páginas ativas do fragmento. De modo análogo ao bloco de controle de um fragmento, o bloco de controle principal também está sujeito a regra de ativação de páginas, ou seja, em um determinado momento apenas uma das duas páginas do bloco é considerada como ativa, desta vez cabendo ao sistema gerenciar tal informação fora dos domínios do mapa de OIDs.

Resumidamente, enquanto uma página binária do bloco de controle de um fragmento tem como função indicar qual das páginas, P-0 ou P-1, de cada um dos blocos de OIDs do fragmento é a página ativa, uma página binária do bloco de controle principal tem como função indicar qual das páginas, P-0 ou P-1, do bloco de controle de cada um dos fragmentos existentes no mapa é a página ativa. Por último, a identificação da página ativa dentro do bloco de controle principal é feita pelo sistema fora dos domínios do mapa de OIDs.

Por sua vez, os blocos de OIDs armazenam dados vinculados aos identificadores dos objetos por meio de páginas de OIDs. A estrutura de uma página de OIDs é exibida

na Figura 25.

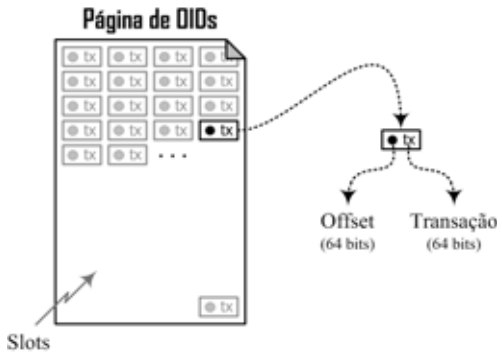

Figura 25 – Página de OIDs P-0 ou P-1 de um bloco de OIDs.

Uma página de OIDs é constituída por diversos *slots*, onde cada *slot* representa um OID e armazena as seguintes informações:

1. Offset — Indica a localização física do objeto na base de dados (arquivo .o).
2. Transação — Indica a última transação que escreveu o objeto fisicamente ou virtualmente.

Considere a situação hipotética representada na Figura 26, onde uma página binária é capaz de armazenar trinta bits e uma página de OIDs pode armazenar dez OIDs. Neste caso cada fragmento do mapa de OIDs contém trinta blocos de OIDs e pode haver no máximo trinta fragmentos. Por questões de simplificação, apenas os dados presentes nas páginas ativas são exibidos e considerou-se que o sistema atualmente mantém a página P-0 do bloco de controle principal como sendo a página ativa.

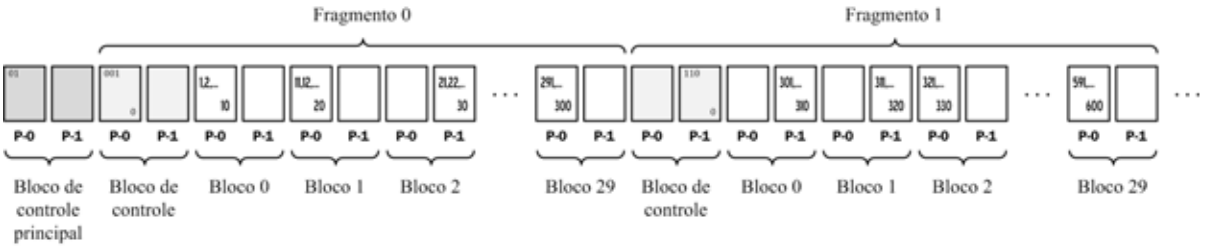

Figura 26 – Exemplo de gerenciamento envolvendo o mapa de OIDs.

De acordo com a Figura 26, a página ativa do bloco de controle principal (P-0) contém a sequência binária “01”, indicando que as páginas P-0 do bloco de controle do Fragmento 0 e P-1 do bloco de controle do Fragmento 1 são as páginas ativas. Em razão disso, a sequência binária “001. . .0”, presente na página P-0 do bloco de controle do Fragmento 0, indica que as páginas P-0, P-0, P-1 e P-0 dos respectivos blocos de OIDs 0, 1, 2 e 29 do fragmento em questão são as páginas ativas e, portanto, contém os OIDs que

conduzem ao estado atual consistente da base de dados. De modo análogo, a sequência binária "110...0", presente na página P-1 do bloco de controle do Fragmento 1, indica que as páginas P-1, P-1, P-0 e P-0 dos respectivos blocos de OIDs 0, 1, 2 e 29 do fragmento em questão são as páginas ativas e, portanto, contém os OIDs que também conduzem ao estado atual consistente da base de dados. Conforme observa-se na figura, o primeiro bloco de OIDs é responsável pelo gerenciamento dos OIDs de 1 a 10, o segundo bloco de OIDs é responsável pelo gerenciamento dos OIDs de 11 a 20 e assim sucessivamente.

Uma vez que o OID de um objeto seja conhecido, é possível obter sua localização física (*offset*) e a última transação a escrevê-lo simplesmente lendo-se a página ativa contendo o *slot* responsável pelo mesmo. Para isso basta identificar o fragmento f onde se encontra o OID, o bloco de OIDs b do fragmento responsável pelo seu armazenamento e, finalmente, a página P-0 ou P-1 a ser lida no bloco de modo a se obter os dados necessários. Essas informações são obtidas por meio das seguintes expressões, onde o tamanho da página é dado em bytes e a constante 16 representa o tamanho de cada *slot* em uma página de OIDs (2 campos de 64 bits = 16 bytes):

$$\text{nr bits por página} = \text{tamanho da página} \cdot 8,$$

$$\text{nr blocos de OIDs por fragmento} = \text{nr bits por página},$$

$$\text{nr OIDs por página} = \left\lfloor \frac{\text{tamanho da página}}{16} \right\rfloor,$$

$$\text{nr OIDs por fragmento} = \text{nr blocos de OIDs por fragmento} \cdot \text{nr OIDs por página},$$

$$f = \left\lfloor \frac{\text{oid} - 1}{\text{nr OIDs por fragmento}} \right\rfloor,$$

$$b = \left\lfloor \frac{\text{oid} - 1}{\text{nr OIDs por página}} \right\rfloor mod \text{ (nr blocos de OIDs por fragmento)}.$$

Conhecidos o fragmento f e o bloco de OIDs b do fragmento responsável pelo OID, resta identificar qual das páginas, P-0 ou P-1, no bloco deve ser lida. Para isso o sistema consulta o f-ésimo bit da página ativa do bloco de controle principal de modo a identificar qual das páginas, P-0 ou P-1, do bloco de controle do Fragmento f é a página ativa. Logo em seguida o b-ésimo bit desta página ativa é consultado de modo a se identificar qual das páginas, P-0 ou P-1, do bloco de OIDs b no respectivo fragmento deve ser lida. Considerando-se que todas as páginas dos blocos de controle são mantidas sempre na memória principal, uma única operação de leitura em disco é o suficiente para se obter a localização física do objeto e a última transação que o escreveu.

Tal estratégia, entretanto, não é eficiente pois sempre exige uma operação de leitura em disco antes do acesso ao objeto propriamente dito. Por essa razão, Draco faz uso de duas estruturas de gerenciamento de OIDs na memória principal:

1. Tabela *hash* de OIDs (cache nível 1) — Uma tabela *hash* é mantida na memória principal onde cada entrada da tabela (*bucket*) é responsável pelo armazenamento do número do OID, da localização física do objeto e da transação responsável pela última operação de escrita sobre o mesmo.

2. Cache de páginas de OIDs (cache nível 2) — As páginas de OIDs mais recentemente acessadas são mantidas na memória principal de modo que, se o OID não for encontrado na tabela *hash*, a busca pelo mesmo ocorre primeiramente no cache de páginas de OIDs e, caso este seja encontrado, seus dados são copiados para a tabela *hash*. Uma operação de busca em disco irá ocorrer apenas se o OID não estiver presente na tabela *hash* e no cache de páginas e, quando isso ocorrer, a página ativa no mapa contendo o OID requisitado é trazida para o cache de páginas e os dados do OID são copiados para a tabela *hash*.

A tabela *hash* de OIDs, exemplificada na Figura 27, armazena três campos:

1. OID — Número do OID presente no respectivo *bucket* da tabela.

2. Offset — Localização física do objeto na base de dados (arquivo .o).

3. Transação — Última transação que escreveu o objeto fisicamente ou virtualmente.

Figura 27 – Tabela *hash* de OIDs.

Desta forma, a busca pelo OID de número *oid* ocorre primeiramente aplicando-se uma função *hash* sobre o mesmo e consultando-se a tabela *hash* de OIDs: custo da operação = $O(1)$. Caso o *bucket* correspondente na tabela *hash* contenha o OID desejado, então o sistema obtém a localização física j do objeto e a transação k responsável por sua última escrita. Se o OID desejado não estiver presente na tabela *hash*, a página ativa responsável pelo mesmo é procurada no cache de páginas: custo da operação = $O(\lg n)$. Os dados do OID são então copiados para a tabela *hash* caso a página seja encontrada. Conforme mencionado, uma operação de busca em disco ocorre apenas se

o OID não estiver presente na tabela *hash* e no cache de páginas. Tal estratégia evita a ocorrência de *thrashing*, onde dois OIDs direcionados a uma mesma entrada na tabela *hash* são constantemente requisitados e, a todo momento, um cede lugar ao outro de modo a atender as requisições. Neste caso, se o cache nível 2 não estivesse presente, operações de leitura em disco seriam necessárias de modo a recuperar os OIDs a cada requisição.

As páginas não ativas no mapa de OIDs são atualizadas durante a fase de *commit* e somente ao término do processo é que as mesmas se tornam ativas. Portanto, páginas P-0 e P-1 agem cooperativamente de modo a conduzir a base de dados sempre de um estado consistente a outro. O projeto do mapa de OIDs foi inspirado no gerenciamento de páginas *shadow* e na infraestrutura de gerenciamento de registros reais, ambos adotados pelo Núcleo Gerenciador de Dados Multimídia — `NuGeM` (VALêNCIO; ALMEIDA, 2013a; VALêNCIO; ALMEIDA, 2013b; VALêNCIO, 2000). Contudo, páginas P-0 e P-1 encontram-se sempre lado a lado e, portanto, comportam-se como páginas irmãs (*siblings pages*) e não estão sujeitas a sobrecarga associada à alocação e liberação de páginas. Embora o espaço requisitado para seu gerenciamento seja maior, a estrutura proposta possibilita o agrupamento de OIDs sequenciais sem qualquer sobrecarga extra.

3.2.2 Arquivo de objetos

O arquivo de objetos (.o) é responsável pelo armazenamento dos objetos lógicos de uma aplicação em uma forma física na base de dados. Todo objeto possui um OID exclusivo, porém as diversas instâncias (versões) de um mesmo objeto compartilham de um mesmo OID, sendo que o campo Offset de cada *slot* no mapa de OIDs aponta sempre para a versão consistente mais recente dos mesmos. O arquivo de objetos é representado na Figura 28, onde três instâncias físicas de objetos distintos (o_1, o_2 e o_3) são exibidas. Com exceção da posição (*offset*) zero, os objetos podem ocupar qualquer lugar no arquivo.

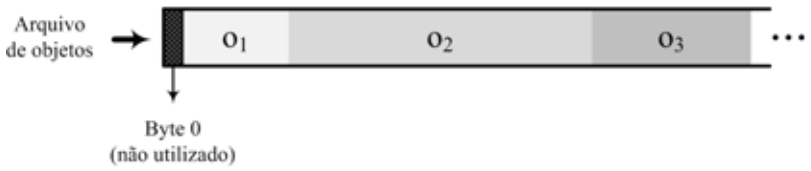

Figura 28 – Arquivo de objetos (.o).

Uma operação de inserção, atualização ou exclusão simplesmente desencadeia um processo de expansão do arquivo onde uma nova instância de um objeto é inserida, possivelmente uma instância representando a morte deste (versão *dead*). Por esse motivo, diz-se que as instâncias físicas existentes possuem caráter imutável. Na Figura 29 tem-se a representação da estrutura de armazenamento de um objeto. Um objeto em sua forma física é dividido em uma região temporal e uma área de dados. Enquanto a região temporal contém dados que auxiliam no gerenciamento da existência do objeto diante de

transações concorrentes e considerando-se um instante no tempo, a área de dados contém o objeto lógico em sua forma serializada.

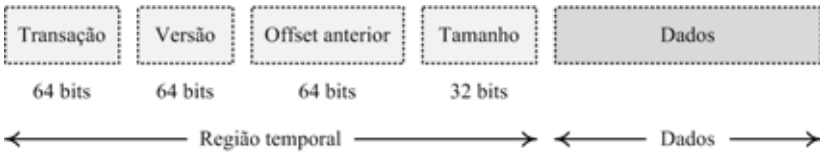

Figura 29 – Estrutura de armazenamento de um objeto.

Os campos que constituem o objeto em sua forma física são descritos a seguir:

- Transação — Indica a transação responsável pela geração da respectiva instância (versão) do objeto.
- Versão — Contém a versão do objeto.
- Offset anterior — Armazena o endereço físico onde se encontra a região temporal da versão anterior do objeto.
- Tamanho — Indica o tamanho, em bytes, do objeto lógico serializado na área de dados.
- Dados — Contém o objeto propriamente dito, porém em sua forma serializada.

Uma vez que a localização física (*offset*) do objeto seja conhecida, Draco efetua a leitura apenas de sua região temporal de modo a verificar se o objeto contido na área de dados é a versão apropriada para a transação solicitante. Esse mecanismo de verificação considera se a instância em questão é visível a transação solicitante com base em aspectos de tempo e concorrência. Caso não seja, a região temporal da versão anterior do objeto é acessada e o mesmo processo de checagem se repete até que uma instância apropriada à transação solicitante seja encontrada.

A título de exemplificação, três versões (v1, v2 e v3) de um mesmo objeto são consideradas na Figura 30, as quais foram geradas pelas transações T_1, T_2 e T_3, respectivamente. A primeira versão possui uma área de dados de 100 bytes, a segunda de 300 bytes e a terceira versão armazena um objeto lógico, em sua forma serializada, de 150 bytes. Se considerado o fato de que a terceira versão é a última instância consistente gerada para o objeto, então o campo Offset do OID deste objeto no mapa de OIDs deve apontar para o início da região temporal desta versão. Conforme observa-se na figura, a região temporal da versão 3 aponta para a região temporal da versão 2 e esta, por sua vez, aponta para a região temporal da versão 1. Finalmente, a versão 1 é a primeira versão do objeto e, portanto, possui endereço de apontamento nulo. Essa estratégia de gerenciamento é denominada neste trabalho de mecanismo de regressão temporal.

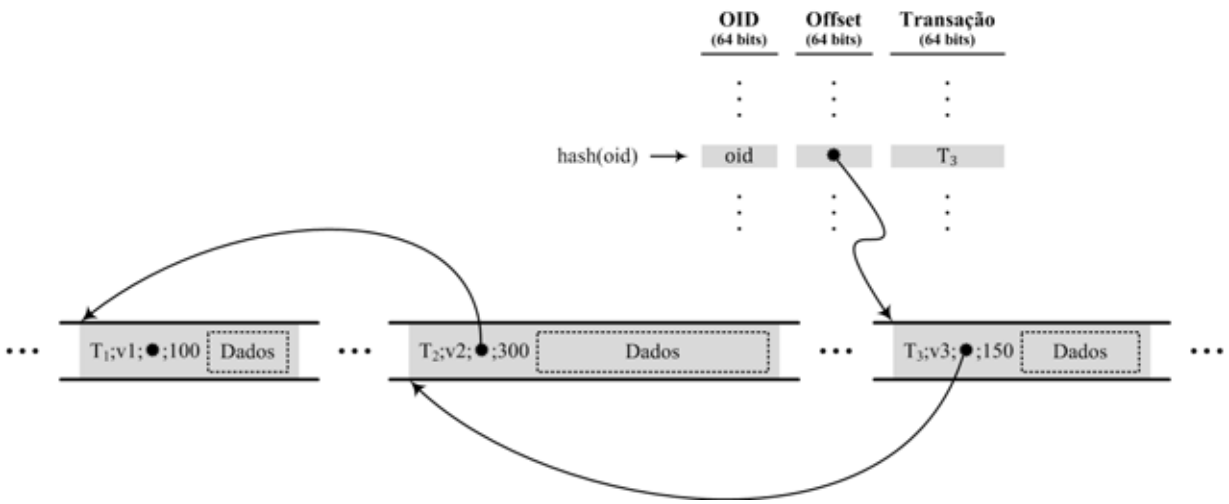

Figura 30 – Mecanismo de regressão temporal considerando-se três versões de um mesmo objeto.

A característica imutável dos objetos físicos na base de dados permite a implementação eficiente de um mecanismo de backup incremental considerando-se o ponto (*offset*) até onde o último backup foi feito e partindo-se dele. Como é garantido que antes desse ponto nenhum segmento de dados sofreu a ação de uma operação de reescrita, basta iniciar o backup a partir do mesmo e até a localização máxima atual consistente. Contudo, tal imutabilidade apresenta como lado negativo o fato do arquivo de objetos estar sujeito apenas a processos de expansão, mesmo quando objetos são excluídos da base de dados. Essa característica, entretanto, também traz como benefício o fato de se possuir uma base de dados com registros históricos completos.

3.2.3 Arquivo de controle

O arquivo de controle, representado na Figura 31, possui tamanho fixo de 49 bytes e é dividido em duas regiões: zona de *commit* e configurações. A zona de *commit* contém informações relacionadas ao processo de *commit* e é atualizada sempre que uma transação que escreveu na base de dados encerra com sucesso. Por sua vez, a região de configurações contém dados que descrevem a estrutura da base de dados ou como deve ser feito o gerenciamento da mesma.

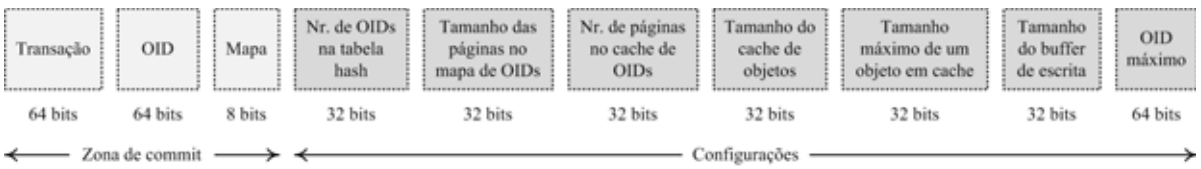

Figura 31 – Estrutura do arquivo de controle.

Os campos presentes no arquivo de controle são descritos a seguir:

- Transação — Armazena o *timestamp* da última transação que modificou a base de dados e encerrou com sucesso. Transações somente leitura ou que abortam não desencadeiam atualização na zona de *commit* em virtude de não alterarem o estado da base de dados.

- OID — Armazena o último número de identificador de objeto (Object IDentifier) que foi gerado pelo sistema.

- Mapa — Identifica a página ativa, P-0 ou P-1, no bloco de controle principal do mapa de OIDs. A página ativa do bloco de controle principal é o ponto de entrada para se obter o estado atual consistente da base de dados.

- Nr. de OIDs na tabela *hash* — Define o tamanho da tabela *hash* (número de *buckets*) que contém os OIDs recentemente acessados.

- Tamanho das páginas no mapa de OIDs — Define o tamanho, em bytes, das páginas que constituem o mapa de OIDs.

- Nr. de páginas no cache de OIDs — Estabelece o número máximo de páginas de OIDs que são mantidas na memória principal. Se um OID não é encontrado na tabela *hash* de OIDs (cache nível 1), as páginas no cache de OIDs (cache nível 2) são consultadas e, caso não seja encontrado, o mesmo é requisitado do mapa de OIDs em disco (arquivo .oid).

- Tamanho do cache de objetos — Tamanho máximo, em bytes, alocado na memória principal para o armazenamento de segmentos do arquivo de objetos contendo a região temporal e/ou a área de dados dos objetos. Draco utiliza a política de substituição LRU (Least Recently Used — Menos Recentemente Utilizado) quando um novo segmento precisa entrar no cache e não há espaço disponível. Dependendo do tamanho do segmento a entrar no cache, pode ocorrer o despejo de mais de uma unidade atualmente alocada.

- Tamanho máximo de um objeto em cache — Tamanho máximo que um objeto deve ter, em bytes, de modo que sua entrada seja permitida no cache.

- Tamanho do buffer de escrita — Estabelece o tamanho, em bytes, da região de memória responsável por receber novas instâncias dos objetos antes que as mesmas sejam escritas em disco.

- OID máximo — Armazena o maior valor de OID atualmente suportado pelo sistema. Este campo é o único da região de configurações que não é informado pelo usuário no momento de criação da base de dados e seu valor é atualizado pelo sistema à medida que novos fragmentos são alocados no mapa de OIDs.

3.2.4 Aspectos transacionais e temporais

Toda transação possui um identificador numérico único denominado *timestamp*, o qual é atribuído pelo sistema no instante em que a transação inicia e representa a ordem sequencial de criação da mesma. Uma lista global contendo os identificadores das transações atualmente em execução é mantida por Draco a todo momento e é chamada de "transações ativas". Além de seu identificador, a transação também recebe no instante de seu início uma cópia dessa lista global, a qual, dentro do contexto da transação, passa a ser denominada "transações concorrentes".

Considere a situação hipotética, exemplificada na Figura 32, envolvendo as transações T_1, T_2, T_3 e T_4.

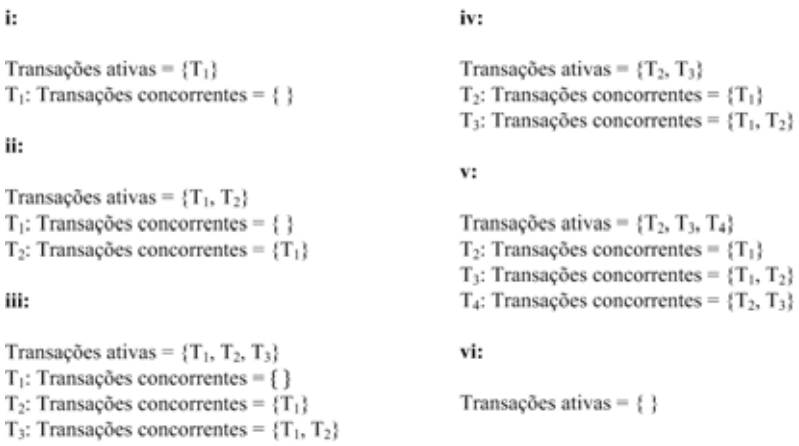

Figura 32 – Gerenciamento de transações ativas e concorrentes.

Em (i) a transação T_1 inicia e recebe o *timestamp* 1. Como nenhuma transação estava em execução no momento de seu início, a lista de transações concorrentes a T_1 fica vazia.

Em (ii) a transação T_2 inicia e recebe o *timestamp* 2. Neste caso a transação T_1 ainda está em execução e, portanto, a lista de transações concorrentes a T_2 é definida como $\{T_1\}$. Note que mesmo T_2 sendo concorrente a T_1, a lista de transações concorrentes a T_1 permanece inalterada. Draco considera qualquer transação com *timestamp* maior que o *timestamp* atribuído a T_1 como sendo concorrente a T_1, mesmo não estando em sua lista de transações concorrentes.

Em (iii) a transação T_3 inicia e recebe o *timestamp* 3. Como as transações T_1 e T_2 ainda estão em execução, a lista de transações concorrentes a T_3 é definida como $\{T_1, T_2\}$. Mais uma vez, implicitamente Draco considera que a lista de transações concorrentes a T_1 é $\{T_2, T_3\}$, por terem iniciado após T_1, e a lista de transações concorrentes a T_2 é $\{T_1, T_3\}$, T_1 por fazer parte de sua lista de transações concorrentes e T_3 por iniciar após T_2.

Em (iv) a transação T_1 encerra. O *timestamp* de T_1 não é removido de nenhuma lista de transações concorrentes em virtude das alterações feitas por T_1 não serem visíveis a transações que consideram T_1 como concorrente, o que condiz com o protocolo de SNAPSHOT.

Em (v) a transação T_4 inicia e recebe o *timestamp* 4. Como as transações T_2 e T_3 estão em execução, a lista de transações concorrentes a T_4 é definida como $\{T_2, T_3\}$. Mais uma vez, Draco considera que as transações T_1, T_3 e T_4 são concorrentes a T_2, T_1 por estar em sua lista de transações concorrentes e, T_3 e T_4 por iniciarem após T_2. De modo análogo a mesma regra se aplica às demais transações. Note entretanto que T_4 terá acesso às modificações feitas por T_1 em virtude de T_1 já ter encerrado no momento em que T_4 inicia.

Finalmente, em (vi) as transações T_2, T_3 e T_4 encerram, deixando a lista de transações ativas vazia.

Intuitivamente, no momento em que uma transação T inicia, as transações em andamento e as transações que iniciarem após T são consideradas por Draco como concorrentes a T e, portanto, suas alterações não devem ser vistas por T.

Três tipos de transações são possíveis em Draco:

1. Transação padrão — Uma transação padrão atua sempre sobre a versão mais recente dos objetos considerando-se o instante de seu início, podendo a mesma fazer alterações na base de dados.
2. Transação de leitura — Uma transação do tipo leitura possui as mesmas características que uma transação padrão. Contudo, somente operações de leitura são permitidas.
3. Transação viajante — Transações viajantes não podem fazer modificações, mas são capazes de voltar no tempo e obter acesso a um estado anterior da base de dados.

A viagem no tempo, resultado da execução de uma transação viajante, só é possível graças a um objeto especial gerenciado pelo sistema. Esse objeto, denominado objeto transacional, possui OID = 1 e sua estrutura é representada na Figura 33.

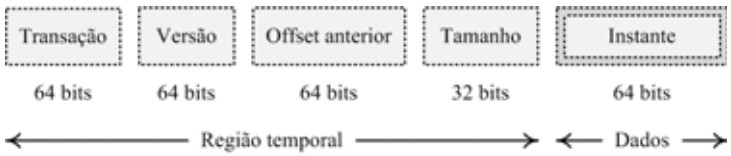

Figura 33 – Estrutura de armazenamento do objeto transacional.

Uma instância (versão) do objeto transacional é gerada para cada transação que modifica a base de dados e encerra com sucesso. A região temporal do objeto transacional

está sujeita às mesmas regras que a de um objeto comum. O campo Instante armazena quando a transação responsável por modificar a base de dados encerrou (*commit time*). Tipicamente o instante deve relatar o dia, mês, ano, hora, minuto, segundo e até milissegundo em que a base de dados migrou de um estado consistente para outro.

Por intermédio das diversas instâncias do objeto transacional é possível obter-se o histórico de alterações na base de dados e, a partir deste, iniciar uma transação viajante. Considere a situação hipotética representada na Figura 34, onde as transações T_1 a T_9 fazem modificações na base de dados e encerram com sucesso.

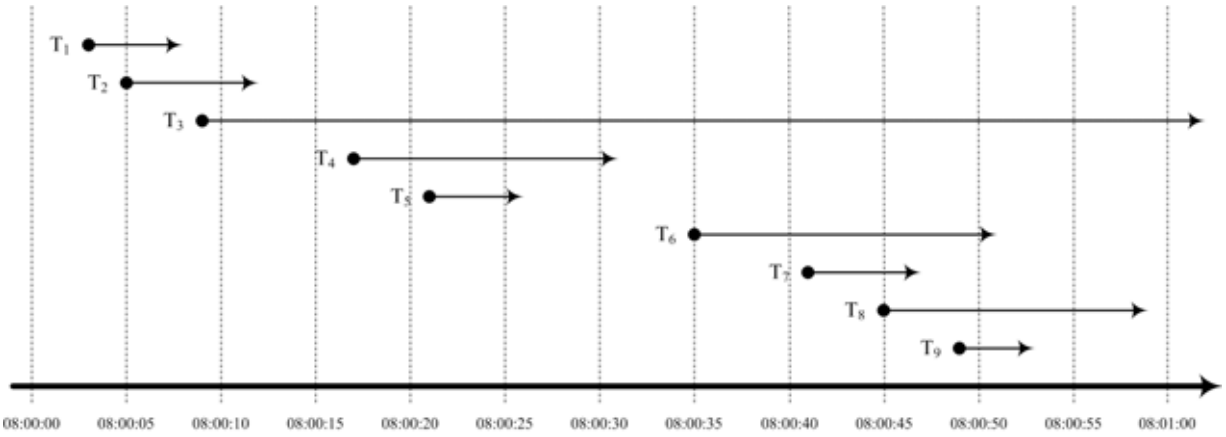

Figura 34 – Linha do tempo resultante da execução das transações T_1 a T_9.

Cada transação representada na figura possui um instante de início e término. T_1, por exemplo, inicia às 08:00:03 e encerra às 08:00:08. Conforme observa-se, o fato das transações T_1, T_2, T_3, etc. serem iniciadas uma após a outra não implica, necessariamente, que o instante de encerramento das mesmas também deva ser na ordem. A transação T_3, por exemplo, é a terceira transação a iniciar mas a última a encerrar.

Como a ordem de *commit* das transações é T_1, T_2, T_5, T_4, T_7, T_6, T_9, T_8, T_3 e as diversas instâncias do objeto transacional são geradas à medida que as transações são encerradas, tem-se como resultado as seguintes versões do objeto transacional:

– Versão = 1, Transação = T_1, Instante = 08:00:08.
– Versão = 2, Transação = T_2, Instante = 08:00:12.
– Versão = 3, Transação = T_5, Instante = 08:00:26.
– Versão = 4, Transação = T_4, Instante = 08:00:31.
– Versão = 5, Transação = T_7, Instante = 08:00:47.
– Versão = 6, Transação = T_6, Instante = 08:00:51.
– Versão = 7, Transação = T_9, Instante = 08:00:53.
– Versão = 8, Transação = T_8, Instante = 08:00:59.
– Versão = 9, Transação = T_3, Instante = 08:01:02.

Portanto, dado um instante t qualquer no tempo, é possível obter uma transação

viajante simplesmente iniciando uma transação de leitura onde sua lista de transações concorrentes é preenchida com todas as transações que encerraram após esse instante t. Pode-se concluir a partir disto que uma transação padrão é, de fato, uma transação viajante com permissão de escrita e que considera o instante no tempo como sendo o presente.

Considerando o exemplo da Figura 34, uma transação viajante atuando às 08:00:34 deve ter sua lista de transações concorrentes definida como $\{T_3, T_8, T_9, T_6, T_7\}$, a qual contém a relação das transações que encerram após o horário 08:00:34.

De modo a se obter a lista de transações concorrentes para uma transação viajante no instante t, Draco varre, a partir de sua versão mais recente, cada instância do objeto transacional até encontrar uma transação que tenha encerrado antes ou exatamente no instante t, adicionando as transações encontradas pelo caminho na lista de transações concorrentes da transação viajante.

3.2.5 Serialização

Quando uma transação gera uma nova instância de um objeto, a localização física (*offset*) dessa instância e o OID do objeto são inseridos na lista de objetos escritos da transação. Se o objeto escrito possuir um *daemon*, os OIDs dos objetos encontrados em toda sua cadeia de dependências também são inseridos na mesma lista, porém com *offset* igual a zero.

Na Figura 35 é dado um exemplo de processamento transacional envolvendo uma atualização na base de dados.

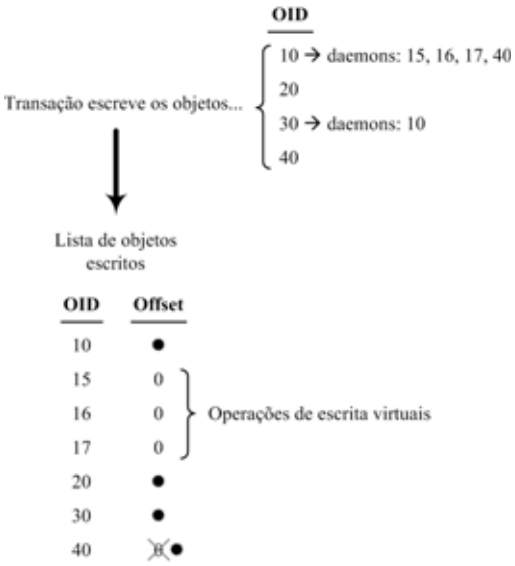

Figura 35 – Objetos escritos por uma transação e as operações de escrita virtuais decorrentes.

No momento em que a transação escreve o objeto de OID 10, Draco atribui uma

posição física para sua nova instância e a escreve na base de dados, possivelmente apenas inserindo-a na área reservada para o *buffer* de escrita. Além disso, seu OID e o novo *offset* são inseridos na lista de objetos escritos da transação. No exemplo em questão, os objetos com OIDs 15, 16, 17 e 40 fazem parte da cadeia de *daemons* do objeto de OID 10 e, neste caso, embora não sejam fisicamente escritos na base de dados, os mesmos também são inseridos na lista de objetos escritos da transação, porém com *offset* igual a zero, caracterizando uma operação de escrita virtual.

Como nenhum *daemon* está associado ao objeto de OID 20, a geração de uma nova instância do mesmo simplesmente faz com que seu OID e novo *offset* sejam incluídos na lista de objetos escritos da transação.

De modo análogo ao que ocorre para o objeto de OID 10, quando o objeto de OID 30 é escrito, seu OID e novo *offset* são inseridos na lista de objetos escritos. Entretanto, o objeto de OID 10, agora atuando como *daemon* do objeto de OID 30, não é inserido na lista de objetos escritos da transação com *offset* igual a zero em virtude do mesmo já estar presente e possuir uma posição física real. Operações de escrita virtuais não sobrepõem, portanto, operações de escrita reais.

Assim como para os anteriores, a escrita do objeto de OID 40 faz com que Draco encontre uma localização física para sua nova instância e insira mais um elemento na lista de objetos escritos. Note neste caso que, embora o objeto de OID 40 tenha sido virtualmente escrito quando da operação de escrita física do objeto de OID 10, o valor zero de seu *offset* é substituído por sua localização física real em virtude do mesmo sofrer uma operação de escrita real. Portanto, ao contrário do que ocorre com operações de escrita virtuais, operações de escrita reais sobrepõem as virtuais.

Uma transação T só pode encerrar se uma transação concorrente a T, que já encerrou, não escreveu um objeto que T também escreveu ou que esteja na cadeia de *daemons* gerada para T. Desta forma, quando a transação entra na fase de encerramento, os objetos presentes em sua lista de objetos escritos são confrontados um a um com a versão atual dos mesmos na tabela *hash* de OIDs de modo a verificar se a última transação a escrevê-los não é uma transação concorrente a T. Caso seja, um conflito espacial e temporal é identificado e, portanto, T deve abortar. Note que as operações de escrita virtuais são checadas como se os objetos ligados às mesmas tivessem sido escritos fisicamente, prevenindo portanto a ocorrência do fenômeno Escrita Oblíqua e garantindo a serialização da transação ou o cancelamento da mesma.

Finalmente, se não houver conflito espacial e temporal, a transação encerra com sucesso e o mapa de OIDs é atualizado com os dados vinculados às operações de escrita na lista de objetos escritos da transação.

3.2.6 Considerações

A infraestrutura proposta para Draco é capaz de manter a consistência da base de dados ainda que a mesma sofra a ação de falhas de hardware ou software, evidentemente considerando-se que o meio físico onde os três arquivos são hospedados permaneça íntegro. No entanto, Draco não possui mecanismos de recuperação de espaços para aqueles objetos que foram inseridos mas são considerados como inexistentes em virtude de falhas de hardware, software ou até mesmo como resultado de uma transação que abortou. As instâncias desses objetos são denominadas *embriões* pois, de fato, não chegaram a nascer e são totalmente inacessíveis.

3.3 Considerações finais

Quando os objetos na cadeia de *daemons* sofrem uma operação de escrita virtual, estes se comportam como bloqueadores e portanto são capazes de oferecer garantia de serialização com desempenho comparável a técnica utilizada pelo "Gerenciador de Bloqueios Externo", mas sem a necessidade de se modificar o código da aplicação e introduzir uma camada de software adicional entre o SGBD e a mesma. Além disso, transações *ad-hoc* passam a ser suportadas.

Embora os objetos na cadeia de *daemons* não sofram a ação de uma operação de escrita real, os mesmos devem ser lidos de modo que o OID do objeto seguinte na cadeia possa ser obtido. Entretanto, como os *daemons* devem ser definidos para objetos que se encontram sob alguma restrição, é esperado que os mesmos estejam presentes na memória cache uma vez que a aplicação deve tê-los consultado de modo a garantir a regra de negócio. Do contrário não teria sentido ter-se um ou mais *daemons* agindo sobre os objetos. Os dados levantados no capítulo anterior sobre as técnicas de serialização baseadas em Snapshot são exibidos novamente na Tabela 10, porém desta vez comparando-os com a técnica que utiliza *daemons*: **Daemon Snapshot**.

Conforme pode-se observar, exceto pelo isolamento Snapshot a garantia de serialização é obtida em todas as técnicas propostas. Devido aos mecanismos de análise estática e gerenciador de bloqueios externos exigirem a investigação de cada tarefa transacional de modo a identificar possíveis conflitos, estas são as únicas abordagens que não suportam transações *ad-hoc*. Em todos os casos as características do protocolo de Snapshot garantem que transações somente leitura não sejam abortadas.

O algoritmo Precisely Serializable Snapshot é o único capaz de permitir todos os históricos que são serializáveis, mas isso é feito ao custo do monitoramento de ciclos em um grafo de serialização e, portanto, a técnica apresenta a maior sobrecarga de gerenciamento quando comparada as demais técnicas propostas.

Tabela 10 – Técnicas de serialização baseadas em SNAPSHOT comparadas a proposta de *daemons*.

	Snapshot	Análise estática	Gerenciador de bloqueios externo	Serializable Snapshot	Precisely Serializable Snapshot	Write Snapshot	Daemon Snapshot
Aspectos positivos							
Oferece garantia de serialização		✓	✓	✓	✓	✓	✓
Suporta transações *ad-hoc*	✓			✓	✓	✓	✓
Transações somente leitura nunca são abortadas	✓	✓	✓	✓	✓	✓	✓
Aspectos negativos							
Bloqueia algumas transações serializáveis (falsos positivos)	×	×	×	×		×	×
Sobrecarga imposta sobre elementos lidos (read-set)				×	×	×	
Sobrecarga imposta sobre elementos escritos (write-set)	×	×	×	×	×	×	×
Sobrecarga associada a detecção de ciclos					×		
Requer alteração no SGBD				×	×	×	×
Requer alteração no código da aplicação		×	×				
Requer alteração na base de dados		×					×
Mantém dados sobre transações encerradas por um tempo				×	×		
Outras características							
Estratégia de monitoramento	ww	ww	ww	rw	ciclo	rw	ww

Os algoritmos SERIALIZABLE SNAPSHOT, PRECISELY SERIALIZABLE SNAPSHOT e WRITE-SNAPSHOT são capazes de garantir serialização com suporte simultâneo a transações *ad-hoc*, mas exigem o monitoramento dos elementos lidos pelas transações. Por sua vez, DAEMON SNAPSHOT é capaz do mesmo impondo sobrecarga de gerenciamento apenas sobre os elementos escritos.

Vale destacar que, embora a estratégia adotada exija alterações na base de dados, tais alterações não são de caráter estrutural, ou seja, nenhuma classe precisa ser remodelada ou inserida de modo a se obter os benefícios apresentados, cabendo apenas a introdução da cláusula **daemon** na definição do esquema.

Finalmente, DAEMON SNAPSHOT utiliza o mecanismo de identificação de conflitos do tipo *write-write*, tornando-o uma opção viável para ambientes transacionais distribuídos que exigem garantia de consistência dos dados.

4 Testes e resultados

Sistemas de *benchmark* são responsáveis por gerar cargas de trabalho contendo as ações de diversas tarefas transacionais, as quais são executadas concorrentemente de modo a simular um ambiente competitivo e permitir o monitoramento do desempenho de um SGBD diante de tal situação. Tradicionalmente esses sistemas são projetados para se medir o desempenho com base em duas métricas principais: vazão ou *throughput* e tempo de resposta. Enquanto a vazão fornece o número de transações que executam com sucesso em uma determinada unidade de tempo, o tempo de resposta informa o tempo total decorrido na execução de uma transação em particular ou o tempo médio obtido considerando-se um histórico de execuções.

Um conjunto padrão de sistemas de *benchmark*, denominado *Transaction Processing Performance Council — TPC* (TPC, 2013), foi formado pela indústria com o propósito de definir exatamente o que deve ser medido e sob quais circunstâncias. Entretanto, sistemas de *benchmark* como TPC-C não desencadeiam a ocorrência do fenômeno Escrita Oblíqua (REVILAK, 2011; FEKETE et al., 2005) e, portanto, não são adequados para se medir a eficiência de um algoritmo fundamentado em SNAPSHOT e adaptado para oferecer garantia de serialização. Em razão deste trabalho lidar com essa garantia, o teste de desempenho deve considerar cargas de trabalho não serializáveis sob SNAPSHOT de modo a se mensurar o custo associado à obtenção de históricos serializáveis. Por esse motivo, o sistema de *benchmark* adotado emprega características do *benchmark* SmallBank (ALOMARI et al., 2008a) e da plataforma de testes utilizada por REVILAK, 2011.

4.1 Plataforma de testes em Draco

Originalmente o sistema de *benchmark* SmallBank foi projetado para o modelo relacional visando a ocorrência de um problema conhecido como "Anomalia de transação somente leitura sob Snapshot", identificado por FEKETE; O'NEIL; O'NEIL, 2004. Um sistema de *benchmark* similar, denominado SmallBank++, foi construído neste trabalho para o modelo de gerenciamento de dados complexos utilizado em Draco e também visa a ocorrência de anomalias quando as transações executam sob o isolamento SNAPSHOT. Assim como em SmallBank, o sistema de *benchmark* SmallBank++ oferece um conjunto de tarefas transacionais que refletem as funcionalidades de um sistema bancário simples. O esquema utilizado por SmallBank++ é constituído por três classes: Cliente, Poupanca e ContaCorrente (veja Figura 36).

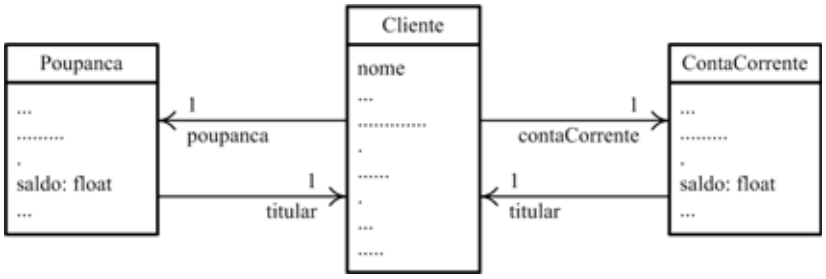

Figura 36 – Representação esquemática das classes que constituem o *benchmark* SmallBank++.

Segundo o diagrama esquemático de SmallBank++, o atributo poupanca de um objeto da classe Cliente contém uma referência a um objeto da classe Poupanca de modo a estabelecer um vínculo entre os mesmos, ou seja, a conta poupança do cliente. De modo análogo, o atributo contaCorrente estabelece a relação entre o cliente e sua conta corrente. Por sua vez, os objetos das classes Poupanca e ContaCorrente possuem o atributo titular, responsável por estabelecer a relação inversa, ou seja, ligar cada tipo de conta a seu respectivo cliente. O diagrama também deixa explícito o fato de que os objetos das classes Poupanca e ContaCorrente possuem o atributo saldo, responsável por manter o saldo de cada conta, e os objetos da classe Cliente possuem o atributo nome, o qual armazena o nome do cliente. Além disso, todo cliente do banco deve possuir, obrigatoriamente, uma conta poupança e uma conta corrente.

O *benchmark* SmallBank++ impõe a seguinte regra de negócio aos clientes do banco: "Um valor pode ser debitado da conta poupança ou conta corrente de um cliente, mesmo não havendo saldo suficiente, desde que o saldo total disponível nas duas contas possa cobrir a retirada."

Na Figura 37 é exibida a linguagem que define o esquema utilizado por SmallBank++ juntamente com a cláusula *daemon* que garante a não ocorrência do fenômeno Escrita Oblíqua em virtude da regra de negócio existente.

SmallBank++ é composto por cinco tarefas transacionais, as quais são executadas aleatoriamente e com igual probabilidade:

1. Checa saldos — Verifica o saldo das duas contas de um cliente.

2. Movimenta conta poupança — Efetua um depósito ou saque da conta poupança de um cliente. Se um saque estiver sendo feito, então o sistema deve consultar também o saldo da conta corrente de modo a garantir que a soma do saldo das duas contas possa cobrir o valor solicitado. O valor é movimentado apenas na conta poupança caso a operação possa ser realizada.

3. Movimenta conta corrente — Efetua um depósito ou saque da conta corrente de um cliente. Se um saque estiver sendo feito, então o sistema deve consultar também o

```
class Cliente
{
    attribute string nome;
    attribute Poupanca poupanca;
    attribute ContaCorrente contaCorrente;
    .
    .
    .
};

class Poupanca
{
    attribute Cliente titular;
    attribute float saldo;
    .
    .
    .
    daemon titular.contaCorrente;
};

class ContaCorrente
{
    attribute Cliente titular;
    attribute float saldo;
    .
    .
    .
    daemon titular.poupanca;
};
```

Figura 37 – Linguagem de definição de dados para as classes que constituem o *benchmark* SmallBank++.

saldo da conta poupança de modo a garantir que a soma do saldo das duas contas possa cobrir o valor solicitado. O valor é movimentado apenas na conta corrente caso a operação possa ser realizada.

4. Transferência de fundos — Transfere o saldo total disponível nas duas contas de um cliente para a conta corrente de outro cliente.

5. Compensação de cheque — Realiza a compensação de um cheque emitido por um cliente. A compensação só é realizada se o saldo total disponível nas duas contas do cliente for suficiente para cobrir o cheque. O valor sendo compensado é retirado da conta corrente caso a operação seja efetuada.

Inicialmente o sistema de *benchmark* faz a carga da base de dados, onde um conjunto de n clientes com suas respectivas contas é criado. Durante este processo um valor aleatório entre \$0 e \$10.000 é gerado para cada cliente, sendo que metade deste valor é atribuído a sua conta poupança e a outra metade é atribuída a sua conta corrente. Em seguida 10% dos clientes são escolhidos também aleatoriamente de modo que 90% das tarefas transacionais sejam executadas sobre esses clientes (*hotspots*). Um número $\frac{1}{10}n \cdot f$ de tarefas é então gerado, onde $\frac{1}{10}n$ equivale a 10% do total de clientes e f corresponde ao fator de carga de trabalho. Dado o caráter aleatório durante a atribuição de uma tarefa a um cliente, teoricamente para $f = 1$ tem-se uma tarefa a ser executada para cada cliente no conjunto de *hotspots*, para $f = 2$ tem-se duas tarefas para cada e assim sucessivamente — lembrando que 90% das tarefas irão agir sobre o conjunto de *hotspots*

e o restante irá agir sobre os demais clientes. Finalmente, as tarefas são processadas de modo que um número t de transações execute concorrentemente e as ações de cada tarefa sejam disparadas uma por vez de forma intercalada.

Desta forma, SmallBank++ permite não apenas que o número n de clientes seja especificado como também o fator f de carga de trabalho que irá agir sobre 10% dos clientes e o número t de transações concorrentes.

Os testes descritos nas seções seguintes monitoram o desempenho obtido por Draco quando se utiliza o isolamento Snapshot em sua forma natural e quando a abordagem Daemon Snapshot é empregada. São consideradas métricas como o número de transações abortadas e a carga associada às operações de leitura e escrita na base de dados em virtude das operações de escrita virtuais requeridas pela técnica proposta. Os testes foram executados em um notebook Dell XPS L502X com processador Intel Core i7–2670QM 2.20 GHz, 8 GB de memória RAM e Windows 7 64 bits. Draco foi compilado utilizando-se a plataforma NetBeans IDE 7.3.1 e o compilador C++ Cygwin 3.4.4.

4.1.1 Teste 1 – Transações abortadas devido a erros de serialização

O objetivo deste teste foi identificar o número de transações abortadas devido a erros de serialização quando se utiliza o isolamento Snapshot em sua forma original e quando a abordagem Daemon Snapshot é empregada.

Inicialmente gerou-se uma base de dados com 10 mil clientes e suas respectivas contas bancárias. Mil clientes foram então escolhidos como conjunto de *hotspots* e mil tarefas transacionais foram processadas aleatoriamente segundo as regras descritas anteriormente para o *benchmark* SmallBank++, ou seja, cerca de 90% dessas tarefas foram processadas sobre o conjunto de *hotspots* e as demais foram processadas sobre os outros nove mil clientes.

O teste foi executado considerando-se uma única transação em atividade e 8, 16, 32, 64 e 128 transações concorrentes. De modo a se obter resultados mais precisos, a carga de trabalho aleatória foi processada por dez vezes seguidas para cada configuração de nível de concorrência, sendo que apenas a média final do número de transações abortadas foi considerada. Os resultados obtidos são mostrados na Figura 38.

Conforme pode-se observar, não há erros de concorrência em um ambiente sem competitividade e, portanto, todas as tarefas são processadas com sucesso tanto no isolamento Snapshot quanto no isolamento Daemon Snapshot.

Com 8 transações simultâneas, nota-se uma taxa de erro de serialização abaixo de 1% em ambas as técnicas. Com 16 ou mais transações concorrentes é possível observar uma taxa percentual de erro de serialização maior na abordagem Daemon Snapshot. Mesmo assim, somente com 128 transações atuando concorrentemente é que se observou

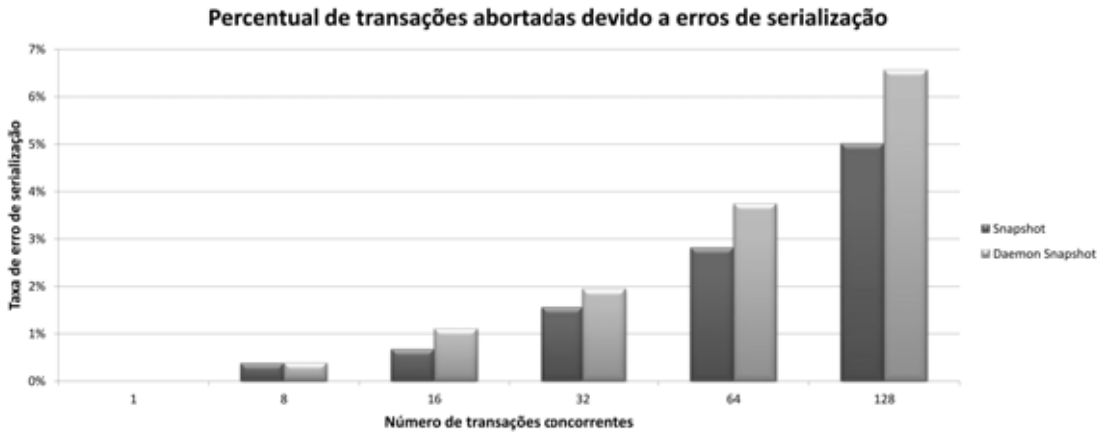

Figura 38 – Percentual de transações abortadas devido a erros de serialização considerando-se o processamento de mil tarefas transacionais.

um aumento de pouco mais de um ponto percentual no número de transações abortadas quando se utiliza Daemon Snapshot. Isso significa um cancelamento de aproximadamente 50 das mil tarefas processadas sob o isolamento Snapshot e 65 dessas sob o isolamento Daemon Snapshot quando 128 transações executam concorrentemente.

Um número maior de transações abortadas sob Daemon Snapshot era esperado em virtude das operações de escrita adicionais disparadas automaticamente para todos os elementos em um caminho de dependências definido por um *daemon*, aumentando, portanto, a possibilidade de ocorrência de ações conflitantes segundo o protocolo Snapshot.

Durante a execução dos testes a base de dados foi levada por algumas vezes a um estado inconsistente sob o isolamento Snapshot, mas sempre se manteve íntegra quando a abordagem Daemon Snapshot foi empregada. Em razão disso e considerando-se os resultados expostos na Figura 38, conclui-se que a garantia de serialização é obtida em ambientes altamente competitivos a um custo de aproximadamente um ponto percentual a mais sobre o percentual de transações que são abortadas quando se utiliza o isolamento Snapshot.

Na Figura 39 tem-se a estimativa percentual de tarefas que conduzem a base de dados a um estado inconsistente quando as transações executam sob o isolamento Snapshot. Nota-se que a media que o número de transações concorrentes aumenta, aumenta-se também a chance de ocorrência do fenômeno Escrita Oblíqua. Sob o isolamento Daemon Snapshot o percentual de tarefas que violam a consistência dos dados é 0% em todos os casos.

4.1.2 Teste 2 – Carga de leitura/escrita exigida

O objetivo deste teste foi identificar a quantidade de bytes que são lidos e escritos na base de dados quando se utiliza o isolamento Snapshot em sua forma original e

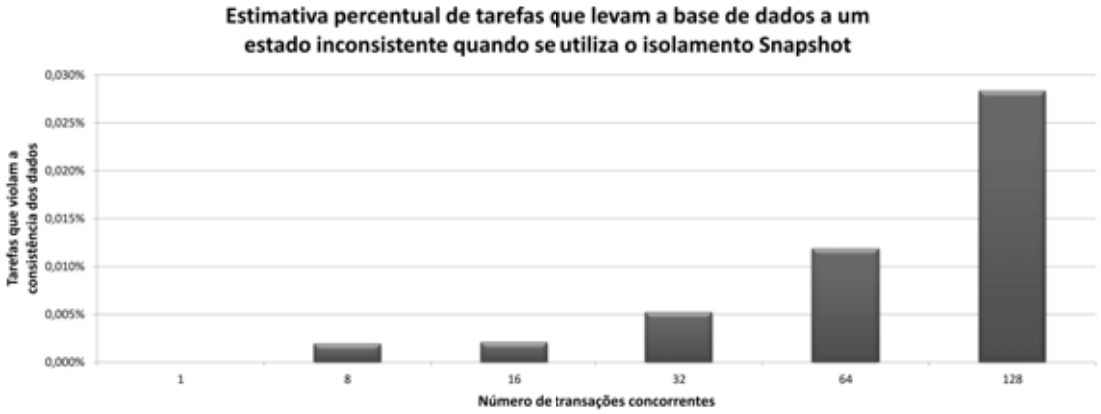

Figura 39 – Estimativa percentual de tarefas que levam a base de dados a um estado inconsistente quando se utiliza o isolamento Snapshot.

quando a abordagem Daemon Snapshot é empregada.

Inicialmente gerou-se uma base de dados com 10 mil clientes, cada um com suas respectivas contas bancárias. Mil clientes foram então escolhidos como conjunto de *hotspots*, mas desta vez 10 mil tarefas transacionais foram processadas aleatoriamente segundo as regras descritas para o *benchmark* SmallBank++.

Assim como anteriormente, o teste foi executado considerando-se uma única transação em atividade e 8, 16, 32, 64 e 128 transações concorrentes. Mais uma vez, de modo a se obter resultados mais precisos, a carga de trabalho aleatória foi processada por dez vezes seguidas para cada configuração de nível de concorrência, sendo que apenas a média final da quantidade de bytes lidos e escritos foi considerada. Os resultados obtidos durante o monitoramento das operações de leitura são mostrados na Figura 40.

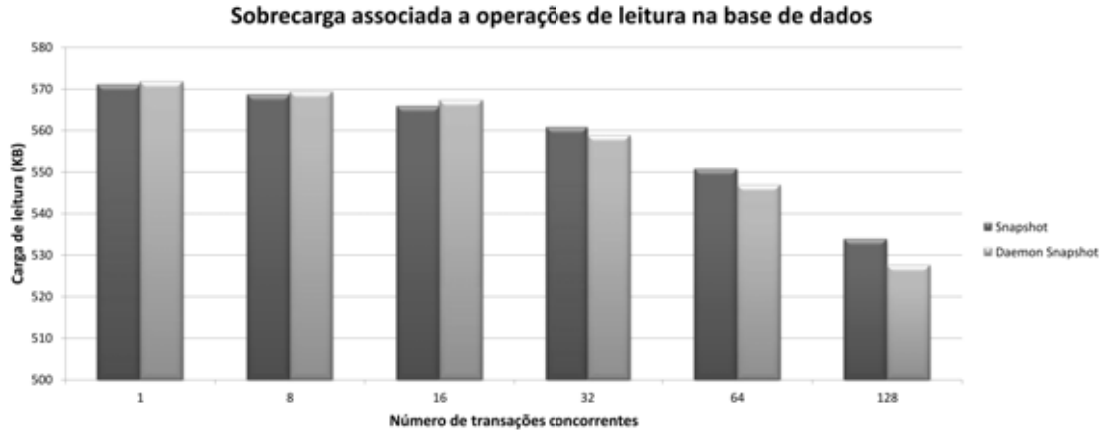

Figura 40 – Sobrecarga associada a operações de leitura na base de dados considerando-se o processamento de 10 mil tarefas transacionais.

À medida que o número de transações concorrentes aumenta, aumenta também a possibilidade de ocorrência de conflitos, possibilidade esta ainda maior sob o isolamento Daemon Snapshot em virtude das operações de escrita virtuais. Conforme observa-se

na Figura 40, a sobrecarga associada a operações de leitura na base de dados é praticamente a mesma em ambas as técnicas na maioria dos casos. Nota-se que a técnica Daemon Snapshot impõe uma maior sobrecarga de leitura em um ambiente de baixa competitividade (aproximadamente até 16 transações concorrentes), enquanto a mesma impõe uma menor sobrecarga de leitura à medida que a concorrência aumenta. A justificativa para tal ganho está relacionada ao teste anterior, onde é possível observar um aumento na diferença percentual de transações abortadas à medida que a concorrência aumenta. Por esse motivo, Daemon Snapshot aborta mais transações que Snapshot nesses ambientes e, consequentemente, a sobrecarga de leitura passa a ser menor em razão de um número maior de transações canceladas.

Entretanto, mesmo em um ambiente pouco competitivo (até 16 transações simultâneas), a sobrecarga de leitura exigida por Daemon Snapshot ainda é baixa quando comparada a Snapshot, conforme pode-se observar na Figura 41.

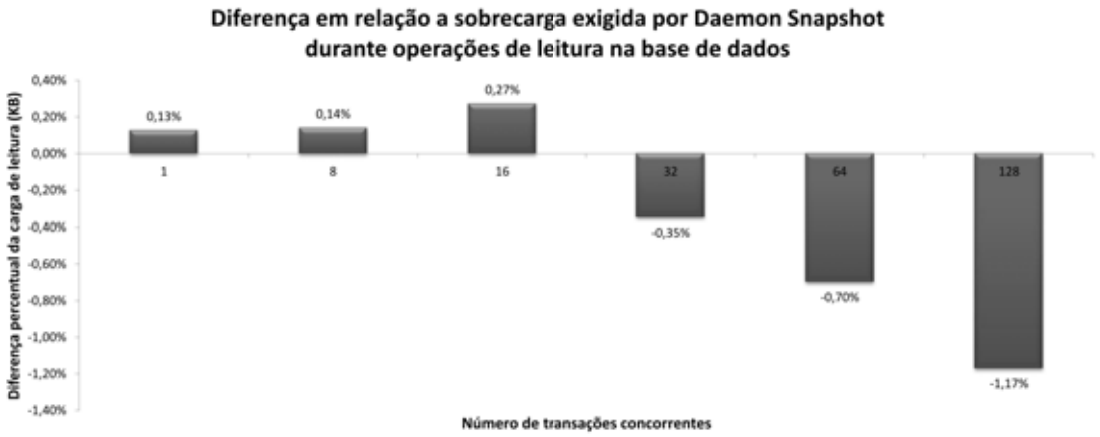

Figura 41 – Comparativo entre a sobrecarga exigida por Daemon Snapshot e Snapshot durante operações de leitura na base de dados.

Daemon Snapshot exigiu 0,27% a mais das operações de leitura na base de dados quando o ambiente foi alvo de 16 transações simultâneas. A maior diferença ocorreu quando 128 tarefas foram executadas concorrentemente e, neste caso, Daemon Snapshot efetuou 1,17% menos leituras na base de dados em razão de um número maior de transações abortadas.

Os resultados obtidos durante o monitoramento das operações de escrita são mostrados na Figura 42.

Assim como para as operações de leitura, a sobrecarga associada a operações de escrita na base de dados é praticamente a mesma em ambas as técnicas na maioria dos casos. Da mesma forma que antes, Daemon Snapshot aborta mais transações que Snapshot à medida que a concorrência aumenta e, consequentemente, a sobrecarga de escrita também passa a ser menor em razão de um número maior de transações canceladas,

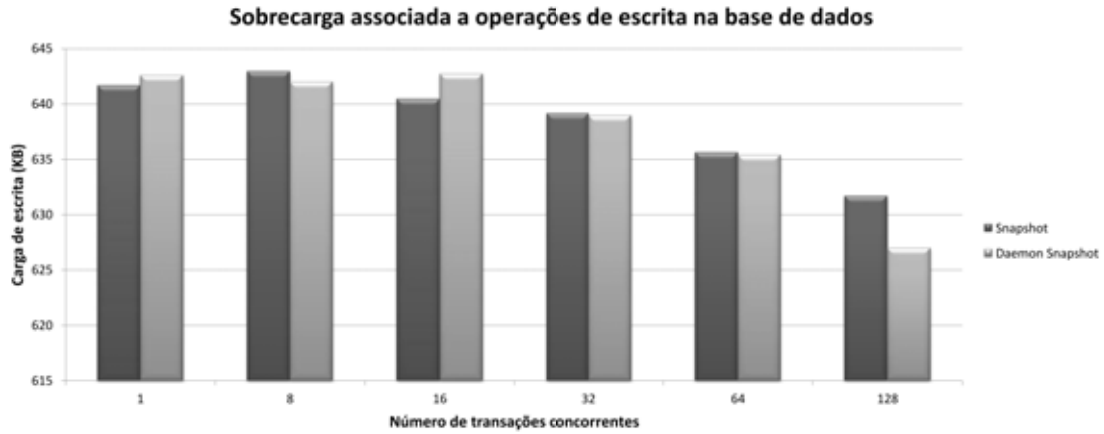

Figura 42 – Sobrecarga associada a operações de escrita na base de dados considerando-se o processamento de 10 mil tarefas transacionais.

situação esta observada na Figura 42 quando 128 transações atuam concorrentemente.

A diferença entre a sobrecarga de escrita exigida por Daemon Snapshot e Snapshot é exibida na Figura 43.

Figura 43 – Comparativo entre a sobrecarga exigida por Daemon Snapshot e Snapshot durante operações de escrita na base de dados.

Daemon Snapshot exigiu 0,36% a mais das operações de escrita na base de dados quando o ambiente foi alvo de 16 transações simultâneas. A maior diferença ocorreu quando 128 tarefas foram executadas concorrentemente e, neste caso, Daemon Snapshot efetuou 0,74% menos escritas na base de dados em razão de um número maior de transações abortadas.

Os resultados do teste comprovam que a garantia de serialização é obtida com Daemon Snapshot a um custo inferior a 0,5% sobre as operações de leitura e escrita na base de dados.

4.1.3 Teste 3 – Desempenho

O objetivo deste teste foi avaliar o impacto no desempenho causado pelas estruturas de dados responsáveis pelo gerenciamento de objetos e controle de *daemons* em Draco.

Assim como nos testes anteriores, gerou-se uma base de dados com 10 mil clientes e suas respectivas contas bancárias. Mil clientes foram escolhidos como conjunto de *hotspots* e, desta vez, uma carga de trabalho de 200 mil tarefas foi processada.

O teste foi executado considerando-se uma única transação em atividade e 8, 16, 32, 64 e 128 transações concorrentes. Ao contrário dos testes anteriores, ao invés da média final foi computado o tempo total de processamento das 200 mil tarefas para cada configuração de nível de concorrência. Os resultados obtidos são mostrados na Figura 44.

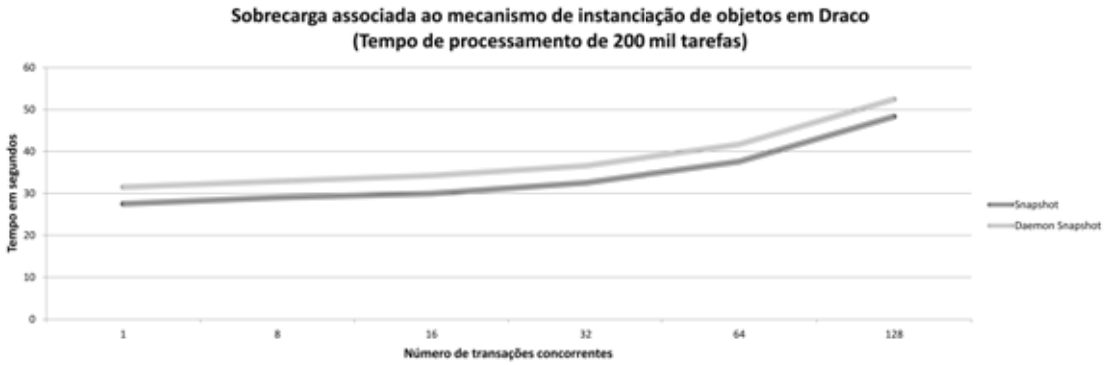

Figura 44 – Tempo total de processamento considerando-se uma carga de trabalho de 200 mil tarefas.

Conforme pode-se observar, embora os testes anteriores tenham comprovado a existência de uma sobrecarga de operações de leitura/escrita na base de dados praticamente equivalente tanto em Snapshot quanto em Daemon Snapshot, o mecanismo de gerenciamento de objetos em Draco impõe uma sobrecarga extra sobre o isolamento Daemon Snapshot em virtude de um número maior de objetos que são instanciados quando há a presença de *daemons*. Essa sobrecarga, associada ao mecanismo de instanciação de objetos, é detalhada na Figura 45.

De acordo com a Figura 45, sob o isolamento Daemon Snapshot, o tempo de processamento das transações aumenta em aproximadamente 13% na maioria dos casos quando há a presença de *daemons*, ou seja, a sobrecarga não está relacionada ao isolamento propriamente dito, mas sim ao custo de instanciação dos objetos presentes na cadeia de dependências definida por um *daemon*. Como consequência, é possível otimizar o tempo de processamento simplesmente oferecendo às transações um *cache* de objetos temporário. De modo a verificar tal possibilidade, o gerenciador Draco foi dotado da capacidade de usar ou não esse sistema de cache e os testes foram executados novamente considerando-se este recurso. Os resultados obtidos são exibidos na Figura 46.

Figura 45 – Sobrecarga associada à instanciação de objetos quando há a presença de *daemons*.

Figura 46 – Sobrecarga associada à instanciação de objetos com e sem a utilização do *cache* de objetos.

O uso do *cache* de objetos reduz a sobrecarga imposta sobre o tempo de processamento das transações de 13% para aproximadamente 7% na maioria dos casos, confirmando portanto a hipótese de tal sobrecarga estar associada ao gerenciamento dos objetos e não à técnica Daemon Snapshot propriamente dita. Em razão disso, e considerando-se os resultados anteriores acerca da sobrecarga sobre operações de leitura/escrita, conclui-se que a busca por métodos mais eficientes de gerenciamento de objetos em Draco pode levar o tempo de processamento de transações sob o isolamento Daemon Snapshot a um tempo de processamento ainda mais próximo de execuções sob o isolamento Snapshot.

4.1.4 Considerações

Conforme demonstrou o Teste 1, as operações de escrita virtuais decorrentes do uso da técnica Daemon Snapshot aumentam a possibilidade de ações conflitantes e, portanto, um número maior de transações é abortada. Entretanto, Daemon Snapshot é capaz de garantir apenas históricos serializáveis. Mesmo diante de operações de escrita virtuais, o Teste 2 comprovou que a sobrecarga associada às operações de leitura/escrita

em uma base de dados Draco é praticamente equivalente nos isolamentos SNAPSHOT e DAEMON SNAPSHOT, liberando o subsistema de armazenamento de Draco de qualquer responsabilidade relacionada a perda de desempenho. Contudo, conforme pode-se concluir pelo Teste 3, existe uma sobrecarga de aproximadamente 13% no tempo de processamento das transações quando há a presença de *daemons*. Essa sobrecarga, entretanto, não está associada à técnica DAEMON SNAPSHOT propriamente dita, mas sim ao mecanismo utilizado em Draco para instanciação de objetos complexos.

Observou-se também pelos resultados obtidos no Teste 3 que quando as transações são dotadas de um *cache* de objetos temporário, a carga de processamento exigida durante a instanciação dos objetos presentes na cadeia de dependências de um *daemon* é reduzida praticamente à metade, o que permite oferecer um ambiente transacional seguro do ponto de vista da consistência dos dados a um custo aproximado de 7% para a maioria dos níveis de concorrência avaliados e a um custo de 4% para ambientes altamente competitivos (128 transações simultâneas) quando comparado ao isolamento SNAPSHOT.

4.2 Plataforma de testes relacional

Nesta seção a estratégia de *daemons* é aplicada aos gerenciadores convencionais SQL Server, Firebird e Oracle de modo a comparar o grau de consistência e concorrência obtidos pelos níveis de isolamento existentes diante da técnica proposta. Diferentemente de Draco, o qual lida com *daemons* diretamente em seu núcleo, a abordagem de *daemons* é simulada nesses gerenciadores por meio de *triggers* especiais que simulam seu processamento. Essas *triggers*, denominadas DAEMON TRIGGERS, são responsáveis por efetuar fisicamente as operações de escrita que seriam realizadas virtualmente caso o sistema gerenciador de banco de dados incorpore a estratégia de *daemons* em seu núcleo.

O sistema de *benchmark* SmallBank++ descrito anteriormente foi adaptado[1] de modo que pudesse ser aplicado aos gerenciadores relacionais SQL Server, Firebird e Oracle. Assim como em sua versão para o gerenciador Draco, a versão de SmallBank++ para o modelo relacional também visa a ocorrência de anomalias quando as transações executam sob o isolamento SNAPSHOT. Contudo, o esquema utilizado por SmallBank++ para o modelo relacional é constituído por quatro tabelas: Clientes, Poupancas, ContasCorrentes e Notificacoes (veja Figura 47).

Segundo o diagrama relacional de SmallBank++, cada linha na tabela Clientes representa um cliente do banco e as linhas nas tabelas Poupancas e ContasCorrentes representam, respectivamente, a conta poupança e conta corrente de um cliente. De acordo com o esquema, todo cliente do banco deve possuir as duas contas, as quais por sua vez

[1] A versão de SmallBank++ para o modelo relacional foi implementada em C# utilizando a plataforma de desenvolvimento Visual Studio Professional 2012 (MICROSOFT, 2013b).

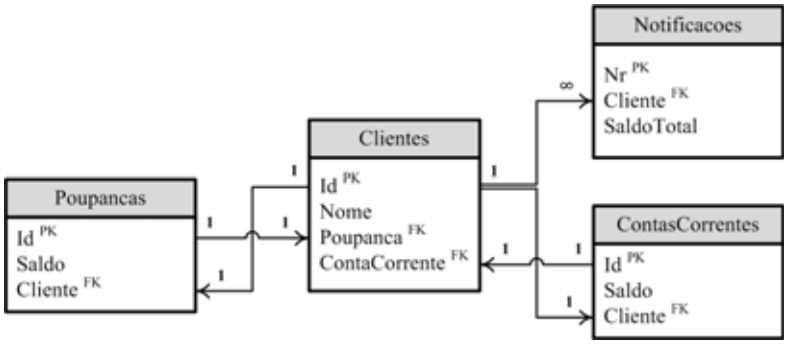

Figura 47 – Representação esquemática das tabelas que constituem o *benchmark* SmallBank++ em sua versão relacional.

devem estar ligadas a um único cliente. Enquanto os atributos Poupanca e ContaCorrente ligam um cliente a suas duas contas, o atributo Cliente em cada uma das contas estabelece a relação inversa ligando uma conta a seu titular. Cada linha na tabela Notificacoes representa uma notificação que foi enviada a um cliente ou por meio do serviço postal tradicional ou via e-mail comunicando o saldo atual somado das duas contas do cliente.

Em sua versão relacional o *benchmark* SmallBank++ impõe a seguinte regra de negócio aos clientes do banco: "Um valor pode ser debitado da conta poupança ou conta corrente de um cliente, mesmo não havendo saldo suficiente, desde que o saldo total disponível nas duas contas possa cobrir a retirada. Além disso, uma notificação a um cliente nunca deve ser feita duas vezes para um mesmo valor de saldo."

A versão de SmallBank++ para o modelo relacional é composta por seis tarefas transacionais:

1. Checa saldos (50% da carga de trabalho) — Verifica o saldo das duas contas de um cliente.

2. Movimenta conta poupança (10% da carga de trabalho) — Efetua um depósito ou saque da conta poupança de um cliente. Se um saque estiver sendo feito, então o sistema deve consultar também o saldo da conta corrente de modo a garantir que a soma do saldo das duas contas possa cobrir o valor solicitado. O valor é movimentado apenas na conta poupança caso a operação possa ser realizada.

3. Movimenta conta corrente (10% da carga de trabalho) — Efetua um depósito ou saque da conta corrente de um cliente. Se um saque estiver sendo feito, então o sistema deve consultar também o saldo da conta poupança de modo a garantir que a soma do saldo das duas contas possa cobrir o valor solicitado. O valor é movimentado apenas na conta corrente caso a operação possa ser realizada.

4. Transferência de fundos (10% da carga de trabalho) — Transfere o saldo total disponível nas duas contas de um cliente para a conta corrente de outro cliente.

5. Compensação de cheque (10% da carga de trabalho) — Realiza a compensação de um cheque emitido por um cliente. A compensação só é realizada se o saldo total disponível nas duas contas do cliente for suficiente para cobrir o cheque. O valor sendo compensado é retirado da conta corrente caso a operação seja efetuada.

6. Notificação (10% da carga de trabalho) — Obtém o saldo somado das duas contas de um cliente e verifica se o mesmo já foi notificado sobre o valor atual consultando a tabela de notificações. Uma notificação é enviada ao cliente somente se não for encontrada uma entrada na tabela.

Inicialmente o sistema de *benchmark* carrega a base de dados com mil clientes e suas respectivas contas. Durante este processo um valor aleatório entre \$0 e \$10.000 é gerado para cada cliente, sendo que metade deste valor é atribuído a sua conta poupança e a outra metade é atribuída a sua conta corrente.

A carga de trabalho a ser processada é então dividida em 10 etapas (S_1 a S_{10}), onde 100 tarefas transacionais são executadas em S_1, 200 tarefas são executadas em S_2 e assim sucessivamente até S_{10} onde 1.000 tarefas são executadas. No final um total de 5.500 tarefas são processadas. Antes do início de cada etapa 10% dos clientes são escolhidos aleatoriamente de modo que 90% das tarefas sejam executadas sobre esses clientes (*hotspots*). Aproximadamente 50% da carga de trabalho é constituída pela Tarefa 1 (Checa saldos) e o restante é distribuído com igual probabilidade entre as demais tarefas, caracterizando portanto uma carga de trabalho onde 50% das tarefas são de leitura e os outros 50% são de leitura e escrita. O sistema de *benchmark* registra então um erro de concorrência para cada tentativa mal sucedida de *commit*. No final de cada etapa SmallBank++ analisa a consistência da base de dados verificando se o saldo somado das duas contas de cada um dos clientes ficou negativo e checando se uma mesma notificação foi enviada a um cliente mais de uma vez.

A carga de trabalho é processada primeiramente considerando-se uma única transação em atividade. Logo em seguida a base de dados é limpa e o processo de geração de clientes e contas se repete antes que a próxima carga de trabalho seja disparada, desta vez considerando-se 8, 16, 32, 64 e 128 transações concorrentes. Em todos os casos as ações de cada tarefa são processadas uma por vez e de forma intercalada de acordo com o número de transações atualmente em atividade.

4.2.1 Testes

Os testes descritos a seguir monitoram o nível de concorrência e consistência obtidos quando o *benchmark* SmallBank++ é executado sobre instâncias do SQL Server, Firebird e Oracle. Foram considerados os níveis de isolamento oferecidos por cada ge-

renciador e uma possível implementação do isolamento DAEMON SNAPSHOT, o qual foi simulado por meio da estratégia de DAEMON TRIGGERS.

O primeiro teste foi realizado sobre uma instância do SQL Server 2012 Express Edition (MICROSOFT, 2013a) considerando-se os isolamentos READ COMMITTED, SNAPSHOT, SERIALIZABLE e DAEMON SNAPSHOT. Os resultados obtidos por SmallBank++ são mostrados na Tabela 11, onde a coluna Tx representa o número de transações concorrentes e cada número k na tabela indica o número de tarefas que não encerraram com sucesso devido a um erro de serialização. Quando o número k é seguido pela expressão $*n$, significa que a base de dados foi levada a um estado inconsistente n vezes.

Tabela 11 – SmallBank++ sobre SQL Server 2012.

READ COMMITTED

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	2	3	7	11	8	10	13	15	18	18	**105**
16	5	7	8	14	12	17	22	21	34	29*1	**169*1**
32	3	13	10	26	26	39	48	41*1	70	55	**331*1**
64	11	22*1	22	37*1	54*2	57	56	89*1	101*1	99	**548*6**
128	3	20*2	22*1	54	69	90*2	106*1	108*2	117	140	**729*8**

SNAPSHOT

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	4	1*1	2	7	9	6	4	4	7	5	**49*1**
16	1	6	5	11	15*1	8*1	13	12*2	9*1	12	**92*5**
32	4	6	16	10	21	14	29*4	20	23	38	**181*4**
64	5	12*1	13*3	23*2	23*2	22	40*3	41*2	51*4	55*1	**285*18**
128	8*1	12*2	14*1	33*3	37*4	58*2	53*2	61*4	74*6	92*3	**442*28**

SERIALIZABLE

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	2	15	13	17	16	30	25	41	39	49	**247**
16	10	16	21	33	31	38	42	55	81	67	**394**
32	10	30	26	53	56	87	81	126	105	140	**714**
64	23	51	61	97	104	117	156	168	218	215	**1210**
128	20	57	91	125	145	185	197	226	255	277	**1578**

DAEMON SNAPSHOT

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	6	2	4	9	11	9	9	7	11	9	**77**
16	3	9	5	18	20	14	20	27	17	24	**157**
32	4	9	22	17	33	22	48	34	37	51	**277**
64	8	19	25	38	39	42	59	61	88	82	**461**
128	12	25	33	52	61	89	88	107	125	146	**738**

O segundo teste foi realizado sobre uma instância do Firebird 2.5.2 (FIREBIRD, 2013) considerando-se os isolamentos READ COMMITTED, SNAPSHOT, SERIALIZABLE e DAEMON SNAPSHOT. Quando SERIALIZABLE é requisitado, Firebird de fato utiliza o isolamento SNAPSHOT TABLE STABILITY, que difere do isolamento SNAPSHOT pelo fato de apenas uma transação por vez poder escrever em uma tabela (VALDERRAMA, 2013). Os resultados obtidos por SmallBank++ são mostrados na Tabela 12.

O último teste foi realizado sobre uma instância do Oracle 11g Express Edi-

Tabela 12 – SmallBank++ sobre Firebird 2.5.2.

READ COMMITTED

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	1	0*1	0	1	2	3	0	1	3	2	**13***1
16	1	1	0	0	3	2	4	2*2	4*1	1	**18***3
32	2	2	10	3	4	6	9*3	11	7	12	**66***3
64	2	5	2*2	10*2	7*3	7*2	17*2	14*1	14*3	19*1	**97***16
128	6*1	6*3	9	13*3	20*1	32*3	25*1	25*2	35*3	34*5	**205***22

SNAPSHOT

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	4	2*1	2	8	11	6	4	4	7	6	**54***1
16	3	6	6	11	18*1	15*1	16	13*2	10*1	15*1	**113***6
32	3	7*1	17	13	21	18	34*4	23	26*1	41	**203***6
64	5	13*1	16*3	27*3	27*3	24	43*3	45*3	61*5	62*1	**323***22
128	8*1	12*2	16*2	39*4	41*7	67*4	56*3	69*3	81*6	102*5	**491***37

SERIALIZABLE

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	47	77	114	177	190	253	264	324	370*1	397	**2213***1
16	53	84	133	163	208	251	243*1	362	404	443	**2344***1
32	51	78	136	194	228	258	295	355	386	433	**2414**
64	61	98	115	190	241	288	318	351	397	466	**2525**
128	42	99	125	211	236	292	320	364	435	470	**2594**

DAEMON SNAPSHOT

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	6	3	4	11	14	9	9	7	12	10	**85**
16	5	10	6	18	23	21	20	31	20	29	**183**
32	5	12	25	20	35	27	54	34	47	57	**316**
64	11	21	27	44	46	44	68	69	102	96	**528**
128	12	27	43	63	66	103	95	119	134	161	**823**

tion (ORACLE, 2013a) considerando-se os isolamentos READ COMMITTED, SNAPSHOT e DAEMON SNAPSHOT. Embora Oracle ofereça o isolamento SERIALIZABLE, este é de fato uma implementação do protocolo SNAPSHOT (PORTS; GRITTNER, 2012) e portanto é tratado como tal neste trabalho. Além disso, Oracle apresenta algumas peculiaridades em seu sistema transacional que limitam a concorrência quando se utiliza o isolamento SNAPSHOT. De modo a obter uma maior concorrência, os testes também foram realizados com a opção ROWDEPENDENCIES definida durante a criação das tabelas de SmallBank++ para que o controle de alterações sobre os registros pelas transações seja feito a nível de linha (row-level) ao invés de a nível de bloco (block-level), evidentemente ao custo de alguns bytes extras de armazenamento por linha (ORACLE, 2013b). Devido a isso, os testes sobre os isolamentos SNAPSHOT e DAEMON SNAPHOT foram realizados duas vezes: primeiro utilizando-se a configuração padrão de modo de bloqueio por bloco (block-level), a qual é reportada neste trabalho como SNAPSHOT* e DAEMON SNAPHOT*, e então no modo de bloqueio por linha (row-level), a qual é reportada como SNAPSHOT e DAEMON SNAPHOT. Os resultados obtidos por SmallBank++ são mostrados Tabela 13.

Os gráficos de serialização e inconsistência resultantes da análise dos dados ob-

Tabela 13 – SmallBank++ sobre Oracle 11g.

READ COMMITTED

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	1	0*1	0	1	2	3	0	1	3	2	**13***1
16	1	1	0	0	3	2	4	2*2	4*1	1	**18***3
32	2	2	10	3	4	6	9*3	11	7	12	**66***3
64	2	5	2*2	10*2	7*3	7*2	17*2	14*1	14*3	19*1	**97***16
128	6*1	6*3	9	13*3	20*1	32*3	25*1	25*2	35*3	34*5	**205***22

SNAPSHOT*

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	14	22	25	38	47	53	56	63	86	95	**499**
16	18	31	47	57	76	73	93	96*1	107	122	**720***1
32	21	39	53	78	103	114	152	155	165	199	**1079**
64	24	53	77	110	151	153*1	183	217	259*1	270	**1497***2
128	29	50	76	111	139	185	199	228	296	308*1	**1621***1

DAEMON SNAPSHOT*

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	27	48	55	92	117	137	139	168	186	208	**1177**
16	37	55	91	115	148	168	208	208	250	290	**1570**
32	31	69	100	134	175	199	245	248	290	318	**1809**
64	35	71	103	148	195	199	253	292	373	382	**2051**
128	46	82	119	171	205	268	281	325	398	434	**2329**

SNAPSHOT

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	4	2*1	2	8	11	6	4	4	7	6	**54***1
16	3	6	6	11	18*1	15*1	16	13*2	10*1	15*1	**113***6
32	3	7*1	17	13	21	18	34*4	23	26*1	41	**203***6
64	5	13*1	16*3	27*3	27*3	24	43*3	45*3	61*5	62*1	**323***22
128	8*1	12*2	16*2	39*4	41*7	67*4	56*3	69*3	81*6	102*5	**491***37

DAEMON SNAPSHOT

Tx	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	Total
1	0	0	0	0	0	0	0	0	0	0	**0**
8	6	3	4	11	14	10	9	7	12	10	**86**
16	5	10	6	19	23	21	23	31	21	30	**189**
32	5	12	25	21	35	28	55	35	48	58	**322**
64	11	21	27	45	47	44	68	70	102	98	**533**
128	12	27	44	64	68	103	96	122	136	164	**836**

tidos para as instâncias do SQL Server, Firebird e Oracle são exibidos na Figura 48 considerando-se apenas os totais na tabelas apresentadas.

Nota-se que não há erros de serialização ou ocorrência de inconsistências quando as tarefas são executadas uma de cada vez, não importando qual seja o gerenciador ou o nível de isolamento em uso. À medida que o ambiente se torna competitivo e o número de transações simultâneas aumenta, a taxa de erros de concorrência e inconsistências também aumenta e, neste caso, além de DAEMON SNAPSHOT, o isolamento SERIALIZABLE no SQL Server é o único capaz de garantir apenas históricos serializáveis e manter a base de dados em um estado consistente sempre.

O número de inconsistências detectadas na base de dados quando se utiliza o isolamento READ COMMITTED é menor no SQL Server do que no Firebird e Oracle, porém

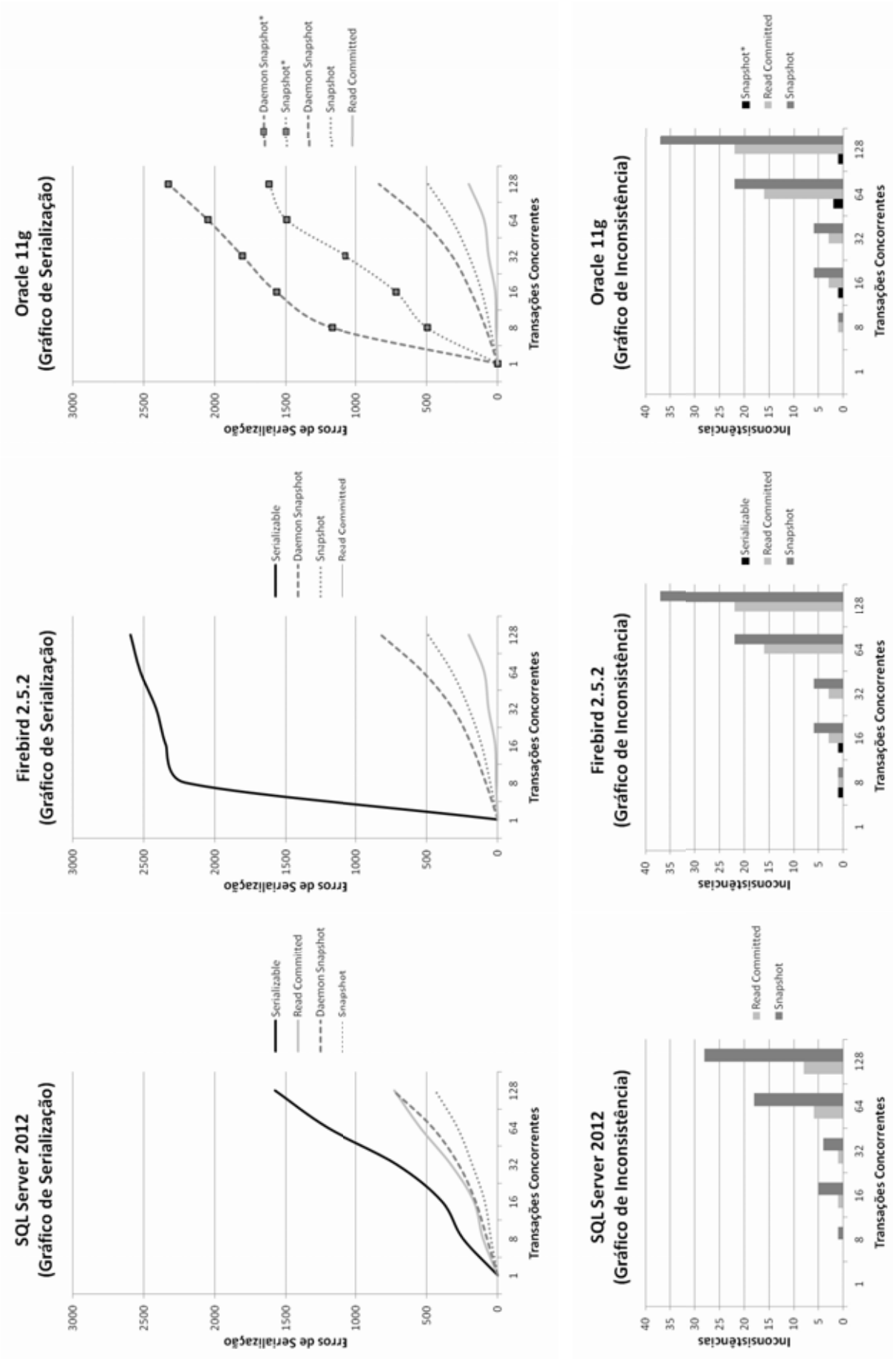

Figura 48 – Gráficos de serialização e inconsistência para instâncias do SQL Server, Firebird e Oracle.

ao custo de um número maior de erros de serialização no SQL Server. Isto se dá devido ao fato do SQL Server utilizar o protocolo 2PL, enquanto que Firebird e Oracle optam por não bloquear os elementos de dados que uma transação lê. Por outro lado, pelo menos considerando-se a carga de trabalho imposta pelo *benchmark* SmallBank++, a implementação do protocolo SNAPSHOT no SQL Server resultou em um número menor de erros de serialização e inconsistências na base de dados. Embora cada um dos sistemas de bancos de dados avaliados possuam sua própria infraestrutura transacional, os resultados obtidos pelo *benchmark* sobre Firebird e Oracle em modo de bloqueio de linha revelam que a política dos isolamentos READ COMMITTED e SNAPSHOT nesses sistemas são praticamente idênticas, especialmente considerando-se que os resultados obtidos para esses isolamentos foram os mesmos.

DAEMON SNAPSHOT pode ser implementado com base no isolamento SNAPSHOT já existente em um SGBD, com a vantagem de oferecer total garantia de consistência dos dados. De acordo com a Figura 48, DAEMON SNAPSHOT garante históricos serializáveis de forma mais eficaz que o próprio isolamento SERIALIZABLE no SQL Server, abortando um número menor transações concorrentes. De fato, o número de transações abortadas devido a erros de serialização quando se usa DAEMON SNAPSHOT ficou inclusive abaixo do isolamento READ COMMITTED no *benchmark* SmallBank++. No Firebird, uma implementação de DAEMON SNAPSHOT pode até mesmo superar seu nível mais forte de isolamento duas vezes, primeiro por abortar um número menor de transações e segundo por garantir apenas históricos serializáveis. É importante mencionar que o isolamento SERIALIZABLE no Firebird não oferece de fato garantia de serialização, como pode-se observar pelas inconsistências detectadas pelo *benchmark*. Por sua vez, Oracle que não oferece suporte a serialização pode implementar DAEMON SNAPSHOT fundamentando-se em seu isolamento SNAPSHOT já presente e, portanto, oferecer garantia de consistência dos dados.

Na Figura 49 é exibida a sobrecarga associada a uma implementação de DAEMON SNAPSHOT fundamentada no isolamento SNAPSHOT já presente em cada sistema de banco de dados avaliado, ou seja, o percentual exibido de erros de serialização representa o percentual de tarefas que foram abortadas pelos isolamentos SNAPSHOT e DAEMON SNAPSHOT. Conforme pode-se observar, um adicional de até seis pontos percentuais foi detectado no número de tarefas abortadas quando se utiliza o isolamento DAEMON SNAPSHOT, contudo obtendo-se garantia total de consistência dos dados. O percentual de tarefas abortadas é maior caso o modo de bloqueio por bloco seja utilizado no gerenciador Oracle, podendo chegar a treze pontos percentuais acima do número de tarefas abortadas quando se utiliza o isolamento SNAPSHOT.

Na Figura 50 são confrontados todos os níveis de isolamento que oferecem garantia de serialização no SQL Server, Firebird e Oracle. Enquanto o SQL Server apresenta a

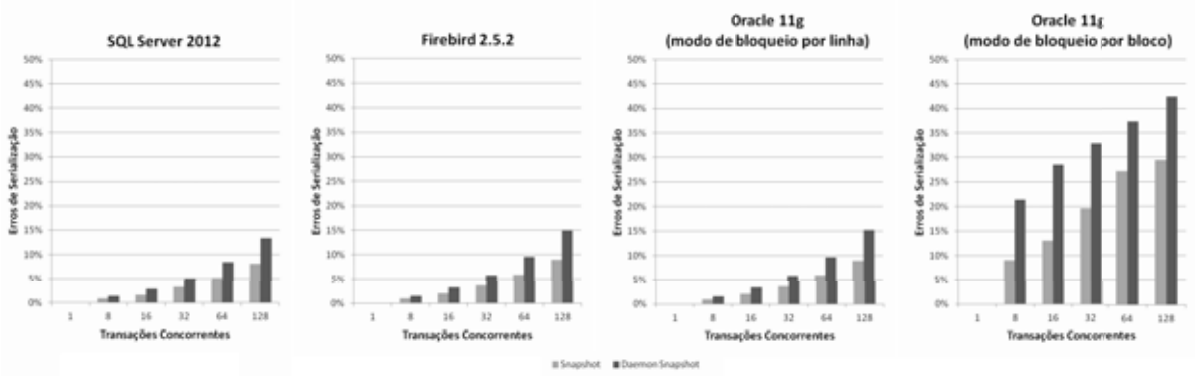

Figura 49 – Sobrecarga associada à obtenção de garantia de serialização considerando-se uma implementação de Daemon Snapshot fundamentada no próprio isolamento Snapshot.

melhor taxa de tarefas abortadas devido a erros de serialização, uma implementação de Daemon Snapshot no Firebird e Oracle em modo de bloqueio de linha (row-level) produz praticamente os mesmos resultados, os quais estão muito próximos dos obtidos no SQL Server.

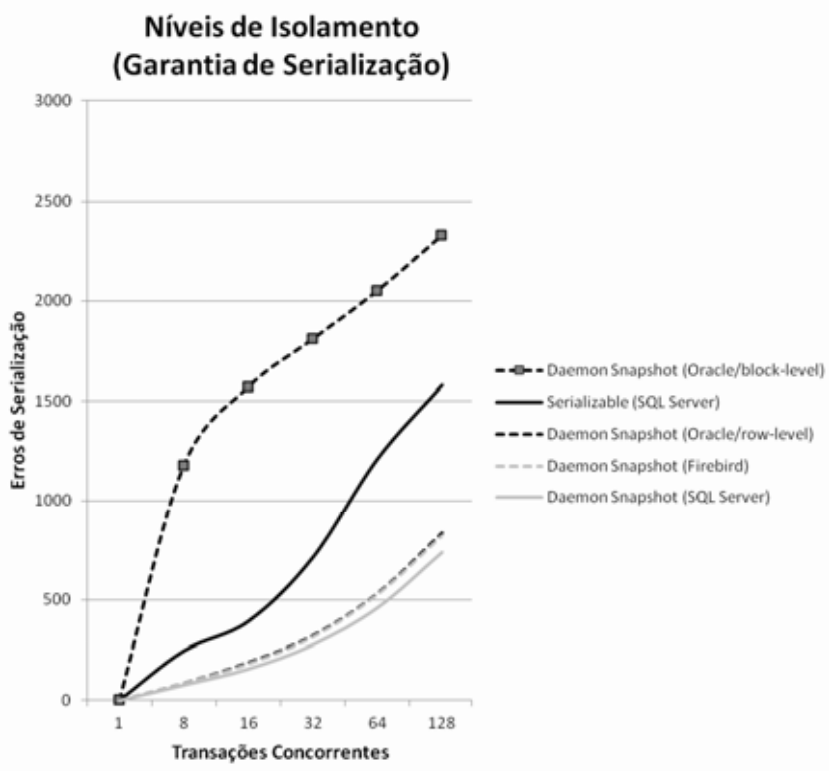

Figura 50 – Níveis de isolamento que garantem apenas históricos serializáveis no SQL Server, Firebird e Oracle. Os dados que representam o isolamento Daemon Snapshot em Oracle/row-level e Firebird são praticamente os mesmos.

4.2.2 Considerações

Conforme pode-se observar, a técnica de *daemons* também pode ser aplicada a gerenciadores que não implementam nativamente o isolamento Daemon Snapshot. No

caso dos sistemas de bancos de dados SQL Server, Firebird e Oracle, o isolamento Daemon Snapshot foi simulado utilizando-se o próprio isolamento Snapshot oferecido em conjunto com *triggers* especiais que desempenham o papel de cada entidade *daemon* ausente. Essas *triggers* foram denominadas neste trabalho de Daemon Triggers.

4.3 Considerações finais

A estratégia de *daemons* é independente de modelo de dados e pode ser aplicada também a sistemas de armazenamento na nuvem. Devido à sobrecarga de gerenciamento ocorrer apenas sobre os elementos de dados que são escritos pelas transações, Daemon Snapshot torna-se um candidato em potencial para ser aplicado em sistemas distribuídos onde algumas tarefas transacionais devem ter garantia de consistência dos dados.

Devido à natureza experimental do protótipo construído para o gerenciamento de dados complexos, a versão do *benchmark* para Draco difere da versão para o modelo relacional principalmente em razão dos SGBDs relacionais avaliados serem sistemas de produção e da necessidade de se confrontar a técnica de *daemons* com os níveis de isolamento disponíveis nesses sistemas. Em razão disso, a política de testes para o modelo relacional foi concebida de modo que anomalias ocorressem não apenas no isolamento Snapshot mas também nos outros níveis de isolamento no intuito de se identificar quais níveis são capazes de oferecer real garantia de serialização.

Conclusão

Em um ambiente transacional onde apenas históricos serializáveis são permitidos, todas as ações conflitantes devem ser impedidas e, consequentemente, o nível de concorrência é reduzido em função de um número maior de transações que são abortadas. Por esse motivo, tipicamente os sistemas transacionais oferecem às aplicações, por meio do conceito de nível de isolamento, a possibilidade de escolha entre garantia de consistência ou uma maior concorrência.

A vantagem de se executar uma transação sob um isolamento que não garante consistência dos dados está, portanto, diretamente relacionada ao fato deste isolamento permitir um número maior de transações concorrentes. Contudo, se a concorrência obtida por tal nível de isolamento for a mesma, ou pelo menos muito próxima, da concorrência obtida quando se utiliza um isolamento mais forte, então não há razão para se executar uma transação sob um isolamento que ofereça menor garantia de consistência.

Dentre os diversos tipos de isolamento existentes, Snapshot se destaca pelo fato de permitir uma alta concorrência e, ao mesmo tempo, oferecer uma garantia de consistência próxima ao nível de consistência máximo. De modo a tirar proveito das características inerentes ao isolamento Snapshot e no intuito de oferecer um nível de isolamento que permita apenas históricos serializáveis, foi proposto neste trabalho um novo tipo de isolamento, denominado Daemon Snapshot. O mecanismo de *daemons* utilizado foi baseado nas especificações de isolamento segundo a norma ANSI revisada. As especificações portáveis não foram consideradas pois estas exigem o gerenciamento não apenas dos elementos que a transação escreve mas também dos elementos que esta lê. O objetivo foi, portanto, conceber uma estratégia que permita obter garantia de consistência tirando proveito dos benefícios intrínsecos ao isolamento Snapshot, principalmente em relação à sobrecarga imposta somente aos elementos que são escritos pela transação.

Em razão da sobrecarga de gerenciamento ocorrer somente sobre os elementos que uma transação escreve, Daemon Snapshot torna-se um candidato em potencial para ser aplicado em ambientes distribuídos onde algumas tarefas transacionais exijam garantia de consistência dos dados.

Os testes comprovaram que o custo associado à obtenção de garantia de consistência por meio de *daemons* é baixo quando comparado ao uso do isolamento Snapshot em sua forma natural. A infraestrutura sugerida e construída para o SGBD permitiu avaliar a sobrecarga imposta durante operações de I/O e a taxa de erros de serialização ob-

tida quando as transações executam sob o isolamento SNAPSHOT e DAEMON SNAPSHOT. Segundo o sistema de *benchmark* que foi projetado para desencadear a ocorrência do fenômeno Escrita Oblíqua, a serialização é obtida a um custo adicional de aproximadamente 1,5 ponto percentual sobre o número de transações abortadas quando as mesmas são comparadas a uma execução baseada apenas em SNAPSHOT, a qual não oferece garantia de consistência dos dados. A versão relacional do *benchmark* construído mostrou que a garantia de consistência dos dados pode ser obtida com DAEMON SNAPSHOT a um custo adicional de até seis pontos percentuais.

Conclui-se portanto que o isolamento DAEMON SNAPSHOT é capaz de oferecer garantia de serialização a um custo relativamente baixo ao mesmo tempo em que impõe sobrecarga de gerenciamento apenas sobre os elementos que são escritos pelas transações. Conforme foi demonstrado no capítulo anterior, DAEMON SNAPSHOT garante históricos serializáveis no SQL Server de forma mais eficaz que o próprio isolamento SERIALIZABLE. No caso dos gerenciadores Firebird e Oracle, DAEMON SNAPSHOT se mostrou ser não apenas uma solução eficaz como também a única possível em virtude desses gerenciadores não oferecerem garantia de serialização.

Contribuições

O trabalho descrito gerou as seguintes contribuições:

1. Artigo "A Logical Approach to the Management of Object Identifiers in Non-conventional Database Management Systems", conferência PDCAT[1] (VALêNCIO et al., 2011).

2. Artigo "Managing Object Identity with Less I/O Operations", conferência PDCAT (ALMEIDA et al., 2012).

3. Relatório técnico "NuGeM – Núcleo Gerenciador de dados Multimídia" (VALêNCIO; ALMEIDA, 2013a).

4. Relatório técnico "NuGeM++ – Otimizações no Núcleo Gerenciador de dados Multimídia" (VALêNCIO; ALMEIDA, 2013b).

5. Artigo "The Storage System for a Multimedia Data Manager Kernel", conferência PDCAT (VALêNCIO et al., 2013).

6. Artigo "Ensuring consistency under the Snapshot isolation" enviado à conferência ADC[2] 2014 e aguardando notificação de aceite.

[1] PDCAT — Parallel and Distributed Computing, Applications and Technologies
[2] ADC — Australasian Database Conference

7. Artigo “A serialization technique based on the Snapshot isolation protocol” a ser submetido para uma revista.

Subprodutos

No intuito de demonstrar a aplicabilidade do uso de *daemons* e a viabilidade do isolamento Daemon Snapshot, os seguintes subprodutos foram gerados:

1. Draco — Protótipo de um SGBD cuja infraestrutura possui características temporais e é voltada ao gerenciamento de dados complexos. Draco permite a definição de *daemons* e implementa os isolamentos Snapshot e Daemon Snapshot.
2. SmallBank++ — Sistema de *benchmark* capaz de conduzir a base de dados a um estado inconsistente quando as transações estão sob o isolamento Snapshot.
3. `NuGeM++` — Versão otimizada do Núcleo Gerenciador de Dados Multimídia, `NuGeM`. O projeto do mapa de OIDs de Draco foi inspirado no gerenciamento de páginas *shadow* e na infraestrutura de gerenciamento de registros reais do projeto `NuGeM++`.

Trabalhos futuros

A técnica proposta e demonstrada neste trabalho foi aplicada no gerenciamento de dados segundo os modelos orientado a objetos e relacional, mas pode também ser utilizada por sistemas de armazenamento na nuvem. Nesse sentido é preciso investigar formas adequadas de se implementar o conceito de *daemons* em modelos de dados como colunar, orientado a documentos, orientado a grafos e chave-valor.

Referências

ADYA, A. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. Tese (Doutorado) — Massachusetts Institute of Technology, 1999. Citado em 11 páginas: 29, 32, 42, 46, 49, 54, 56, 59, 60, 63 e 71.

ADYA, A.; LISKOV, B.; O'NEIL, P. Generalized isolation level definitions. In: IEEE. *Data Engineering, 2000. Proceedings. 16th International Conference on*. [S.l.], 2000. p. 67–78. Citado em 11 páginas: 29, 30, 31, 32, 42, 43, 44, 46, 49, 51 e 59.

AGUIAR, L. *7 dragões famosos na cultura pop – Superlistas*. 2011. Disponível em: <http://super.abril.com.br/blogs/superlistas/7-dragoes-famosos-na-cultura-pop>. Citado na página 81.

ALMEIDA, F. R. d.; VALêNCIO, C. R.; FERRIZZI, A. C.; TRONCO, M. L.; SOUZA, R. C. G. a. d. Managing object identity with less i/o operations. In: *Proceedings of the 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*. Washington, DC, USA: IEEE Computer Society, 2012. (PDCAT '12), p. 213–218. ISBN 978-0-7695-4879-1. Disponível em: <http://dx.doi.org/10.1109/PDCAT.2012.91>. Citado em 3 páginas: 27, 83 e 120.

ALOMARI, M. *Ensuring Serializable Executions with Snapshot Isolation DBMS*. Tese (Doutorado) — University of Sydney, 2008. Citado em 3 páginas: 26, 62 e 71.

ALOMARI, M.; CAHILL, M.; FEKETE, A.; ROHM, U. The cost of serializability on platforms that use snapshot isolation. In: IEEE. *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. [S.l.], 2008. p. 576–585. Citado na página 99.

ALOMARI, M.; CAHILL, M.; FEKETE, A.; RÖHM, U. Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels? In: SPRINGER. *Database Systems for Advanced Applications*. [S.l.], 2008. p. 267–281. Citado na página 61.

ALOMARI, M.; FEKETE, A.; ROHM, U. A robust technique to ensure serializable executions with snapshot isolation dbms. In: IEEE. *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. [S.l.], 2009. p. 341–352. Citado em 2 páginas: 26 e 62.

ANSI. *ANSI X3.135-1992, American National Standard for Information Systems — Database Language — SQL*. [S.l.], 1992. Citado em 9 páginas: 29, 33, 34, 40, 50, 51, 53, 56 e 59.

BERENSON, H.; BERNSTEIN, P.; GRAY, J.; MELTON, J.; O'NEIL, E.; O'NEIL, P. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, ACM, v. 24, n. 2, p. 1–10, 1995. Citado em 19 páginas: 29, 30, 31, 32, 34, 40, 41, 42, 50, 51, 52, 53, 54, 55, 56, 59, 74, 79 e 80.

BERLER, M.; EASTMAN, J.; JORDAN, D.; RUSSELL, C.; SCHADOW, O.; STANIENDA, T.; VELEZ, F. *The object data standard: ODMG 3.0*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. ISBN 1-55860-647-5. Citado na página 76.

BERNSTEIN, P. A.; GOODMAN, N. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, ACM, v. 13, n. 2, p. 185–221, 1981. Citado na página 32.

CAHILL, M. J. *Serializable Isolation for Snapshot Databases*. Tese (Doutorado) — The University of Sydney, 2009. Citado em 6 páginas: 26, 60, 63, 65, 70 e 71.

CAHILL, M. J.; RöHM, U.; FEKETE, A. D. Serializable isolation for snapshot databases. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008. (SIGMOD '08), p. 729–738. ISBN 978-1-60558-102-6. Disponível em: <http://doi.acm.org/10.1145/1376616.1376690>. Citado em 3 páginas: 26, 63 e 70.

CAHILL, M. J.; RöHM, U.; FEKETE, A. D. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 34, n. 4, p. 20:1–20:42, dez. 2009. ISSN 0362-5915. Disponível em: <http://doi.acm.org/10.1145/1620585.1620587>. Citado em 5 páginas: 26, 29, 30, 63 e 71.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to algorithms*. 3. ed. [S.l.]: Massachusetts Institute of Technology, 2009. Citado na página 68.

FEKETE, A.; LIAROKAPIS, D.; O'NEIL, E.; O'NEIL, P.; SHASHA, D. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, ACM, v. 30, n. 2, p. 492–528, 2005. Citado em 9 páginas: 26, 52, 54, 60, 61, 62, 63, 65 e 99.

FEKETE, A.; O'NEIL, E.; O'NEIL, P. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, ACM, v. 33, n. 3, p. 12–14, 2004. Citado em 5 páginas: 56, 74, 76, 78 e 99.

FERRIZZI, A. C. *Uma abordagem lógica para o gerenciamento de identificadores de objetos em sistemas gerenciadores de banco de dados não convencionais*. 88 f. Dissertação (Mestrado em Ciência da Computação — área de Banco de Dados) — Instituto de Biociências, Letras e Ciências Exatas, Universidade Estadual Paulista Júlio de Mesquita Filho, São José do Rio Preto, São Paulo, 2010. Citado na página 83.

FIREBIRD. *Firebird 2.5.2 Superserver instance*. 2013. Disponível em: <http://www-.firebirdsql.org>. Citado na página 112.

GRAY, J.; REUTER, A. *Transaction Processing: Concepts and Techniques*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. 1070 p. ISBN 1-55860-190-2. Citado em 4 páginas: 29, 30, 65 e 71.

GRAY, J. N.; LORIE, R. A.; PUTZOLU, G. R.; TRAIGER, I. L. *Granularity of locks and degrees of consistency in a shared data base*. [S.l.]: IBM Research Division, 1976. Citado em 9 páginas: 15, 29, 32, 33, 40, 50, 51, 56 e 59.

JORWEKAR, S.; FEKETE, A.; RAMAMRITHAM, K.; SUDARSHAN, S. Automating the detection of snapshot isolation anomalies. In: VLDB ENDOWMENT. *Proceedings of the 33rd international conference on Very large data bases*. [S.l.], 2007. p. 1263–1274. Citado em 4 páginas: 26, 61, 62 e 71.

KUNG, H.-T.; ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, ACM, v. 6, n. 2, p. 213–226, 1981. Citado na página 32.

LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. [S.l.]: Addison Wesley Professional, 2004. ISBN 0-13-148906-2. Citado na página 73.

LIU, L.; ÖZSU, M. T. *Encyclopedia of database systems*. 1. ed. [S.l.]: Springer, 2009. Citado em 2 páginas: 29 e 30.

MICROSOFT. *Microsoft SQL Server 2012 Express Edition*. 2013. Disponível em: <http://www.microsoft.com/en-us/sqlserver/editions/2012-editions/express.aspx>. Citado na página 112.

MICROSOFT. *Visual Studio Professional 2012*. 2013. Disponível em: <http://www-.microsoft.com/visualstudio/eng/products/visual-studio-professional-2012>. Citado na página 109.

NASSU, E. A.; SETZER, V. W. *Bancos de dados orientados a objetos*. 1. ed. [S.l.]: Editora Edgard Blücher Ltda., 1999. ISBN 85-212-0171-0. Citado na página 25.

OMG. *Object Management Group – UML*. 2013. Disponível em: <http://www.uml.org>. Citado na página 73.

ORACLE. *Oracle Database 11g Express Edition*. 2013. Disponível em: <http://www-.oracle.com/technetwork/products/express-edition/overview/index.html>. Citado na página 113.

ORACLE. *Oracle Database Advanced Replication — 11g Release 2*. 2013. Disponível em: <http://docs.oracle.com/cd/E11882_01/server.112/e10706/repmaster.htm>. Citado na página 113.

PORTS, D. R.; GRITTNER, K. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 5, n. 12, p. 1850–1861, 2012. Citado em 4 páginas: 69, 70, 71 e 113.

REVILAK, S.; O'NEIL, P.; O'NEIL, E. Precisely serializable snapshot isolation (pssi). In: IEEE. *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. [S.l.], 2011. p. 482–493. Citado em 3 páginas: 26, 65 e 70.

REVILAK, S. A. *Precisely Serializable Snapshot Isolation*. Tese (Doutorado) — Office of Graduate Studies, University of Massachusetts Boston, 2011. Citado em 3 páginas: 26, 65 e 99.

TECHNOLOGICA. *Glossários – Termos Técnicos de Informática*. 2013. Disponível em: <http://www.technologica.inf.br/glossario/exibe.asp>. Citado na página 73.

TPC. *Transaction processing performance council*. 2013. Disponível em: <http://www-.tpc.org>. Citado na página 99.

VALDERRAMA, C. *IB Experts: The database experts — Transaction options explained*. 2013. Disponível em: <http://ibexpert.net/ibe/index.php?n=Doc-.TransactionOptionsExplained>. Citado na página 112.

VALêNCIO, C. R. *Núcleo Gerenciador de Objetos - Compatibilizando Eficiência e Flexibilidade*. 149 f. Tese (Doutorado em Ciências: Física Aplicada — opção Física Computacional) — Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2000. Citado em 2 páginas: 83 e 88.

VALêNCIO, C. R.; ALMEIDA, F. R. d.; MACHADO, J. M.; COLOMBINI, A. C.; NEVES, L. A.; SOUZA, R. C. G. a. d. The storage system for a multimedia data manager kernel. In: *Proceedings of the 2013 14th International Conference on Parallel and Distributed Computing, Applications and Technologies*. [S.l.]: IEEE Computer Society, 2013. (PDCAT '13). Citado em 2 páginas: 27 e 120.

VALêNCIO, C. R.; ALMEIDA, F. R. de. *NuGeM – Núcleo Gerenciador de dados Multimídia*. [S.l.], 2013. 66 p. Relatório Técnico, ISBN 978-85-8224-052-6. Citado em 3 páginas: 27, 88 e 120.

VALêNCIO, C. R.; ALMEIDA, F. R. de. *NuGeM++ – Otimizações no Núcleo Gerenciador de dados Multimídia*. [S.l.], 2013. 120 p. Relatório Técnico, ISBN 978-85-8224-051-9. Citado em 3 páginas: 27, 88 e 120.

VALêNCIO, C. R.; FERRIZZI, A. C.; ALMEIDA, F. R. d.; CARREIRA, J. A.; TRONCO, M. L. A logical approach to the management of object identifiers in non-conventional database management systems. In: *Proceedings of the 2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*. Washington, DC, USA: IEEE Computer Society, 2011. (PDCAT '11), p. 293–298. ISBN 978-0-7695-4564-6. Disponível em: <http://dx.doi.org/10.1109/PDCAT-.2011.6>. Citado em 3 páginas: 27, 83 e 120.

WHEELER. *Security Taxonomy and Glossary – Anne & Lynn Wheeler*. 2013. Disponível em: <http://www.garlic.com/~lynn/secure.htm>. Citado na página 73.

YABANDEH, M.; FERRO, D. G. A critique of snapshot isolation. In: ACM. *Proceedings of the 7th ACM european conference on Computer Systems*. [S.l.], 2012. p. 155–168. Citado em 16 páginas: 26, 29, 30, 31, 32, 41, 42, 51, 52, 53, 54, 55, 59, 60, 68 e 70.