UNIVERSIDADE DE CAXIAS DO SUL
Centro de Computação e Tecnologia da Informação
Curso de Bacharelado em Ciência da Computação

Fábio Hentz Lunkes

API PARA PERSISTÊNCIA TRANSACIONAL DE OBJETOS

Caxias do Sul

2010

Fábio Hentz Lunkes

API PARA PERSISTÊNCIA TRANSACIONAL DE OBJETOS

Trabalho de Conclusão de Curso para obtenção do Grau de Bacharel em Ciência da Computação da Universidade de Caxias do Sul.

Helena Graziottin Ribeiro

Caxias do Sul

2010

RESUMO

Atualmente o padrão de programação orientada a objetos vem sendo muito aplicado ao desenvolvimento de software para diversos tipos de aplicações. As aplicações desenvolvidas que empregam armazenamento de dados em sistemas relacionais e implementam seu software com um padrão de programação orientada a objetos, devem possuir subsistemas para converter as operações entre os dois padrões. Neste trabalho, será proposto a implementação de uma API, interface de programação, que oferece recursos para armazenamento de objetos transacionalmente, realizando o processamento do lado do servidor. Neste trabalho foi utilizado a linguagem de programação Java.

Palavras-chave: API, Persistência, Objetos, Transacional, DBM, *key-value, Database Engine.*

ABSTRACT

Currently the object oriented design standard has been widely applied to the development of software for various types of applications. Applications developed employing data storage relational systems and implement your software with a standard object oriented programming, should have subsystems for converting between the two operations standards. In this paper, we proposed the implementation of an API programming interface, which provides resources for storing objects transactionally, performing processing on the server side. This work used the Java programming language.

Keywords: API, Persistence, Object, Transaction, DBM, key-value, Database Engine.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Sigla	Significado em Português	Significado em Inglês
SGBD	Sistema Gerente de Banco de Dados	Database Management System
DBM	Sistema Gente de Dados	Database Managment
LRU	Usado a Mais Tempo ou Menos Recente	Least Recently Used
SQL	Linguagem de Consulta Estruturada	Structured Query Language
ISO	Organização Internacional para Padronização	International Organization for Standardization
ANSI	Instituto Nacional Americano de Padronização	American National Standards Institute
C++	Linguagem de Programação	Programing Language
API	Interface de Programação de Aplicativo	Application Programming Interface
ACID	Atomicidade, Consistência, Isolamento e Durabilidade	Atomicity, Consistency, Isolation and Durability
CKPT	Ponto de Verificação	Checkpoint
SGBDOO	Sistema Gerenciador de Banco de Dados Orientado a Objetos	Oriented Object Database Management System
SGBDR	Sistema Gerenciador de Banco de Dados Relacional.	Relational Database Management System
BSD	Sistema Operacional Unix-*Like*	Berkeley Software Distribution
UML	Linguagem de Modelagem Unificada	Unified Modeling Language
TCP/IP	Protocolo de Controle de Transmissão e Protocolo de Interconexão	Transmission Control Protocol and Internet Protocol.

SUMÁRIO

1 INTRODUÇÃO

Os fundamentos da tecnologia de banco de dados relacionais surgiram inicialmente dentro da empresa norte americana IBM, nas décadas de 1960 e 1970, motivadas por pesquisa para automação de escritório. Neste período foi descoberto que estava sendo muito caro empregar pessoas para armazenar e organizar (indexar) arquivos. Em função disso, seria importante investir em pesquisa para encontrar um meio mecânico de realizar estas tarefas tornando-as mais eficientes e econômicas.

Muitas das tecnologias utilizadas até hoje tem sua origem nesse período, em especial, podemos citar os modelo modelos hierárquico, de rede e relacional.

Em 1970 o pesquisado da IBM, Ted Codd, publicou o primeiro artigo sobre bancos de dados relacionais. Este artigo tratava sobre o uso de cálculo e álgebra relacional para realizar operações de armazenamento e recuperação de dados por usuários não técnicos. Em seu modelo, Codd, visualizava que, as operações seriam realizadas por comandos em inglês e os dados estariam armazenados em tabelas. Entretanto, devido a natureza técnica de seu artigo e sua base matemática, o artigo não teve seu entendimento e proposição realizados de forma completa (SILBERSCHATZ, ABRAHAM; 2008, p. 109).

Mesmo assim, Codd conseguiu iniciar um projeto dentro da IBM, chamado System R(Sistema R), que no futuro poderia se tornar um produto. O Sistema R foi composto pela criação de um sistema de banco de dados relacional, que evoluiu para o um novo sistema chamado SQL/DS e por fim para o sistema DB2. Neste sistema de banco de dados o grupo de pesquisa criou a linguagem SQL(Linguagem de Consulta Estruturada, *Structured Query Language*), servindo de interface para realizar as operações com o sistema de banco de dados relacional. Esta linguagem tornou-se um padrão industrial para realizar as operações com os sistemas de banco de dados, sendo hoje, padronizada pela ISO (International Organization for Standardization) e pela a ANSI (American National Standards Institute).

Sistemas de banco de dados tem grande número de aplicações em diversas atividades de nossa sociedade, empresas, escolhas, lojas entre outros podem depositar neles todo

conteúdo de informações, que anteriormente, como foi mencionado no inicio, eram em arquivos de papel. Apesar disso em meados da década de 1980, ficou claro, que várias áreas não eram aplicáveis os sistemas de bancos de dados relacionais. Entre algumas dessas áreas podemos citar medicina, multimídia e física de energia elevada. O que estas áreas tinham em comum era a necessitar de criar suas próprias representações de dados.

Em função da necessidade do usuário de criar suas próprias representações de dados e formas de busca, surge a necessidade de desenvolverem pesquisas em torno de sistema de banco de dados orientados a objetos. Juntamente neste período, começam a surgir as primeiras linguagens orientadas a objetos, como por exemplo, o C++. Antes do C++ a linguagem Simula 67 foi precursora na implementação deste conceito(SILBERSCHATZ, ABRAHAM; 2008, p. 250).

Um dos primeiros bancos de dados orientados a objetos foi da Objectivity. Hoje existem vários sistemas, porém, este modelo de banco de dados não prevaleceu como maioria no mercado de software, porém, as linguagens de programação orientadas a objetos evoluíram muito e são muito utilizadas hoje, sendo um paradigma de programação importante utilizado pela indústria de software.

As linguagens de programação orientadas a objetos tem muitas aplicações e quando existe a necessidade de realizar a persistência de dados, ainda na maioria dos casos, se faz uso dos bancos de dados relacionais. Para estes casos, existe a necessidade de um conjunto de sistemas que realizem a conversão entre as operações geradas pelos objetos e operações sobre as bases de dados relacionais. A necessidade de conversão entre estas operações é justificada pela diferença entre os dois modelos, orientado a objetos e relacional. Esta conversão traz muitos custos como, performace, gerência de código para tratar as operações, adaptação do sistema e custos de implementação, já que, exige pessoas com conhecimento nestes sistemas.

Estes custos podem tornar interessante a utilização de APIs para Persistência Transacional de objetos, que ofereça simplicidade, portabilidade, perfomace e escalabilidade tendo aplicações em sistemas computacionais para computadores pessoais, dispositivos móveis, aplicações embarcadas e grandes sistemas para grandes quantidades de dados. APIs para persistência de objetos, padrão key-value, podem ser consideradas como uma opção para armazenamento de dados na computação distribuída, como *cloud-computing*.

2 TRANSAÇÕES CONCORRENTES

Definimos uma transação como sendo uma “unidade de programa que acessa, e possivelmente, atualiza vários itens de dado” (SILEBERSCHATZ, 1999, p. 441). Essa unidade lógica, representa do ponto de vista do usuário, uma única operação.

Para qualquer operação de modificação ou leitura da base de dados será necessário uma transação. As transações concorrentes serão formadas por um conjunto de requisições a base de dados acessando concorrentemente um recurso compartilhado. Este recurso compartilhado, será formando por um conjunto de subsistemas como o sistema de armazenamento, sistema operacional e sistema de arquivos dos dados.

2.1 PROPRIEDADES DAS TRANSAÇÕES

Assim como no processamento de transações concorrentes, a transação será um elemento fundamental que todos os módulos do sistema de armazenamento vão processar. Por isso vamos definir quais serão as propriedades e comportamentos de uma transação. Qualquer operação sobre a base de dados será sempre realizada por meio de uma transação. As operações serão solicitadas pelo usuário e são ações de sua necessidade, como por exemplo, ler, adicionar, apagar ou alterar informações da base de dados.

O sistema de armazenamento possui vários módulos que serão abordados durante o texto. Cada módulo irá contribuir de alguma forma para que as propriedades das transações sejam implementadas, pois, elas vão definir no final as propriedades do sistema como um todo. Uma transação de banco de dados deve possuir as propriedades ACID – Atomicidade, Consistência, Isolamento e Durabilidade. Estas propriedades de uma transação, vão garantir o funcionamento da transação conforme a conceituamos. Abaixo descrevemos com detalhes cada propriedade:

1. Atomicidade: Capacidade que a transação deve possuir para executar um conjunto de operações que atualizam a base de dados como se fossem uma única unidade lógica, ou seja, estas operações devem todas serem executadas por completo ou nada será

executado. Na formação da unidade, sempre deverá conter a conclusão de todas as operações de atualização da base de dados ou conter nenhuma operação de atualização da base. Dessa forma, busca-se evitar atualizações parciais da base de dados, que seria uma fragmentação da unidade.

2. Consistência: Tornar permanente a transição de um estado consistente para outro no banco de dados depois de concluir a transação.
3. Isolamento: Assegurar que o estado obtido pela execução de transações concorrentes, sejam o mesmo que seria obtido caso elas fossem executadas serialmente uma após a outra.
4. Durabilidade: Garantir que, se uma transação tenha sido concluída com sucesso, as atualizações na base de dados sejam permanentes. Caso houver falha antes da transação ser concluída com sucesso, o componente gerente de recuperação deverá restaurar o banco de dados para o último estado consistente. A durabilidade poderá ser obtida se as atualizações do banco de dados foram gravados em disco antes da transação terminar e se as informações gravadas em disco sobre as atualizações realizadas, são suficientes para restaurar o banco de dados quando o sistema for iniciado novamente.

Conforme evidenciamos, uma transação com propriedades ACID, podem terminar com sucesso ou falhar. Uma transação que chegou a efetivar as modificações com sucesso, é uma transação efetivada (*commited*). Quando ocorre um erro e as modificações no banco de dados devem ser desfeitas, dizemos que a transação foi cancelada (*rolled back*). Cada transação possui um estado durante seu tempo de execução. Este estado vai identificar a situação na qual a transação se encontra dando condição ao processamento delas no sistema. São os seguintes estados que uma transação possui:

1. Ativa: Estado inicial da transação, vai permanecer neste estado enquanto estiver executando.

2. Em efetivação parcial: Estado após a execução da última operação.

3. Em falha: Após descoberto que a execução normal não poderá ser realizada.

4. Abortada: Depois que a transação foi desfeita, e o estado do banco de dados foi restabelecido ao estado anterior ao da execução da transação.

5. Em efetivação: Após a conclusão com sucesso da transação.

Na Figura 1, temos um diagrama de estado da transação representando os estados e suas transições.

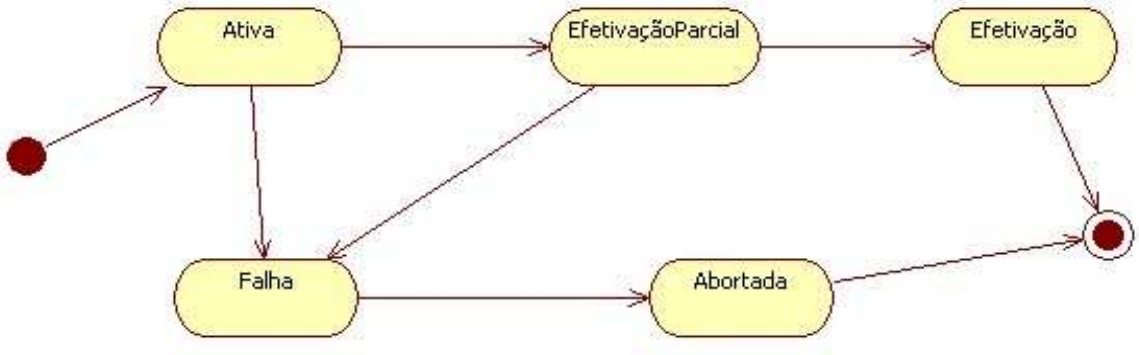

Figura 1: Estados de uma transação (SILBERSCHATZ, ABRAHAM; 1999, p. 109)

2.2 ARQUITETURA DE UM SISTEMA PARA PROCESSAMENTO DE TRANSAÇÕES CONCORRENTES

De forma genérica e simples, podemos dividir e descrever a arquitetura de um sistema de armazenamento, através dos seus módulos e interações que existem entre eles. Na Figura 2, temos uma simplificação dos módulos de um sistema.

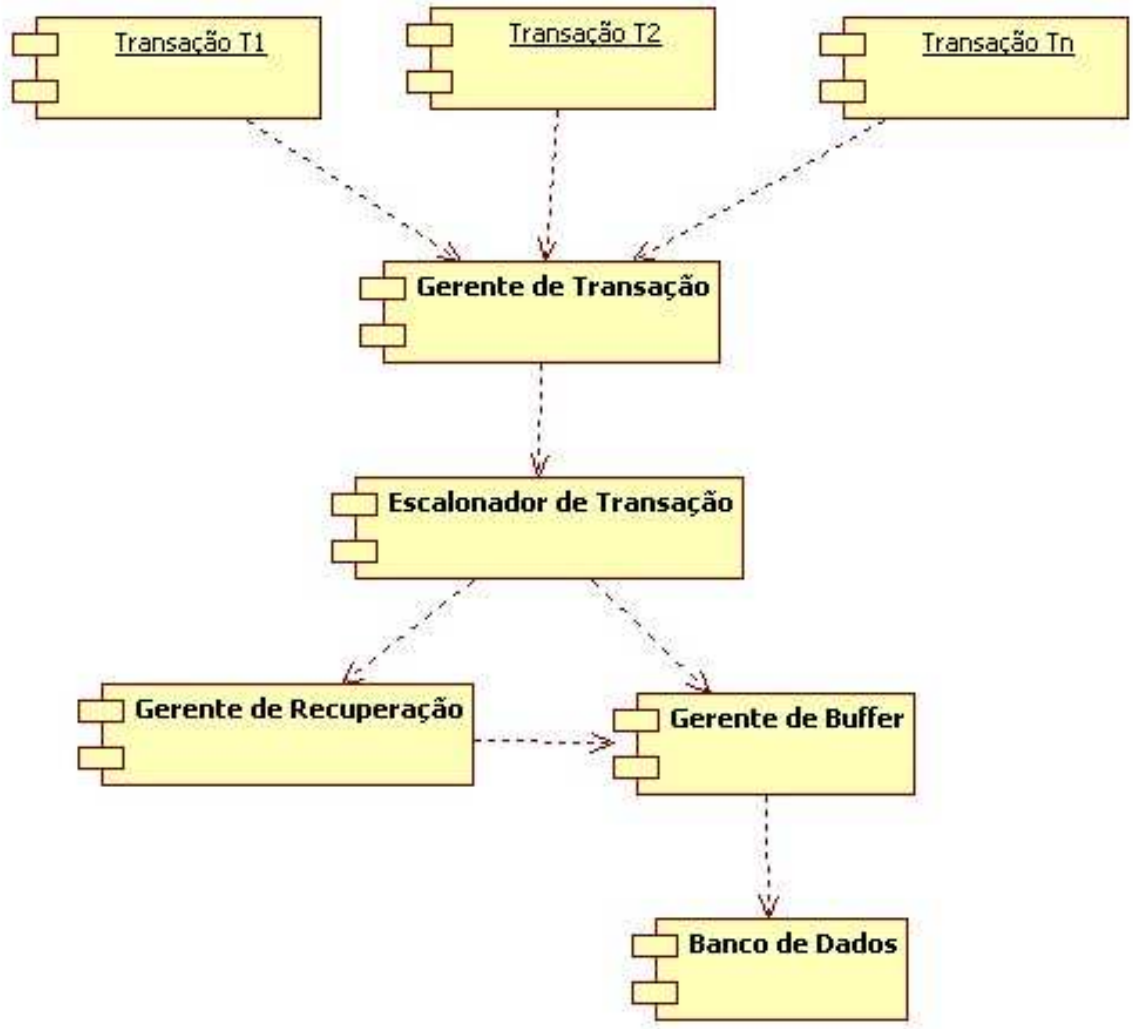

Figura 2:Arquitetura de um DBM (BERNSTEIN, PHILIP; 1987, p.18)

As transações são os principais objetos do sistema, são como usuários solicitando serviços para o sistema gerente de banco de dados. O escalonador irá definir escalas válidas, verificando ordens válidas para execução das operações das transações. Ele tem este trabalho para haver o máximo de aproveitamento do processador, e com isso, fazer muito processamento em paralelo com outros processos.

O Gerente de *Buffer*, tem como função *bufferizar* os dados solicitados. É ele que vai carregar os dados dos arquivos do banco de dados e do arquivo de log. O conceito que o gerente de *buffer* utiliza é o mesmo que os sistemas operacionais implementam com a memória virtual.

E por fim, o Gerente de Recuperação, que tem por função recuperar o banco de dados para um estado consistente antes da ocorrência de uma falha.

As estruturas de dados que serão manipuladas em memória principal pelo Gerente de Transação e Escalonador de Transação, precisam ter o mínimo de complexidade de tempo para poderem dar uma resposta rápida as transações. O mesmo ocorre com as estruturas de dados aplicadas em arquivo, que no caso, serão tratadas pelo Gerente de *Buffer* e Gerente de Recuperação. Os dados em um arquivo lidos do disco rígido precisam ser manipulados em unidades chamadas de bloco, pois, a transferência de dados entre o disco e a memória principal também se dará em bloco com tamanho definido no sistema operacional. Para este tipo de necessidade, a estrutura de dados em arvore B tem um comportamento desejado, pois, minimiza a complexidade de tempo e espaço para leitura dos blocos. A árvore B é uma estrutura de dados que pode representar a disposição e indexação dos dados do banco de dados.

2.3 ORGANIZAÇÃO DOS ARQUIVOS

Na arquitetura de um sistema de banco de dados podemos dividi-lo em dois grades grupos, um para tratar as estruturas de dados da memória principal e outro para tratar das estruturas de dados da memória secundária. Um estrutura de dados muito utilizada para memória secundária é a árvore B+. A árvore B+ é aplicada para indexar dados em um arquivo. Ela é semelhante a organização de arquivos em multinivel, porém, pode possuir tantos níveis quantos for necessário. Um árvore B+ é formada por nodos ou páginas, que possuem um conjunto de chaves e um conjunto de ponteiros para nodos filhos. As chaves, em um página, são organizadas em ordem crescente e um parâmetro n, indica o número máximo de chaves que um nodo pode possuir. Se uma árvore B+ possui n chaves, deverá possuir n+1 ponteiros para filhos. Abaixo descrevemos as regras para construir uma árvore B+.

1. Uma árvore B sempre vai possuir um nodo raiz. Este nodo, é o primeiro nodo a ser visitado para percorrer a árvore. Uma raiz sempre poderá possuir no mínimo uma chave e dois ponteiros para os nodos do nível abaixo.
2. Em uma folha, que são os nodos do ultimo nível, o ultimo ponteiro aponta para o nodo com as próximas chaves mais altas. Esse ponteiro facilita o processamento sequêncial

em pesquisas delimitas por intervalos de valor da chave. Os outros n ponteiros, devem ser ocupados por uma quantidade mínima definida pelo maior inteiro de (n + 1) / 2 apontando para blocos de dados que contém a chave relacionada com este ponteiro.

3. Nos nodos internos, que ficam entre a raiz e os nodos folha, as chaves, estarão como sempre, ordenadas em ordem crescente, tendo no máximo n chaves e consequentemente n+1 ponteiro para os filhos do nível abaixo. O número mínimo de ponteiros filhos, será o inteiro maior de (n+1)/2. Para um i-ésima chave Ki, existe um ponteiro Pi que aponta para uma subárvore que contém apenas nodos com chaves menores de Ki, sendo que i, contém seus valores entre 1..n. Para o ponteiro P(n+1), ou seja, o último ponteiro, as chaves são as maiores chaves de todo o nodo.

Na Figura 3, representamos a estrutura descrita para entendermos melhor o papel de cada propriedade e como os nodos se relacionam.

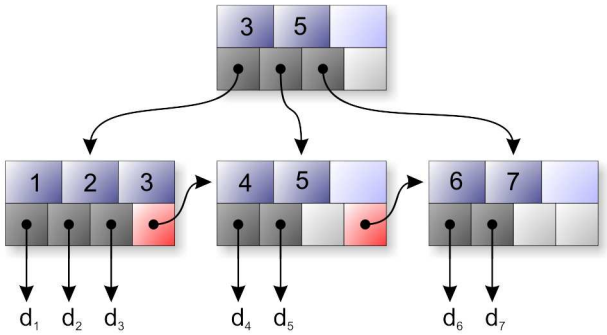

Figura 3: Arvore B+ (LUNKES, FÁBIO; 2010)

Os quadrados em cor lilas, representam as chaves Ki, os quadrados em cor preta são os ponteiros Pi para os filhos e o quadrados em cor vermelha, são os ponteiros para o nodo com a próxima chave maior em relação ao nodo atual.

A árvore B+ é uma estrutura que pode tanto servir como organização de dados de um arquivo de índice, como também pode ser utilizada para organizar os dados de um arquivo de dados relacionados a chave do arquivo de índice. A vantagem da árvore B+ está na performace para realizar operações de inserção, exclusão e pesquisa e na complexidade de espaço, pois, como os nodos precisam ocupar no mínimo a metade de seu espaço, temos um uso do espaço em pelo menos 50% do arquivo.

2.4 SISTEMA PARA PROCESSAMENTO DE TRANSAÇÕES CONCORRENTES

Geralmente denominado Sistema de Gerencia de Transações Concorrentes, sua motivação existe em função de haver um conjunto de sequências de operações que transações acessando concorrentemente a base dados, não podem executar. Estas sequências de operações, que várias transações podem emitir, forma o que chamamos de escala de execução. Esta escala pode ser válida ou não. Nossa preocupação, e a do gerente de processamento de transações concorrentes, está sobre a escala que não é permitida. Uma escala permitida é dita serializavel. Escalas não serializaveis devem ser tratadas pelo gerente de transações concorrentes para adiar operações sobre a base de dados ou até mesmo aborta-las. Para identificar escalas inválidas é necessário definir regras de acesso entre operações de transações diferentes.

Com o gerente de transações concorrentes obtemos ganhos de performace sobre as operações emitidas pelas transações. O ponto mais importante está baseado nas características distintas entre as memórias e o processamento paralelo. O tempo gasto entre operações de entrada e saída na memória secundária, quando comparadas com o tempo de processamento, é muito grande, ou seja, enquanto uma transação realiza uma operação de entrada ou saída sobre a memória secundária, milhões de outras instruções podem ser processadas em paralelo para atender outras transações, aproveitando assim, o paralelismo que podemos realizar entre ciclos de CPU e atividades de I/O. Em função desta grande diferença, é possível obter um rendimento ou aproveitamento bem maior do processamento de uma máquina computadorizada. Outra motivação está baseada nos tempos distintos de processamento para transações diferentes, pois, caso elas fossem executadas sequêncialmente, uma pequena transação poderia esperar muito tempo até outra maior ser concluída. Se as transações puderem ser executadas em paralelo, oferecendo uma alocação do processamento cooperativo, uma transação pequena pode não precisar esperar uma grande transação terminar.

Este aproveitamento do processamento está baseado em conceitos de programação paralela que são implementados pelos sistemas operacionais, pelo SGBD ou ambos. Alguns sistemas operacionais, por exemplo o Microsoft Windows, implementam um escalonamento de processamento chamado Round-Robin, ou seja, rodizio de processamento. Cada processo ganha um quantum de tempo, uma pequena fração de tempo lhe é oferecida para suas

necessidades de processamento. Ao termino de seu quantum de tempo, um próximo processo adquire o processador e realiza o processamento para seu quantum de tempo. Assim, segue sucessivamente o processamento de cada um. É evidente que o processador irá processar instruções uma de cada vez, porém, em função do rodizio de processamento, existe uma distribuição mais igualitária do tempo de processamento para cada processo. Isso será bem evidente, por exemplo, em nosso caso quando tivermos transações com necessidades de mais tempo de processamento, ou seja, transações com tempos de processamento bem distintos (H. M. DEITEL; P. J. DEITEL, 2003, p.771).

O sistema gerente para processamento de transações concorrentes é um módulo do sistema de armazenamento que vai contribuir para que as propriedades de uma transação sejam atendidas.

Baseado nestas premissas cabe ao gerente de transações concorrentes escalonar, ou seja, criar uma escala de execução, que nada mais é que, a sequência das operações que as transações podem executar concorrentemente. A determinação de uma escala de execução existe em função da dependência dos itens de dados aplicados a operações de várias transações. Estas dependência exigem que um item de dado seja primeiramente processado por uma transação, para só depois, poder ser aplicado a outra transação. Sua ideia básica está em construir uma escala de execução das operações que produza o mesmo efeito que uma escala de transações executadas sequêncial, uma após a outra.

2.4.1 OPERAÇÕES DAS TRANSAÇÕES

As transações, são para o sistema de processamento de transações concorrente, clientes que emitem pedidos de serviço. Estes pedidos, são operações básicas que vamos descrever agora, compondo assim, as ações que a transações vão realizar sobre a base de dados. Abaixo descrevemos o funcionamento de duas operações básicas de transações e suas implicações.

- *READ*(X): Realiza a transferência de um item de dado X do banco de dados para um *buffer* local alocado à transação que executou a operação de READ.

- *WRITE*(X): Realiza a transferência de um item de dados X do *buffer* local da transação que executou a operação, para o banco de dados.

Quando um conjunto de operações de várias transações diferentes concorrem um mesmo item de dado, podem ocorrer alguns problemas com leituras e escritas sobre este item de dado, caso não tomemos alguns cuidados com a ordem em que estas operações são executas.

Um dos problemas está relacionado as operações conflitantes de transações concorrentes. Vamos a um exemplo clássico, exemplificado na Tabela 1, na área de banco de dados. Duas contas correntes A e B, com os respectivos saldos de R$ 1000,00 e R$ 2000,00. Será realizado uma operação de transferência de dinheiro da conta A para a conta B no valor de R$ 50,00. Imaginemos que a conta A e B sejam cada uma um item de dado individual no banco de dados. Dessa forma imaginemos duas transações acessando concorrentemente estes dois itens de dados. Com a descrição das operações *READ*, *WRITE* e nosso exemplo, vamos descrever uma escala de execução das operações de duas transações executadas serialmente. Primeiro a transação T1 executa suas operações, depois a transação T2 executa as suas. Em todos os momentos vamos representar a chamada das operações das transações por meio de uma tabela. As colunas representam transações e as linhas suas operações no tempo. A linha do tempo cresce no sentido das linhas, então a primeira linha é o primeiro instante e a última o final.

Tabela 1: Escala de Transações

T1	T2
read(A)	
A←A - 50	
write(A)	
read(B)	
B←B + 50	
write(B)	
	read(A)
	temp ← A * 0,1
	A ← A - temp
	read(B)
	B ← B + temp
	write(B)

Na Tabela 2 está um exemplo de duas transações executadas concorrentemente com operações conflitantes.

Tabela 2: Transações Concorrentes

T1	T2
read(A)	
A ← A - 50	
write(A)	
	read(A)
	Temp ← A * 0,1
	A ← A - temp
	write(A)
read(B)	
B ← B + 50	
write(B)	
	read(B)
	B ← B + temp
	write(B)

Conforme a Tabela 2, transações concorrentes podem gerar operações conflitantes. As operações conflitantes são aquelas em que um item de dado só pode ser acessado por uma transação depois que outra terminar o processamento. Podemos evidenciar isso nas operações *write*(A), *read*(A) e depois *write*(B) e *read*(B). Por exemplo, a primeira operação de *read*(A) da transação T2 deve esperar o transação T1 terminar de escrever *write*(A), pois, foi realizada uma operação A := A – 50. Se a transação T2 lê-se o item de dado A antes da T1 terminar de escrever o item de dado A, ela poderia ler o valor antes de realizar o cálculo, gerando um erro. Existem várias abordagens para tratar de transações concorrentes e assim implementar o isolamento. Todas as técnicas empregadas tem como objetivo identificar uma ordem na qual as operações não são permitidas.

Nas próximas seções vamos analisar alguns métodos para tratar operações conflitantes entre transações. Um sistema para processamento destas transações, pode utilizar um ou mais

métodos para escalonar estas operações.

2.4.2 PROTOCOLO POR BLOQUEIOS

Esta ideia está baseada no principio de conceder bloqueio ao item de dado Q, para a execução de uma operação de uma transação, apenas se este bloqueio do item for compatível a operação que a transação pretende executar (SILEBERSCHATZ, 1999, p. 471). Existem dois tipos de bloqueio.

- Compartilhado: Se uma transação Ti, obteve um bloqueio compartilhado, denotado por S, do item de dado Q, esta transação só pode executar a operação de leitura sobre Q.

- Exclusivo: Se uma transação Ti, obteve um bloqueio exclusivo, denotado por X, do o item de dado Q, então Ti pode executar operação de leitura e escrita sobre Q.

Os bloqueios serão controlados pelo escalonador, que irá conceder os modos de bloqueio para quando uma transação solicita-lo em função de uma operação que ela está emitindo. A decisão para o escalonador conceder um bloqueio sempre vai ser determinada em função de um modo de bloqueio que o item de dado Q já tenha. Para tanto, vamos exemplificar os modos de bloqueio compatíveis na Tabela 3. S representa um modo de bloqueio compartilhado e X um modo de bloqueio exclusivo.

Tabela 3: Compatibilidade de Bloqueios

	S	**X**
S	VERDADEIRO	FALSO
X	FALSO	FALSO

A Tabela 3, representa a compatibilidade entre um par de tipos de bloqueio. Quando for verdadeiro, indica que o bloqueio da linha e coluna correspondentes são compatíveis.

Para o escalonador conceder bloqueios será necessário seguir um protocolo para poder definir as regras de funcionamento. Estas regras são necessárias para garantir a consistência e isolamento.

> • Seja o conjunto de transações {T0, T1, … , Tn} participantes de uma escala de execução S. Dizemos que Ti precede Tj em S, denotado Ti → Tj, se há um item de dado Q tal que Ti consegue bloqueio do tipo A sobre Q e depois Tj consegue bloqueio do tipo B sobre Q e comp(A,B) = falso. Se Ti → Tj, então essa precedência implica que, em qualquer escala serial equivalente, Ti precisa aparecer antes de Tj(SILEBERSCHATZ, 1999, p. 475).

O protocolo acima mencionado garante seriabilidade de conflito se e somente se para todas as escalas possíveis as relações de precedência formem um grafo acíclico. Apesar de tudo, ainda existem problemas, uma transação pode ficar inane, ou seja, sempre ficar esperando obter um bloqueio sobre um item de dado que sempre é rebloqueado por outra transação que emite uma operação compatível com o bloqueio atual do item de dado. Para tanto, o escalonador deve seguir duas regras para conceder bloqueios. O bloqueio no modo M para uma transação Ti, só é concedido para o item de dado Q quando:

1. Não há nenhuma outra transação com bloqueio sobre Q, que seja conflitante com o modo M.

2. Não há nenhuma outra transação esperando obter um bloqueio que tenha realizado sua solicitação antes da transação Ti.

Uma protocolo baseado em bloqueios que garante a serialização de conflito é o bloqueio em duas fases. Sua regra está baseada forma como são concedido e liberados os bloqueios que já citamos. Uma transação emite seus pedidos de bloqueio em duas fases.

1. Fase de expansão: Uma transação pode obter bloqueio, mas não pode liberar nenhum.

2. Fase de encolhimento: Uma transação pode liberar bloqueios, mas não consegue obter nenhum bloqueio novo.

Mesmo assim, não tem-se a garantia de não gerar um *deadlock* entre transações. Outro problema que existe no bloqueio em duas fazes é o *rollback* em cascata, caso uma transação Ti em conflito com uma transação Tj, leia um item de dado Q gerado primeiro por Tj que

falha depois que a operação de Ti tenha sido executada. Para evitar isso, é necessário um bloqueio rigoroso, que nada mais é que manter o bloqueio sobre o item de dado até que a transação que o possui seja consolidada. Ainda assim podem haver escalas de serialização de conflito que não podem ser obtidas por meio do protocolo de bloqueio em duas fases.

2.4.3 PROTOCOLO BASEADO EM TIMESTAMP

Assim como o protocolo baseado em bloqueios o protocolo baseado em *timestamp* tem o mesmo objetivo de definir uma ordem serializavel das operações de transações. Este protocolo, difere do protocolo por bloqueios, baseia-se em *timestamp*. O *timestamp* é uma medida de tempo que é atribuída a cada transação quando ela é ativada. Pode-se utilizar como medida de *timestamp* o relógio do sistema operacional ou um contador lógico incrementada a cada nova transação. O que é importante é ter uma informação que caraterize a ordem em que as transações são ativadas no tempo.

A ideia principal do protocolo baseado em *timestamp,* é definir um *timestamp* para cada transação e assim obter a informação da ordem delas no tempo. De posse do registro de timestamp na transação, deve-se atribuir ao item de dado o ultimo timestamp de cada transação que o leu e o modificou. Para as próximas transações, será comparado o *timestamp* de leitura ou escrita do item de dado, para avaliar se a operação da transação é permitida(HECTOR; ULLMAN; WIDOM, 2001, p.556).

O método por *timestamp* tem seu protocolo definido por dois conjuntos de regras, um para operação *read* e outro para operação *write*. Para a operação *write* iremos utilizar a regra de escrita de Thomas. Iremos convencionar, conforme padrão de outras bibliografias, que as propriedades serão sentenciadas primeiro e seguidas da entidade a qual pertencem entre parenteses. Exemplo, TS(Ti), indica que a propriedade TS (*Timestamp*) é da transação Ti. Abaixo descrevemos nossa legenda:

1. TS(Ti) : Representa o *timestamp* da transação Ti.

2. R-timestamp(Q): Representa o maior *timestamp* da transação que leu o item de dado Q

com sucesso.

3. W-timestamp(Q): Represeta o maior *timestamp* da transação que escreveu o item de dado Q com sucesso.

4. C(Q): Chamado bit de consolidação, informação de tipo lógico. Vai possuir valor verdadeiro se e somente se a transação mais recente que o gravou, já foi consolidada. Seu objetivo é evitar leituras sujas, ou seja, evitar que uma transação T lê dados gravador por uma transação U, e depois U é abortada.

O escalonador baseado em *timestamp* pode retornar três respostas para a solicitação da operação de uma transação.

1. Cancelamento da solicitação

2. Abortar a transação Ti(se Ti violar a realidade física) e reiniciar Ti com um novo valor de *timestamp*. Abortar e reiniciar é frequentemente chamado de reversão.

3. Atrasar Ti e mais tarde decidir abortar Ti, ou conceder a solicitação.

Algoritmo para a operação *read*(Q) de uma transação Ti exibido na Figura 4 (HECTOR; ULLMAN; WIDOM, 2001, p.561).

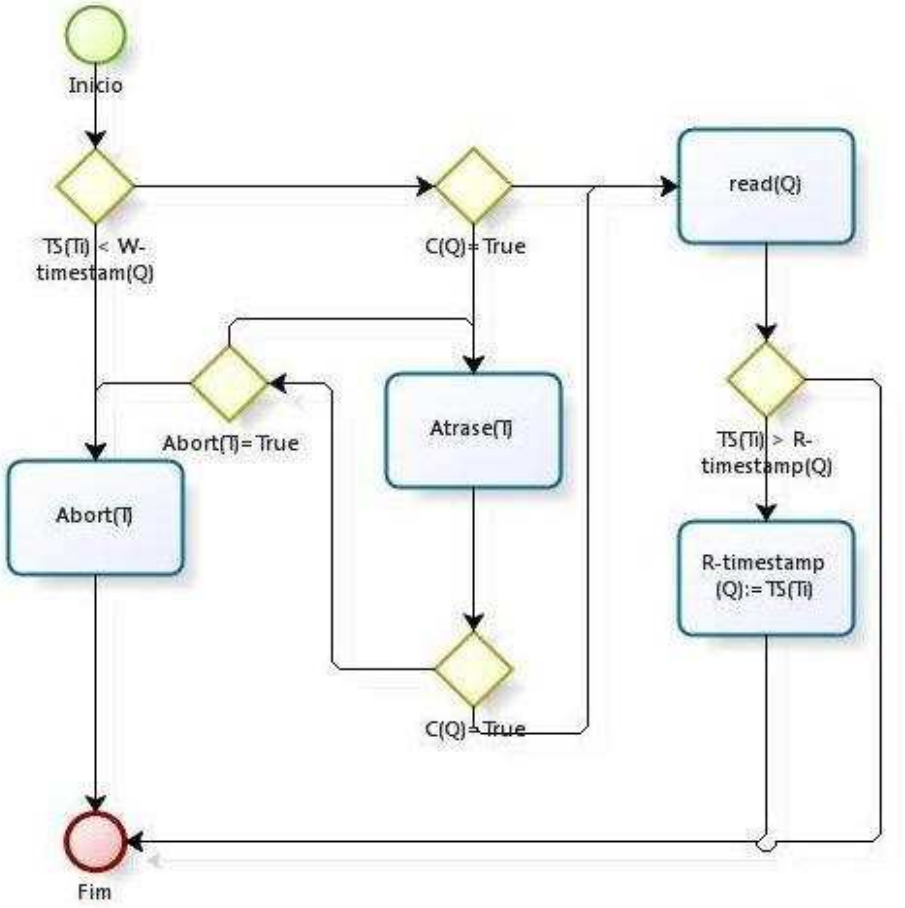

Figura 4: Operação *Read* da Transação

A. Se TS(Ti) < W-timestam(Q), então Ti precisa ler um valor de Q que já foi sobrescrito por uma transação mais recente. A operação *read* de Ti é abortada e reiniciada novamente com um *timestamp* novo e maior.

B. Se TS(Ti) >= W-timestamp(Q) a leitura é fisicamente realizável .

   I. Se C(Q) é verdadeira, conceda a solicitação. Se TS(Ti) > R-timestamp(Q), defina R-timestamp(Q) com o valor de TS(Ti); caso contrário não altere R-timestamp(Q).

   II. Se C(Q) é falso, atrase T até C(Q) se tornar verdadeira ou a transação que gravou Q abortar.

Algoritmo para a operação *write*(Q) pela regra de Thomas de uma transação Ti exibido

na Figura 5 (HECTOR; ULLMAN; WIDOM, 2001, p.561).

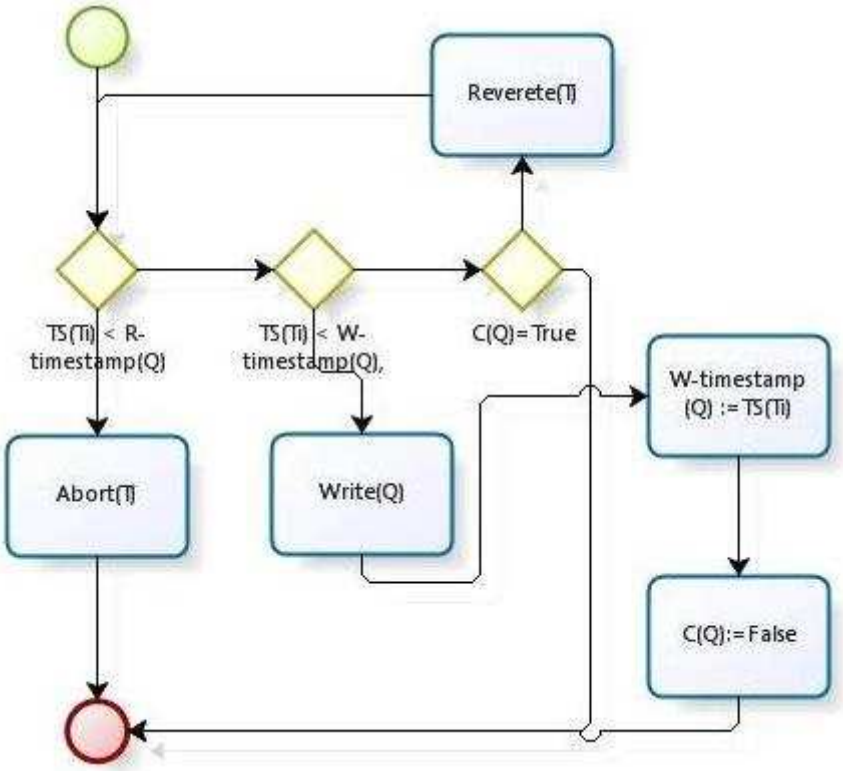

Figura 5: Operação *Write* da Transação

A. Se TS(Ti) < R-timestamp(Q), então o valor Q que Ti esta gerando já foi usado anteriormente e o sistema considerou que não seria mais necessário. Indica que o item de dado Q da transação Ti está sendo escrito por uma transação mais velha do que a última que leu o item de dado Q. A operação *write* será rejeitada e Ti revertida.

B. Se TS(Ti) >= R-timestamp(Q) mas TS(Ti) < W-timestamp(Q), então Ti está tentando escrever um valor de Q velho, outra transação mais nova já escreveu Ti. Se C(Q) é verdadeiro, então a transação que gravou Q ates se consolidou, e simplesmente ignoramos a gravação feita por Ti; permitimos que Ti prossiga e não fazemos nenhuma modificação no banco de dados. Entretanto, se C(Q) for falso, devemos retardar Ti como no item B.II.

C. Se TS(Ti) >= R-timestamp(Q) e TS(Ti) >= W-timestamp(Q) a gravação é

fisicamente realizável e deve ser executada.

I. Grave o novo valor de X.

II. Defina W-timestamp(Q) com o valor de TS(Ti).

III. Defina C(Q) como falso.

Em um sistema de banco de dados a operação de *delete* sempre é uma operação lógica, podendo assim, ser considerada como uma operação de escrita(*write*), que neste caso, irá apenas marcar o item de dado como excluído. Na base de dados o espaço deste registro poderá ser ocupado por outro registro.

Algoritmo para a operação *delete*(Q) de uma transação Ti na Figura 6. Este algoritmo irá definir operações conflitantes entre transações concorrentes. Para este algoritmo vamos considerar que Ti emita uma operação de *delete*(Q).

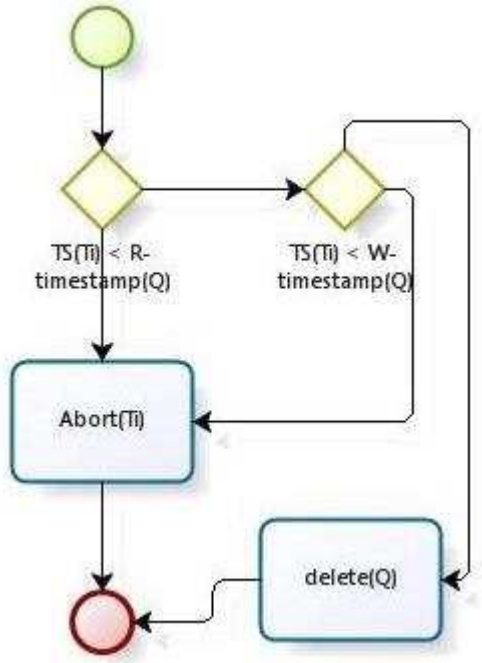

Figura 6: Operação *delete* da Transação

A. Se TS(Ti) < R-timestamp(Q), então a transação Ti está excluindo um item de dado

Q, que recebeu um leitura por uma transação mais recente. A operação *delete* será rejeitada e a transação Ti desfeita.

B. Se TS(Ti) < W-timestamp(Q), então a transação Ti está tentando excluir um item de dado Q, que foi escrito por uma transação mais recente. A operação *delete* é rejeitada e a transação Ti é desfeita.

C. Em outros casos, a operação *delete* é executada.

A operação de *insert*, emitida por uma transação, entra em conflito com as operação de *read*, *write* e *delete*, pois, um item de dado que não existe não pode sofrer uma operação de *read*, *write* ou *delete*. Sendo assim, quando um item de dado Q é inserido por uma transação Ti, os atributos de Q, R-timestamp(Q) e W-timestamp(Q) são atribuídos pelo TS(Ti). Quando uma transação Ti emitir uma operação de *read*, *write* ou *delete* sobre um item de dado Q que não existe será rejeitada e desfeita.

2.4.4 PROTOCOLO BASEADO EM VALIDAÇÃO

Até o momento vimos dois protocolos para ordenação das operações das transações, um baseado em bloqueios e outro em *timestamp*. Conforme funcionamento desses dois protocolos, podemos perceber que um escalonador baseado em qualquer um deles impõe sobrecarga de processamento de transações, pois, algumas podem ser atrasadas ou desfeitas. Quando as transações são de somente leitura, mesmo sem haver uma supervisão da ordem das operações, não haveria como a base de dados ficar inconsistente. Em função das transações de somente leitura não precisarem supervisão do escalonador para deixar a base de dados sempre consistente, seria uma oportunidade para diminuir o *overhead* de processamento no escalonador. Com isso, foi criado o protocolo baseado em validação, onde, cada transação possui duas ou três fases durante sua vida, dependendo da operação que ela executa (SILBERSCHATZ, 2006, p. 438). Vamos as fases na ordem que estão declaradas:

1. Fase de leitura: Nesta fase a transação realiza a leitura dos itens de dados e os copia para seu *buffer* local. Todas as operações de leitura e escrita serão sobre o seu *buffer* local sem atualizar a base de dados.

2. Fase de validação: A transação solicita um teste de validação para o escalonador, na tentativa de verificar se sua operação *write* dos resultados de seu *buffer* não irão violar a serialização.
3. Fase de escrita: Se a transação teve sucesso na validação da fase 2, então ela copia os dados para a base de dados. Caso contrário, o sistema reverte a transação.

Semelhante ao protocolo por *timestamp*, agora iremos associar a cada transação três *timestamps*, um para cada fase, e assim saber quando cada uma ocorreu, para poder então, determinar se uma operação de escrita é permitida. Cada *timestamp* será definido em ordem de tempo começando pela fase de leitura, depois fase de validação e por fim fase de escrita. Para deixar claro a função de cada *timestamp*, vamos descreve-los:

1. *Start*(Ti), representa o momento em que Ti teve inicio.
2. *Validation*(Ti), representa o momento em que Ti terminou sua fase de leitura e começo sua fase de validação.
3. *Finish*(Ti), representa o momento em que Ti terminou sua fase de escrita.

Será determinado a ordem de serialização pelo método do *timestamp*, usando como *timestamp* da transação, o *timestamp* de validação. Dessa forma se TS(Ti) < TS(Tj), para qualquer escala criada, deverá respeitar a ordem em que Ti vem antes de Tj. Usando o *timestamp* da validação para ser o timestamp da transação, obtemos um tempo de resposta menor do escalonador, quando solicitado a validação, caso a concorrência não seja grande.

O teste de validação para uma transação Tj exige que, para todas as transações Ti com TS(Ti) < TS(Tj), uma das duas condições a seguir necessitem ser mantidas(SILBERSCHATZ, 2006, p. 439).

1. *Finish*(Ti) < *Start*(Tj). Como Ti completa sua fase de gravação antes de Tj iniciar, a ordem de serialização é realmente mantida.
2. O conjunto de itens de dados escritos por Ti não possui intersecção com o conjunto de itens de dados lidos por Tj, e Ti, complete sua fase de escrita antes que Tj comece sua fase de validação(*Start*(Tj) < *Finish*(Ti) < *Validation*(Tj)). Este teste tem como objetivo evitar que as escritas de Ti e Tj não sejam sobrescritas.

2.4.5 PROTOCOLO DE ÁRVORE

Este protocolo, baseado também em bloqueios, tem o objetivo de oferecer bloqueios

para acesso concorrente a estruturas de dados organizadas em níveis hierárquicos, como as árvores. A estrutura de dado em árvore é aplicada para realizar a indexação dos dados de um banco de dados ou até mesmo organizar o arquivo de dados.

O protocolo de árvore tem seu funcionamento definido pela aplicação bloqueios aos nodos da árvore, assim como foi explicado na seção sobre o protocolo por bloqueio. Sendo ele um protocolo que funciona por bloqueios, suas características são as mesmas do protocolo por bloqueio, exceto no *deadlock* que não é gerado. Seu funcionamento é muito simples, e ele tem condições de oferecer acesso concorrente bem maior, quando comparado com um protocolo de duas fases ou *timestamp*, para uma árvore.

O protocolo em duas fases deixaria o acesso à uma arvore praticamente serial, inaceitável, pois, como sabemos, ele precisa em uma primeira fase adquirir todos os bloqueios e numa segunda fase libera-los juntos. Por exemplo, se um bloqueio exclusivo é adquirido no nodo raiz de uma arvore B+, nenhuma transação poderá executar nada mais até ele ser liberado. Assim uma transação por bloqueios, em duas fases, poderia bloquear muitos nodos, inclusive a raiz, permitindo o acesso apenas depois que todos os nodos fossem desbloqueados. Essa demanda por bloqueios não é interessante, deixa o nível de concorrência muito baixo.

Estruturas de dados como a árvore B+ são muito utilizadas em sistemas de armazenamento, tanto para indexar informações como também para organizar os arquivos de dados. Dessa forma, o protocolo em árvore tem grande aplicação nestes sistemas. Abaixo descreveremos as regras de funcionamento do protocolo (HECTOR; ULLMAN; WIDOM, 2001, p.551):

1. O primeiro bloqueio de uma transação pode ser aplicado em qualquer nodo.
2. Bloqueios a nodos subsequentes só podem ser adquiridos se a transação possui bloqueio sobre o nodo pai.
3. Os nós podem ser desbloqueados em qualquer momento.
4. Uma transação pode não bloquear novamente um nodo que tenha liberado bloqueio, mesmo que mantenha ainda bloqueio sobre o nodo pai.

A ordem serial que o protocolo em bloqueio impõe sobre as transações, é criada em função da necessidade de acesso a um nodo por meio de um nodo pai, ou seja, se uma transação Ti precisa acessar o nodo N2, sendo N1 pai de N2, primeiro é necessário adquirir um bloqueio sobre N1. Depois que Ti adquirir este bloqueio, se uma transação Tj tentar

também adquirir bloqueio sobre o nodo N1, ela precisa esperar Ti libera-lo. A aquisição dos bloqueios aos nodos pai, vai representar a ordem das operações das transações, tratando a serialização de conflito.

Alguma condições que um nodo se encontra nos motivam a aplicar ou não um bloqueio. Por exemplo, se estivermos inserindo uma nova chave em um nodo de uma árvore B+, considerando que ela possua muitos níveis, mais de 3, ao deixar o primeiro nível da raiz é de nosso interesse desbloqueá-la, pois, ainda serão visitados mais dois nodos em dois níveis abaixo e precisamos bloquear apenas o nodo pai, de um nodo, para modifica-lo, que no caso seria o nodo do segundo nível.

2.4.6 COMPARAÇÃO DOS PROTOCOLOS

Na Tabela 4, estão algumas características consideradas relevantes para este estudo de cada protocolo. Na primeira coluna estão as características, e nas outras colunas, estão os protocolos.

Tabela 4: Características dos Protocolos

Característica/ Protocolo	**Bloqueio**	**Timestamp**	**Validação**
Esquema de validação	Pessimista	Pessimista	Otimista
Concorrência alta	Atrasa	Reverte	Reverte
Espaços para armazenamento	Número de elementos bloqueados	Elementos lidos e modificados	Elementos lidos, modificados e conjuntos do protocolo.
Overhead de Processamento	Alto	Baixo	Baixo
Tratamento de reversão necessário	Reverte	Antecipa	Reverte
Deadlock	Gera	Não Gera	Não Gera
Rollback cascata	Atrasa	Atrasa	Atrasa
Complexidade	Grande	Pequena	Média
Seriação de visão	Não identifica	Identifica Com regra *write* de Thomas	Identifica Com regra *write* de Thomas

O protocolo de árvore que citamos na seção 2.4.5, vai fazer parte das características da coluna bloqueio, pois, como seu funcionamento é baseado em bloqueios, sua diferença reside em ser aplicado sobre estruturas de dados de árvore e evitar *deadlock*. O *deadlock* é evitado em função da disposição dos nodos da árvore, como como sabemos, uma árvore é um grafo acíclico, não possui ciclos, e assim não gera *deadlock*, pois, os acessos das transações serão ordenados pela disposição dos nodos da estrutura, formando uma árvore caso fossem assim representados.

As características dos protocolos que foram levantadas foram identificadas nos estudos para este trabalho e tem como objetivo oferecer condições de avaliar as implicações

computacionais nas suas implementações. Por exemplo, espaço de armazenamento nos da uma dimensão da complexidade de espaço sobre a memória principal, o quanto é necessário em termos de estruturas de dados para armazenar os controles da concorrência. A concorrência, seriação de visão, *deadlock* e *overhead* de processamento nos dão um dimensão da complexidade de tempo dos protocolos, elas influenciam no tempo definido pelo processamento que lhes é exigido. É claro que a dimensão relacionada é qualitativa, seria necessário uma implementação específica de cada protocolo para poder comparar os tempos quantitativamente.

2.5 SISTEMA DE RECUPERAÇÃO

Este sistema tem como objetivo garantir as propriedades de atomicidade e durabilidade. Este objetivo é atingido por meio da restauração do banco de dados para um estado consistente que existia antes da ocorrência de uma falha. Estas falhas podem ser falhas de transação lógicas ou acidentais, quedas de sistema ou falhas de disco. Para realizar esta tarefa é necessário implementar o que comumente é chamado de armazenamento estável (SILBERSCHATZ, ABRAHAM; 1999, p. 511).

O armazenamento estável será objeto de discussão do capítulo. Ele é implementado realizando o tratamento entre os dados que estão no meio volátil e não-volátil. Para entendermos como o armazenamento estável irá ser composto, vamos definir como será realizado o acesso aos dados e o seu envio para o banco de dados.

Como já havia comentado, as transações irão emitir operações de leitura, escrita, exclusão e modificação para o gerente de *buffer.* Para uma transação realizar suas operações, será necessário interagir com três espaços de endereço:

1. Espaço de endereços dos blocos, que contem os itens de dados no disco.
2. Espaço de endereços da memória principal, gerenciado pelo gerente de *buffer.*
3. Espaço de endereço dos dados de uma transação.

Para uma transação poder carregar a sua área local de dados, ela irá interagir com o

sistema gerente de *buffer.* Para uma transação, o banco de dados é o próprio sistema gerente de *buffer.* Quando um item de dados é escrito ou lido, a transação está operando sobre o gerente de *buffer.* O sistema gerente de *buffer,* por sua vez, será responsável por armazenar definitivamente os dados no disco e recupera-los. A unidade de transferência de dados entre o disco e o gerente de *buffer,* é o bloco. Blocos de dados no disco são chamado de bloco físicos e blocos de dados na memória principal são chamados de blocos de *buffer.* Um bloco contém n itens de dados. Um item de dado não poderá ter uma tamanho superior a de um bloco. Essa restrição é uma realidade na maioria dos sistemas de armazenamento. A escolha do critério que o gerente de *buffer* irá adotar para escrever um bloco de *buffer* ou ler um bloco físico de dados, tem como objetivo oferecer alta performace, por meio da minimização de acesso ao disco rígido realizado nas operações de leitura e escrita. Antes de descrever as condições em que isso irá ocorrer, vamos definir as operações primitivas do gerente de *buffer.*

- ***input*(B)**: Transfere o bloco físico B para o memória principal, criando um bloco de *buffer.*

- ***output*(B)**: Transfere um bloco de *buffer* B para o disco, trocando o bloco físico do disco pelo novo bloco de *buffer.*

As operações *input* e *output* serão emitidas pelo gerente de *buffer* quando julgar necessário. Quando as transações solicitarem itens de dados para o gerente de *buffer,* ele vai precisar utilizar a operação input(B), caso o item de dado não esteja em um bloco de *buffer.* De posse da descrição das operações *input* e *output* emitidas pelo gerente de *buffer,* podemos descrever outras duas operações, também emitidas pelas transações, que interagem com o gerente de *buffer.* Para realizar a descrição, vamos definir que xi é o item de dado da área local de uma transação Ti e o item de dado será X representará o mesmo item, porém, dentro do seu bloco de *buffer.* É importante entender que o item de dado X e xi representam o mesmo item de dado, porém, são duas cópias, ou seja, tem duas regiões de memória distintas.

- ***read*(X)**: Copia o item de dado X para a área local xi.

    - Se o bloco Bx, no qual reside X, não está na memória, então emite um *input*(Bx).

- Copia o valor de X do bloco de *buffer* para a área local xi.

- ***write*(X)**: Copia o valor de xi para a o item de dado X residente no bloco de *buffer.*

  - Se o bloco Bx que contém o item de dado X não existe na memória, emite um *input*(Bx) .

  - Copia o item de dado xi para o item X do bloco de *buffer* Bx.

Conforme descrevemos, fica fácil perceber que tando a operação de *read* ou *write* podem pedir para o gerente de *buffer* executar a operação *input*, para transferir dados do disco para o gerente de *buffer.* O momento em que será executado a operação de *output* será de responsabilidade do gerente de *buffer* e deverá existir um conjunto de condições para que isso aconteça em momentos apropriados. Esta relação entre o gerente de *buffer* e as transações, deverá ocorrer por algoritmos que implementem recuperação e atomicidade. Para poder implementar a recuperação, o gerente de *buffer* vai precisar armazenar seus dados em armazenamento estável, bem como todos os detalhes da modificação da base de dados. Estas informações serão necessárias para realizar a recuperação e atomicidade.

2.5.1 SISTEMAS DE RECUPERAÇÃO BASEADO EM LOG

A recuperação baseada em log é uma técnica aplicada para implementar a recuperação e atomicidade. Ela está baseada em um principio básico da redundância, ou seja, a cópia dos dados antes de alterarmos efetivamente a base de dados. A base de dados é uma área de endereços no disco e o log é outra área, também no disco. Vai depender da implementação estas áreas serem no mesmo arquivo ou um conjunto de arquivos (SILBERSCHATZ, ABRAHAM; 1999, p. 517).

O arquivo de log vai conter o registro de todos os eventos gerados pelas transações sobre os itens de dados. Estes eventos serão compostos por marcas de controle que indicam o estado que uma transação passou, valor antigo de um item de dado e valor novo.

Para que um dado seja gravado efetivamente na base de dados, serão necessários, antes de mais nada, registrar no log todos os eventos ocorridos. Antes de descrevermos as regras, vamos descrever cada tipo de registro do log para formar o conjunto de controles que descreverão melhor os eventos:

1. <START T>: Este tipo de registro representa que a transação T iniciou.

2. <COMMIT T>: Representa que a transação T, foi consolidada, possui o registro de modificação da base registrado no log.

3. <ABORT T>: Representa que a transação T foi abortada, em função de algum problema ou por necessidade da aplicação. Uma transação abortada deve possuir os valores dos itens de dados restaurados com os valores antigos.

4. <T, X, v>: Representa o registro de modificação realizado pela transação T de um item de dado X, onde v é o seu novo valor.

5. <T, X, v, w>: Representa a modificação do valor do item de dado X, pela transação T, onde o valor antigo de X, é v, e o novo, é w. Antes de qualquer modificação no banco de dados por uma transação T, deve-se primeiro gravar no log o registro <T, X, v, w>.

6. <START CKPT(T1...Tk)>: Ponto de verificação para delimitar uma extremidade de verificação dentro do log. T1...Tk são todas as transações ativas no momento da geração deste registro. As transações ativas não foram concluídas, pois, ainda não gravaram as modificações no disco. A criação desse registro deve ocorrer na seguinte ordem:

   I. Gravar no log o registro <*START* CKPT(T1...Tk)> e executar *FLUSH LOG*. T1...Tk é o conjunto de todas as transações ativas no momento da criação deste registro, aquelas que ainda não foram concluídas, ou seja, não possuem um registro *COMMIT* ou *ABORT.*

   II. Gravar todos os *buffers* sujos, ou seja, *buffers* que sofreram modificação e não as possuem gravadas em disco. Esta modificação é no sentido do elemento que está no disco do banco de dados ser diferente daquele do *buffer.* Durante este tempo

recebe-se novas transações também.

III. Grave o registro <*END* CKPT> do log e execute *FLUSH LOG*.

B. <*END* CKPT>: Registro gravado depois de todas as transações ativas T1...Tk forem gravadas no disco. Logo depois de gravar o registro, esvazia o log novamente. Com esse registro, temos a certeza que as transações ativas, durante a criação do ponto de verificação, foram todas esvaziadas para o disco.

Estes registros, assim como no controle de concorrência estão sendo representados com palavras em inglês, e siglas para as variáveis. Veremos no projeto do sistema estas mesmas representações, porém serão representadas por constantes numéricas para os eventos e inteiros longos para as transações. Os dados serão vetores de bytes. Estas representações facilitarão operações para processamento e ocuparão menos espaço.

O tratamento entre o valor antigo e valor novo de um item de dado, classifica dois tipos de sistema de recuperação denominados como modificação adiada do banco de dados e modificação imediata do banco de dados. Veremos cada um deles na sequência.

2.5.2 MODIFICAÇÃO ADIADA DO BANCO DE DADOS

Já descrevemos acima a função de cada registro de log. Agora vamos entender como eles se relacionam para compor o funcionamento do sistema com modificação adiada do banco de dados. A modificação adiada do banco de dados ou registro de log refazer, como o próprio nome diz, está relacionada no adiamento da modificação do banco de dados quando o item de dado é enviado para armazenamento estável.

Quando surge uma transação no gerente de *buffer*, a primeira coisa que deve ser realizada é o registro <*START* T>, para representar que aquela transação foi iniciada. Na sequência, o gerente de *buffer* pode enviar para o log os registros de modificação da base de dados. Se a operação for de escrita *write*, haverá um registro no log contendo o novo valor do item de dado no formato <T, X, v>. Neste ponto, deve-se observar que ao realizar este registro no log, nenhuma modificação no arquivo da base de dados foi realizada. Quando a transação

estiver parcialmente confirmada, ou seja, a última operação sua foi executada, um registro <*COMMIT* T> poderá ser enviado. Quando este registro *COMMIT* for enviado para o log, e somente depois dele ter sido gravado é que realmente pode-se realizar a modificação na base de dados. Até este momento a modificação do banco de dados foi adiada, por isso do nome da técnica.

Com este processo, o sistema de recuperação baseado em modificação adiada, tem condições de executar a operação de refazer modificação, pois, tendo apenas o novo valor e o registro de que a transação foi consolidada, por meio do registro *COMMIT*, lhe resta apenas refazer uma modificação em caso de falha. Esta operação é comumente chamada por diversos autores de operação *REDO*(T), ou seja, refaz a transação T. Com isso quando ocorre uma falha o registro do log é percorrido do inicio até o fim e refaz as modificações no banco de dados apenas para as transações com o registro *COMMIT*. As outras transações são desconsideradas, pois, temos a certeza de que nada foi modificado, foi adiado.

2.5.3 MODIFICAÇÃO IMEDITA DO BANCO DE DADOS

Modificação imediada ou registro de log desfazer/refazer(HECTOR; ULLMAN; WIDOM, 2001, p.484) é um método que contará com um registro de modificação de um item de dado maior que o registro do sistema com modificação adiada. Neste sistema cada item de dado conta com um registro no log que contém o novo e antigo valor no formato <T, X, v, w>.

O registro de log <T, X, v, w> tem como objetivo desfazer ou refazer o valor de um item de dado X, modificado pela transação T. Uma modificação será desfeita com o valor antigo v e refeita com o novo valor w. As operações realizadas pelo gerente de recuperação deverão tornar a base em um estado consistente que havia antes de uma falha qualquer. Para este registro, deve-se seguir duas regras quando uma transação for modificar um item de dado.

UR1: Se uma transação T, modificar um item de dado X, então deve aparecer primeiro no disco a gravação do registro de *log* <T, X, v, w> e apenas depois o novo valor da base de dados (HECTOR; ULLMAN; WIDOM, 2001, p.485).

UR2: Assim que o registro *COMMIT* aparecer no log, ele deve ser descarregado para o

disco. A operação *FLUSH LOG* deve ser executada na sequência (HECTOR; ULLMAN; WIDOM, 2001, p.486).

Como as operações tanto para a gravação dos blocos de *buffer* do banco de dados como os blocos de *buffer* do log são controladas pelo gerenciador de *buffers*, é necessário uma operação de *esvaziar log* para forçar a escrita no disco dos registros de log. Chamaremos de *FLUSH LOG* esta operação. Esta operação deve ser executada pelo gerente de log, que irá solicitá-la para o gerente de *buffer* que deverá copiar para o disco todos os registros de log. Esta operação será muito importante, pois, ela vai suceder o registro de log *COMMIT* e <T, X, v, w> que informam as modificações que serão realizadas na base de dados.

Com a descrição das operações podemos saber os efeitos sobre os três diferentes espaços de endereços, transação, *buffer,* banco de dados e os respectivos registros no log. Vamos descrever o estado de cada espaço de endereço na Tabela 5. Para abreviar o nome das colunas, M-X indica cópia do item de dado X para o *buffer* de memória e D-X a cópia do item de dado X para o disco.

Tabela 5: Ações sobre as memórias(inclusive arquivo de log)

Etapa	Ação	t	M-A	M-B	D-A	D-B	Log
1							<*START* T>
2	*read*(A, t)	8	8		8	8	
3	t← t * 2	16	8		8	8	
4	*write*(A,t)	16	16		8	8	<T, A, 8, 16>
5	*read*(B,t)	8	16	8	8	8	
6	t← t * 2	16	16	8	8	8	
7	*write*(B,t)	16	16	16	8	8	<T, B, 8, 16>
8	*FLUSH LOG*						
9	*output*(A)	16	16	16	16	8	
10							<*COMMIT*>
11	*FLUSH LOG* (*esvaziar log*)						
12	*output*(B)	16	16	16	16	16	
13							

Nota que *FLUSH LOG* aparece logo após a etapa 10 do registro *COMMIT*. Isso deve ser realizado para forçar o registro *COMMIT* a ser descarregado para o disco.

O registro *COMMIT*, pode aparecer antes ou depois das modificações do elemento do banco de dados. Esta liberdade existe em função do registro de modificação <T, X, v, w> possuir antigo e novo valor.

Um ponto importante a ser considerado é que uma transação não pode ler dados sujos de outra transação, porém, isso será tratado pelo gerente de transações concorrente.

Em função do registro permitir desfazer e refazer as modificações que uma transação realizou sobre os itens de dados, duas operações distintas devem ser realizadas. Elas devem atender o seguinte protocolo:

1. **Refazer,** todas as transações consolidadas, da mais antiga para a mais recente, ordem cronológica crescente. As transações consolidadas possuem o registro *COMMIT*.

2. **Desfazer,** todas as transações incompletas, da mais recente para a mais antiga, ordem cronológica decrescente. As transações incompletas são todas aquelas que possuem o registro *ABORT* ou não possuem o registro *COMMIT*.

É muito importante perceber que devido a regra UR1, disponível pelo registro de log desfazer/refazer, a ordem que o registro de log *COMMIT* aparece no log em relação a gravação dos dados no disco do banco de dados, não tem importância. Isso significa que posso ter uma transação consolidada com algumas ou todas as modificações gravadas em disco, ou uma transação não consolidada com algumas ou todas as modificações em disco. Se atender apenas as regras UR1 e UR2 poderei ter esta situação que é valida. Isso dará flexibilidade para o gerente de *buffer* administrar a gravação de seus *buffers* no disco, podendo aumentar a capacidade de operações de transações concorrentes.

Para o sistema recuperar-se de uma falha, será necessário percorrer o espaço de endereço do arquivo de log. Este espaço de endereço, deve ser lido para o passo 1 e 2 acima descritos. Este espaço é definido pela transação mais antiga até o último registro do log. Para

determinar a transação mais antiga, temos a ajuda dos registros de *ponto de verificação*.

Se o ultimo registro de ponto de verificação for um registro <*END* CKPT>, sabemos que o seu respectivo registro <*START* CKPT(T1...Tk)> foi gravado, também sabemos que todos os *buffers*, sujos existentes logo após o registro <*START* CKPT(T1...Tk)> foram gravados em disco(arquivo do log e do banco de dados) e que novas transações criadas durante a criação do ponto verificação também puderam ser gravadas no disco. Dessa forma conclui-se que para refazer as alterações é necessário verificar a partir deste último registro <*START* CKPT(T1...Tk)> até o fim do arquivo, todas as transações que possuem um registro *COMMIT*, deve-se refazer suas modificações no disco da base de dados. Não existe preocupação se estivermos refazendo transações que possivelmente já tenham gravado no disco suas modificações, pois, a ordem em que isso realmente ocorreu vai depender de como o gerente de *buffer* gravou seus blocos. Os únicos critérios obedecidos são as regras UR1 e UR2.

Se o ultimo registro do ponto de verificação for um <*START* CKPT(T1...Tk)>, então uma pane ocorreu durante a criação do ponto de verificação. Dessa forma tudo aquilo que tínhamos certeza que foi gravado quando encontramos primeiro o registro <*END* CKPT>, não poderá ser garantido, ou seja, não temos certeza que se todos os blocos sujos dos *buffers* foram gravados e também não temos certeza que todas as transações novas foram gravadas no disco. Dessa forma, conclui-se que devemos refazer todas as transações com registro *COMMIT* a partir do ultimo ponto de verificação completo (Entende-se por ponto de verificação completo aquele que possui um registro <*END* CKPT> em uma posição i, e seu correspondente registro de inicio <*START* CKPT(T1...Tk)> em uma posição j, sendo j menor que i e represente a ordem dos eventos) até o final do arquivo. Se este ponto de verificação não existir, vamos até o inicio do arquivo.

1) Se último o registro de ponto de verificação for <*END* CKPT>, então visite o seu registro <*START* CKPT(T1...Tk)>. Se o último registro não for <*END* CKPT>, então vá para o passo 3.

2) Para cada transação Ti com registro <*COMMIT*> até o final do arquivo, refaça a modificação no disco.

3) Visite o primeiro registro <*END* CKPT> anterior ao registro <*START* CKPT(T1...Tk)> do passo 1. Se ele não existir, vá para o inicio do arquivo.

4) Visite o registro <*START* CKPT(T1...Tk)> pertencente ao registro <*END* CKPT> do passo 3.

5) Execute o passo 2 e termina.

Se o último registro do ponto de verificação for <*END* CKPT> então uma pane ocorreu depois desse registro. Com o registro <*END* CKPT> temos a certeza que os *buffers* sujos foram gravados no disco e as transações criadas depois e durante também foram para o disco. Dessa forma, toda a transação sem o registro *COMMIT* ou com um registro *ABORT,* é candidata a ter necessidade de desfazer modificações no disco, pois, as garantimos com o registro <*END* CKPT>. Da mesma forma que ao refazer alterações, aqui também não nos preocupamos se iremos desfazer modificações já desfeitas. Sendo assim, iniciando a partir do final do arquivo em direção do seu inicio, desfazemos todas as transações sem *COMMIT* ou com *ABORT* que estivessem ativas na criação do ponto de verificação <*START* CKPT(T1...Tk)> e que tenham sido criadas depois ou durante o ponto de verificação. Para desfazer uma transação, deve-se, visitar todos os seus registros <T, X, v, w>, mesmo que isso fique antes do registro <*START* CKPT(T1...Tk)>, até o seus registro <*START* Ti>.

Se o ultimo registro do ponto de verificação for um <*START* CKPT(T1...Tk)>, a pane ocorreu durante a criação do ponto de verificação. Agora também não sabemos até onde as modificações ocorreram. Sendo assim, iremos desfazer as alterações de todas as transações sem registro *COMMIT* ou com registro *ABORT* que estivessem ativas durante a criação do ponto de verificação completo anterior e que tenham sido criadas depois e durante o ponto de verificação completo anterior. Se não existir um registro completo de ponto de verificação anterior ao último, ir até o inicio do arquivo.

1. Se o último registro do ponto de verificação for <*END* CKPT> então, criar um conjunto L de todas as transações ativas durante este último ponto de verificação e criadas depois sem registro <*COMMIT*> ou com registro <*ABORT*>.

2. Para cada transação do conjunto L, desfazer suas modificações com o registro <Ti, X,

v, w> até encontrar o registro <*START* Ti> de cada transação.

3. Se o último registro de ponto de verificação não for <*END* CKPT>, então iremos criar um conjunto de transações L que não possuem o registro <*COMMIT*> ou possuem o registro <*ABORT*> ativas durante o último ponto de verificação e criadas depois. Se não existir um ponto de verificação completo, ir até o inicio do arquivo.

2.5.4 COMPARATIVO ENTRE MODIFICAÇÃO ADIADA E MODIFICAÇÃO IMEDIATA

A modificação adiada comparada com a modificação imediata do banco de dados impactam diretamente sobre o gerente de *buffer*, aumentando ou diminuindo o número de informações necessárias para seu funcionamento.

Dependendo das condições que o gerente de *buffer* é submetido para gravação de seus blocos de dados, o número de operações de entrada e saída sobre o disco pode aumentar ou diminuir.

Na modificação adiada o tamanho do *buffer* precisa ser maior quando comparado com a modificação imediata, pois, precisamos manter os blocos no *buffer* até as transações se consolidarem. Isso quer dizer que depois de realizando o registro no log dos novos valores dos itens de dados, será necessário manter os blocos no *buffer* pois ainda é necessário envia-los para a base de dados. Essa condição vai aumentar o tamanho do *buffer* ou deixa-lo com resposta em tempo menor.

Com a modificação imediata, tempos uma flexibilidade bem maior, pois neste caso, os dados dos blocos são esvaziados do *buffer* conforme as condições de gerência de espaço do *buffer*, sem precisar tratar de restrições relacionadas ao conteúdo dos blocos ou regras do registro de log em questão. Na presença de qualquer problema tudo pode ser tratado, pois existe o valor antigo e valor novo do item de dado. É evidente que o tamanho do log será maior, mais isso trará mais benefícios que tornar o gerente de *buffer* dependente do registro de log para esvaziar seus blocos.

3 A PERSISTÊNCIA DE OBJETOS

A persistência dos objetos é um recurso que torna possível o mapeamento de objetos que estão na memória principal para a memória secundária e vice versa. Neste processo a carga de todas as variáveis é realizada automaticamente pelo sistema de persistência, tornando transparente o processo de leitura e escrita de objetos na base de dados. Neste trabalho o sistema de persistência será dividido em duas partes. Uma delas será realizada automaticamente pela API do Java, e a outra forma, será uma especialização da persistência padrão. A persistência automática poderá ser utilizada para persistência dos objetos da lógica de domínio, e a implementação do sistema gerente do banco de dados será realizada através da especialização do sistema de persistência padrão da API do Java. Mesmo que no presente trabalho a implementação seja realizada com a linguagem de programação Java, nada impede que o sistema venha a ser escrito para qualquer outra linguagem de programação. A única particularidade que vai existir, será o formato para representação dos dados em cada linguagem de programação. Para os objetos da aplicação a serem persistidos, suas classes deverão implementar uma *interface* chamada *Serializable,* concluindo assim que a persistência será por classe(SILBERSCHATZ, 1999, p. 262) e objetos não persistidos serão declarados com a palavra reservada *transient*. As estruturas que irão implementar o sistema gerente de banco de dados serão especializadas conforme a necessidade para atender requisitos do sistema de banco de dados.

Em Java o processo de persistir um objeto em disco é realizado através de um recursos de serialização dos objetos e logo após gravação do dado no disco. Serializar um objeto consiste em converte-lo para um vetor de bytes que contém um formato particular da API para representar os objetos pertencentes a uma classe. Um objeto serializado contém todas as informações necessárias para resgata-lo para memória secundária e recria-lo na memória principal.

3.1 VANTAGENS DA PERSISTÊNCIA DE OBJETOS

A maior vantagem em empregar a API para serialização de objetos do Java está no fato de pertencer a API padrão da linguagem que foi escolhida para implementar o trabalho e por tornar o processo automático sem nenhum código adicional. Caso fosse empregado outra linguagem, seria necessário implementar este sistema gerenciando ponteiros persistentes que alguns autores costumam chamar de *Swizzling* de Ponteiro (SILBERSCHATZ, 1999, p. 326).

3.2 API'S PARA PERSISTÊNCIA DE OBJETOS

Inicialmente, em função da necessidade de tratar, através de sistemas computacionais, grandes quantidades de dados, a área de banco de dados começou a se desenvolver, e durante o tempo, novos desafios, em novos contextos são apresentados para poderem ser resolvidos. Atualmente é impreterível a importância que os sistemas de banco de dados desempenham em nossa sociedade. Durante toda a sua história, diversos tipos ou famílias destes sistemas surgiram, como por exemplo modelo de rede, relacional, objeto-relacional e orientado a objetos.

Além dos tipos de sistemas de armazenamento citados, existe outro não muito comum, denominado sistema de banco de dados *key-value*, que é a proposta deste trabalho. Este tipo de SGBD, combina um pouco de um SGBDOO com SGBDR. Ele pode ser considerado apenas como um núcleo de um sistema de banco de dados, pois irá tratar apenas o armazenamento e recuperação dos dados. Não haverá linguagem de consulta, pois, ele irá fornecer suas funcionalidades através da interface de uma API do seu sistema de banco de dados.

Um sistema gerente de banco de dados, padrão chave-valor, contem seu registro formado por uma tupla, onde um elemento representa a chave e o outro o valor. A chave identifica o valor exclusivamente e o elemento valor, contém o dado a ser armazenado. Em um nível mais baixo de representação dos dados, os dois elementos são formados por um vetor de bytes que podem representar qualquer tipo de dado, seja ele um objeto, ou um valor padrão como inteiros, caracteres, números com ponto flutuante, data e valor lógico. Neste tipo

de sistema, o formato da chave ou do elemento valor, não tem importância, todos podem ser tratados como objetos para generalizar a representação de dados que o sistema aceita. O que deve ser levado em consideração é que as operações sobre o tipos de dados, sejam quais forem elas, devam ser executadas de forma transparente na memória do cliente.

Os sistemas gerentes de banco de dados chave-valor tiveram suas primeiras APIs junto com sistemas operacionais Unix e forneciam um meio de realizar armazenamento de dados por meio de programas escritos em linguagem C. Uma dessas API's é denominada DBM – Database Manager escrita em C, possibilita realizar operações de armazenamento e busca de dados. A partir da API DBM surgiram outras APIs, que foram apresentando um desenvolvimento gradativo das funcionalidades para armazenamento de dados. Sucessoras a DBM surgiram Ndbm em 1986, Sdbm em 1987, Tdbm, Qdbm, Tokyo Cabinet, VSDB e JDBM. Algumas não tem suporte a transação ou são para um único usuário. Veremos mais adiante com detalhes suas características.

Uma característica interessante, que é obtida com uma API de banco de dados chave-valor, é a realização de persistência de objetos por meio da própria linguagem de programação, tornando-a assim, um sistema persistente de objetos que forma uma extensão da API da linguagem de programação. Isso torna mais transparente operações sobre um sistema modelado no padrão orientado a objetos para realizar operações de armazenamento e recuperação, podendo encapsular todo o processamento para tais operações na API do SGBD. Havendo uma boa divisão entre as responsabilidade de armazenamento, por meio do encapsulamento de tais operações, fica dividido a implementação da lógica de domínio e armazenamento, minimizando seu impacto por meio da transparência destas operações.

A aplicação de um sistema para persistência de objetos transacional pode ser realizada para toda aplicação que exija armazenamento de dados pertencentes aos objetos do sistema a ser implementado. Podemos citar alguns exemplos de sistemas de banco de dados que possuem uma abordagem semelhante aos sistemas gerente de banco de dados chave-valor.

1. AmazonSimpleDB

2. Google BigTable

3. Jdbm

A API Jdbm, implementada por Cees de Groot e Alex Boisvert, é uma implementação simples de uma API escrita com a linguagem de programação Java, que permite a persistência de objetos Java ou *arrays* de bytes. A Jdbm é uma implementação simples que não possui todos os recursos que o sistema de banco de dados deve possuir. Esta API não possui suporte a transações, gerenciamento de concorrência(acesso serial) e sistema de recuperação. Atualmente o seu único suporte a transação é todo realizado na memória principal, não havendo a implementação do armazenamento estável que foi explicado no capítulo 2.5. Sua licença é BSD, e portanto, de código aberto(BOISVERT, Alex. Projeto Jdbm. 2001. Disponivel em: http://jdbm.sourceforge.net/).

A AmazonSimpleDB é uma API do mesmo padrão chave-valor, porém, bem mais madura. Sua versão é um produto da empresa Amazon Web Services LLC e suas principais características estão em oferecer alta disponibilidade, escalabilidade, flexibilidade e baixo custo com administração da base de dados, pois, este modelo não possui um gerenciador para administrar o sistema. As bases de dados são distribuídas geográfica, aumentando a disponibilidade e são realizados balanceamentos de carga conforme a demanda de dados (Amazon. AmazonSimpleDB. 2010. Disponivel em: http://aws.amazon.com/simpledb).

O GoogleBigTable é um sistema de armazenamento de dados desenvolvido para necessidades específicas da empresa Google para armazenamento e acesso a grandes bases de dados, na ordem de petabytes, armazenado em seus servidores. Atualmente este sistema é utilizado para as bases de dados de aplicações da Google como *Google Search, Google Earth & Maps, Google Finance, Google Book, Orkut YouTube e Blogger*(Google. GoogleBigTable. 2006. Desponivel em: labs.google.com/papers/bigtable.html).

É importante entender que muitas das características apresentadas são frutos do modelo computacional do sistema de armazenamento. E evidente que alguns sistemas, por já possuírem um evolução maior, tenham um grau de amadurecimento superior e assim um conjunto maior de funcionalidade já disponíveis.

3.3 AS INTEFACES DE PROGRAMAÇÃO

Na API do Java a serialização de objetos é realizada através da implementação da interface *Serializable* como já foi mencionado. Esta interface não possui nenhum método para ser implementado sendo necessária apenas para indicar para API que o objeto pode ser serializado. Um atributo chamado *serialVersionUID* é utilizado pelo runtime para controle de versões de objetos. O valor deste atributo é verificado quando um vetor de bytes está sendo transformado em um objeto. Caso o valor do atributo da classe que implementa a *interface Serializable* seja diferente daquele objeto já armazenado, uma *exception InvalidClassException* será lançada comunicando ao programador da diferença entre versão.

4 PROPOSTA DE UMA API TRANSACIONAL PARA PERSISTIR OBJETOS

Já mencionamos que a API para persistência de objetos é dentro do padrão chave-valor. Agora vamos abordar um nível mais baixo de abstração, onde tratamos das estruturas do sistema para implementar o sistema gerente de transações concorrentes e o sistema gerente de *buffer*. Estes dois sistemas vão compor a API para realizar persistência dos objetos. Os dois elementos, chave e valor, são considerados neste nível como simplesmente dois vetores de elementos do tipo de dado byte. Neste ponto, não considerando a representação dos dados que o vetor pode formar, não existe diferenças significativas entre um sistema de banco de dados relacional, objeto-relacional ou orientado a objetos. Acima desta camada existem sim diferenças, mas em termos de armazenamento de forma generalizada existem apenas vetores com elementos do tipo de dado byte, sejam eles a representação de um objeto complexo ou um tipo de dado primitivo como inteiros, caracteres, lógicos ou números de ponto flutuante.

O projeto do software implementado vai está embasado nas referências bibliográficas deste trabalho. São utilizados os seguintes artefatos de software da UML para representar o sistema, digrama de caso de uso gráfico, caso de uso textual, diagrama de classes, digrama de estado e diagrama de sequência. O diagrama de estado está sendo considerado em função da ter sido utilizado na seção 2.1, Figura 1, pois, os estados da transação devem ser considerados para a construção do sistema e assim ser parte do projeto.

A solução proposta, por meio da implementação do *software,* é realizada através de um protótipo de uma biblioteca de *software* que atende os requisitos de um sistema de armazenamento, realizando a persistência de objetos no formato binário gerado pela biblioteca de serialização da linguagem de programação Java. A persistência dos objetos serializados não poderá ser totalmente atendida com as propriedade ACID, pois, não foi implementado o sistema de recuperação.

A arquitetura do sistema, foi apresentada na Figura 2, no inicio deste trabalho. Trata dos módulos de um sistema completo, que não é o caso deste trabalho. Suas funcionalidades são as mesma já descritas anteriormente. Abaixo na Figura 7, está a representação da

arquitetura implementada. As elipses sobre os módulos, indicam quais foram implementados.

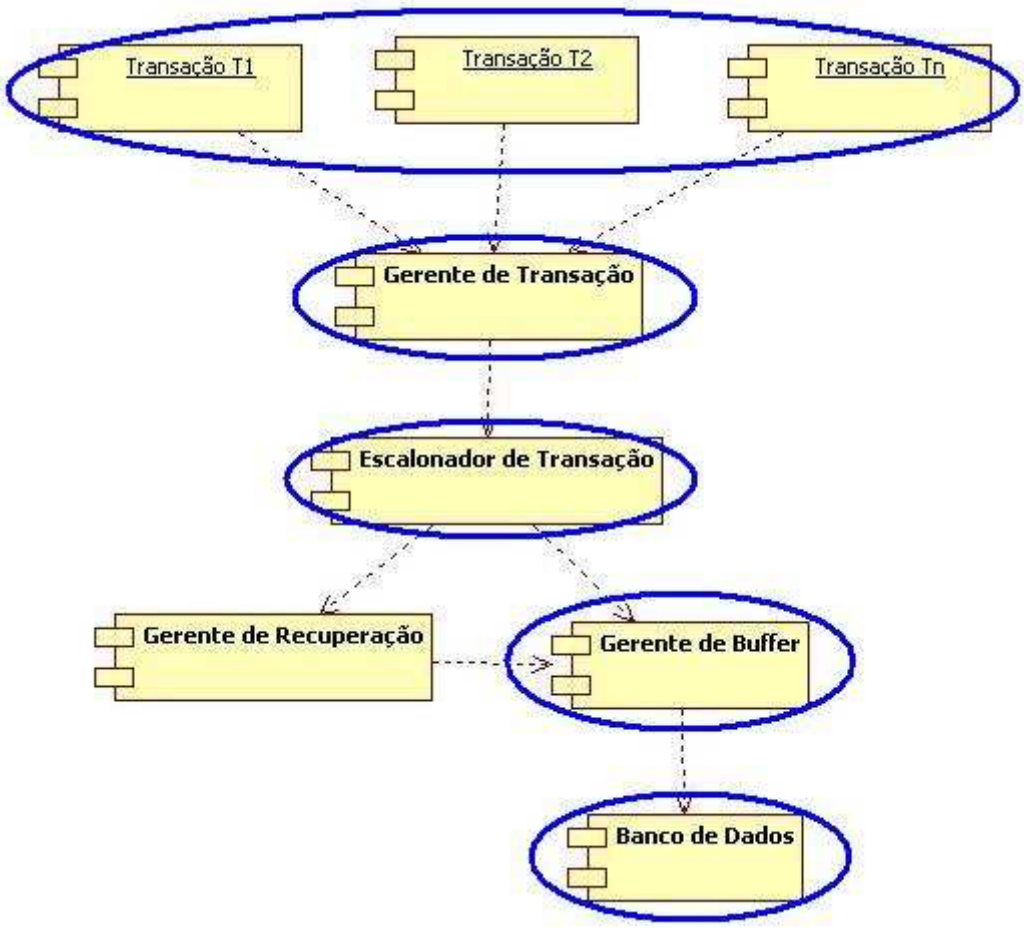

Figura 7: Arquitetura Implementada

Já que a persistência de objetos vem para facilitar a manipulação de objetos que precisam ser persistidos em uma base de dados, nada mais interessante que demonstrar a relação entre o software aplicativo e a API padrão chave-valor para persistir seus objetos. Na Figura 8, representamos esta relação, e na Figura 9, mostro um exemplo codificado.

Figura 8: Relação Cliente com a API

Na Figura 8, o ator Aplicativo Cliente, é o software que vai utilizar a API para persistir objetos, o elemento Rede representa os subsistemas de rede sob protocolo TCP/IP e por fim API-*KeyValue*, a API do trabalho proposto.

Para ficar mais claro vamos simular, por meio da Figura 9, uma interface da API no lado do cliente comunicando com a API que persiste os objetos no lado do servidor. Esta arquitetura é definida em duas camadas. Uma camada é representada pelo sistema do lado do cliente, que por exemplo, pode ser o sistema do nosso exemplo na Figura 9, e a outra camada é o sistema que é executado no servidor, o SGBD propriamente dito.

```
public class Aplicativo {

    public static void main(String args[]){

        DBM dbm = new DBM("Db1", "Server1", 5000);

        dbm.insere("123", new MinhaClasse());

        System.out.println(dbm.busca("123"));

        dbm.delete("123");

        dbm.beginTrans();

        dbm.insere("125", new MinhaClasse());
        dbm.insere("124", new MinhaClasse());

        dbm.commit();

    }

}
```

Figura 9: Exemplo de Aplicativo

É importante entender que esta sugestão de interface do aplicativo do cliente com a API, não foram implementadas, pois, seria uma interface do aplicativo com as interfaces do sistema de armazenamento dos dados. Aquilo que está internamente na classe DBM do exemplo da Figura 10, compõe os subsistemas para persistência de objetos. Uma sugestão para a comunicação do cliente com a API chave-valor, seria a utilização de Sockets, transferindo pela rede um objeto que contenha um código da operação, chave, valor, fonte

lógica e identificador da transação. A fonte lógica, é a separação lógica para ser aplicada sobre os dados.

A estrutura de dados para indexar as informações e organizar os dados foi a árvore B+. Em função dela, o escalonador de transações implementa o protocolo baseado em bloqueios de árvore.

O gerente de *buffer* utiliza a politica de LRU para troca de páginas trabalhando com blocos de 4096 bytes. Os bloqueios são realizados em nível de bloco e não a nível de item de dado, pois, as dependência no sistema são sobre o bloco. Um bloco pode conter n itens de dados e apenas uma página da árvore B+.

Os bloqueios sobre os objetos acessados concorrentemente foram implementados com os métodos da classe *Thread*, sincronizadores ou semáforos.

4.1 REQUISITOS DO SISTEMA

Os requisitos do sistema estão baseados nas propriedades que um sistema de armazenamento de banco de dados, deve possuir para persistir seus dados. Dessa forma, como já tratamos deste assunto no capítulo 2, teremos mais facilidade para entende-lo. São eles:

1. Poder armazenar e recuperar dados por meio de acesso concorrente, várias transações acessando um mesmo item de dado, ou seja, apesar da transação estar sendo executada em um ambiente com acesso concorrente, seus efeitos sobre os itens de dados devem ser os mesmos caso o dado fosse processado por uma única transação(princípio da correção).

2. Realizar a leitura de dados com uma performace na ordem de complexidade dada pela expressão da Figura 10. As variável n representa o número de itens de dado e a variável t representa a ordem da estrutura de dados árvore B+.

Figura 10: Complexidade da Leitura

3. Antes de serem executadas as operações de modificação e leitura na base de dados, a base de dados estará em um estado consistente. Após realizar as operações de leitura e escrita sobre a base de dados, o banco de dados deve permanecer em um estado consistente.

4. Os dados gravados por transações consolidados devem ser permanentes ou em caso de falha e não haver possibilidade de recuperação, deve retornar a base de dados ao estado anterior às modificações. Não pode haver modificações parciais, devem ser completas ou totalmente desfeitas.

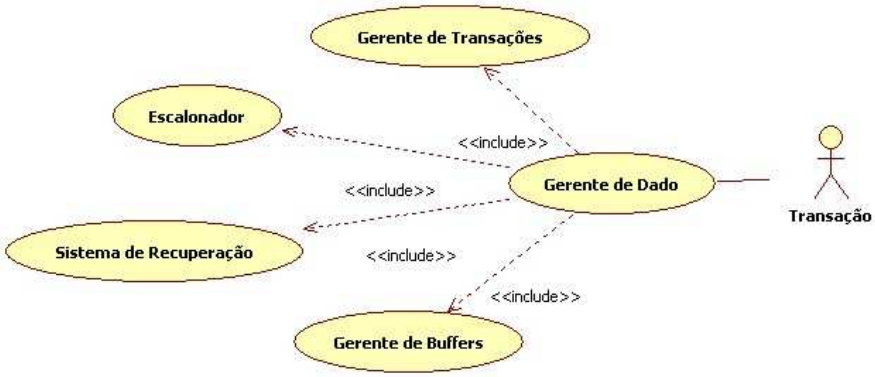

Figura 11: Caso de Uso

Antes de apresentar o casos de uso textual da Figura 11, iremos listar os requisitos que farão parte do caso de uso.

Tabela 6: Caso de Uso Textual

Caso de Uso: Gerente de Dados
Referência: Requisitos dos itens a, b, c e d acima mencionados.
Ator: Transação
Objetivo: Realizar operações de leitura, inclusão, exclusão e modificação de dados em qualquer tipo de objeto instanciado na linguagem de programação Java. Estas operações serão solicitadas pelo ator Transação e obterão resposta da operação com ou sem sucesso.
Descrição: A Transação será criada pela biblioteca, através de um produtor de objetos que vai conceder um bloqueio ao item de dado desejado. Com este bloqueio o escalonador das transações terá condição de definir interações válidas entre várias transações sobre um mesmo recurso. Os processamentos de transações que puderem ser executadas concorrentemente serão aproveitados para aumentar o nível de concorrência e assim dar uma resposta para cada transação em um tempo menor. Cada transação vai oferecer operações de leitura, criação, exclusão e modificação de itens de dado. A utilização destas operações deve ser transparente, ou seja, a complexidade que estar operações contem, deve ser oculta para que as utiliza.
Fluxo: 1. Produzir uma nova transação. 2. Realizar operações de leitura, criação, exclusão e modificação de itens de dados. 3. Consolidar ou abortar uma transação.
Casos de Teste: 1. Criar várias transações e executar suas operações sobre um mesmo item de dado. 2. Consolidar transação e recuperar os dados depois. 3. Realizar escritas e leitura de dados e medir performace por meio de medições de tempo para um senário com várias transações. Obter o tempo de resposta para cada transação.

4.2 MODELO CONCEITUAL

Na Figura 12, apresentamos um modelo conceitual das classes do sistema.

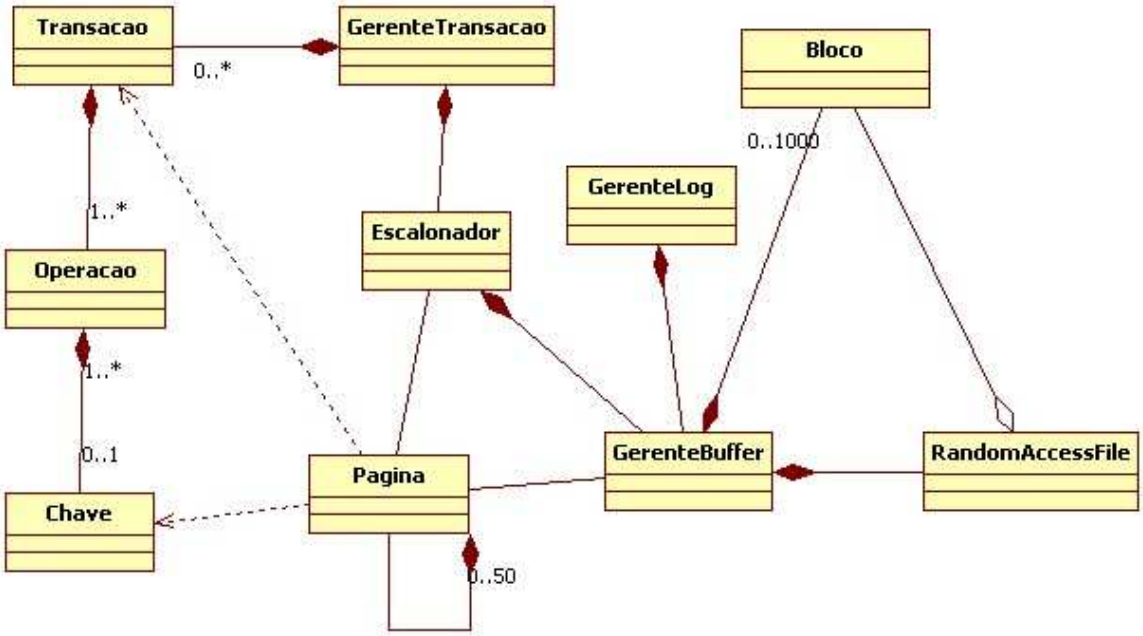

Figura 12: Modelo Conceitual

Este modelo simplificado possui as principais classes do sistema. Com ele podemos ter uma visão global do sistema.

A classe Transação será responsável pela definição do escopo de uma transação, contendo um conjunto de objetos Operação. A classe Operação é uma abstração da operações de uma transação, nela estão representados as operações de leitura, escrita, exclusão, modificação dos dados, inicio de transação, *commit* e *rollback*. Dentro desta classe Operação existe um identificador para cada tipo de operação, um buffer dos dados e da chave.

A classe Chave define um modelo de identificação de um dado, este objeto identificada um dado do usuário através do identificador que ele atribui para o dado, mais um identificador de divisão lógica, que acaba sendo um prefixo da chave da arvore B+.

O objeto Escalonador recebe chamadas de seus método através dos objetos Transação. Cada objeto Transação acessa concorrentemente o objeto Escalonador, porém, a execução de cada operação pertencente a um objeto transação é sequencial, pois, neste modelo o objeto Transação representa as operações do usuário. Em um sistema completo isso não poderia ser considerado uma verdade, pois, um cliente poderia possuir um conjunto de transações que deveriam ser executadas sincronamente, e cada conjunto dessas transações, é que deveria concorrer paralelamente no escalonador. Para simplificar, deixamos apenas o objeto transação

com seu papel mais o do cliente.

A classe Pagina é o nodo da arvore B+. Todo acesso ao arquivo seja para nodos de índice ou blocos de dados serão por realizados por meio do objeto Página. Uma página poderá possuir até 50 chaves e consequentemente 51 filhos, para não ocupar mais que 4065 bytes.

A classe GerenteBuffer, representa o sistema gerenciador de *buffers*. Ela é responsável por alocar objetos bloco da classe Bloco. Cada bloco é uma estrutura de dado *slotted-page* que armazena qualquer tipo de dado, seja ele do usuário, do índice ou do log.

E por fim, temos a classe GerenteLog que é responsável por realizar os registros de log. Nela vão existir operações para recuperar um valor velho de um bloco e gravar novos blocos. Como não está sendo implementado tudo o que contemplaria um sistema de recuperação, esta classe vai ficar limitada ao seu conjunto de funcionalidades.

5 IMPLEMENTAÇÃO

Neste capítulo, vamos abordar as soluções tomadas para implementar o sistema, ou seja, codifica-lo na linguagem de programação Java, compilador 6.0, biblioteca do JDK 1.6.0_14, sistema operacional Microsoft Windows XP 2002 em um microcomputador com 512MB de memória RAM e processador ADM Sempron +2800 com 1,99GHz. Além da definição das abordagens utilizadas para codificar o sistema, vou descrever também alguns dos problemas ocorridos durante a implementação do sistema, bem como os caminhos tomados para resolve-los.

O ambiente de desenvolvimento utilizado para codificação do sistema foi o Eclipse, porém, nada impede a utilização de qualquer outro editor de texto no padrão ASCII para editar os fontes, pois não existe nenhum outro arquivo de configuração necessário.

Inicialmente vamos definir os blocos de dados, bem como o formato de cada um, para já podermos relacionar as principais variáveis envolvidas no sistema. Da mesma forma que vai existir a representação dos dados por meio dos blocos de dados no disco, vão existir representações correspondentes destas estruturas na memória principal através das classes da Figura 12.

5.1 BLOCO DE DADOS

O bloco de dados é a estrutura para realizar a gravação dos dados. Este tipo de estrutura é também chamada de *slotted-page* (SILBERSCHATZ, 2008, p.316). Este bloco de dado será a estrutura que o gerente de *buffer* irá empregar para gravar a página da arvore B+ e o registro de log. O campo 9 da Tabela 7, será a área de dados que vai conter empacotado os blocos de dados do usuário, da arvore B+ e do log. Nas próximas seções descrevemos o formato de cada um destes blocos.

Tabela 7: Bloco de Dados

Ordem	Descrição	Espaço (bytes)
1	Indica o tipo de dado do bloco. Pode ser um tipo índice da arvore B+, um bloco de dados do usuário ou registro de log.	1
2	Número de entradas no bloco.	2
3	Deslocamento para o fim do espaço livre no bloco.	2
4	Indicador se o bloco é fragmentado	1
5	Indicador de posição. Valor zero(0) indica que o bloco fragmentado é o primeiro, valor um(1) indica que ele é o ultimo. Usado no log.	1
6	Ponteiro para o próximo bloco	8
7	Ponteiro para o bloco anterior	8
8	Código de verificação de erro	8
9	Área de dados. Cada entrada de dados é acompanhado de um ponteiro de 2 bytes com o deslocamento dentro do bloco e outro campo com o tamanho também usando 2 bytes.	4065

Para referenciar este bloco será necessário o endereço físico do bloco. Para referenciar um dado é necessário então, o endereço fisco do bloco mais o deslocamento dentro do bloco que indica o local da informação da entrada deste dado. O bloco de dados tem grande importância na *performace* e fragmentação do espaço alocado no disco. Podemos citar as seguintes vantagens do bloco acima descrito:

1. Diminuir a fragmentação do espaço alocado no disco.
2. Oferecer flexibilidade para reorganizar um arquivo de dados e assim, eliminar a fragmentação existente.
3. Oferecer *performace* de leitura e escrita no disco, pois, o bloco será tratado sempre como a menor unidade para leitura e gravação. Seu tamanho deve ser dimensionado em função do tamanho dos blocos que o sistema operacional utiliza. Os blocos sempre devem possuir tamanhos que sejam múltiplos inteiros destes dos blocos do sistema operacional (HECTOR; ULLMAN; WIDOM, 2001, p.41).

5.2 BLOCO DA ARVORE B+

O bloco de dado da arvore B+, contém a representação dos dados para uma página da

árvore. Da mesma forma que um bloco de dados, seu tamanho também é importante que seja múltiplo de inteiros do tamanho do bloco que o sistema operacional utiliza para gravar dados no disco. No nosso caso, a classe Pagina será serializada e seu vetor de dados vai ser uma entrada única dentro de um bloco apresentado na seção 5.2. Consequentemente, um objeto Pagina serializado não poderá possuir mais que 4065 bytes.

Tabela 8: Bloco da Arvore B+

Ordem	Descrição	Espaço (bytes)
1	*Flag*, indicando se a página é uma folha.	1
2	Quantidade de chaves na página.	2
3	Ponteiro para o bloco onde reside o bloco contendo a página pai. Se não possui pai, o valor será -1.	8
4	Deslocamento para a entrada que indica a posição da página pai dentro do bloco. Se não possui pai, o valor será -1.	2
5	Vetor de bytes que representa o objeto Chave serializado. Isso vai possibilitar que a arvore B+ possua 31 chaves em sua página.	120
6	Ponteiro para o bloco que contém a página da subárvore com chaves menores que a do campo 5.	8
7	Deslocamento para e entrada da página mencionada no campo 6.	2

Na Tabela 8, apresentamos o bloco de dados para os dados da página ou nodo de uma árvore B+. Foi tomado o cuidado de pode selecionar os tamanhos dos dados o menor possível para não haver desperdício de espaço ocupado no disco.

5.3 BLOCO DO LOG

Este bloco de dados da Tabela 9, vai representar uma sugestão para o registro dos dados do log para o sistema de recuperação. Este bloco vai conter no campo C1 valores pré-definidos para representar o tipo de registro do *log*. Por isso, na Tabela 10 é representado o conjunto de valores e sua descrição para serem aplicados ao campo C1 da Tabela 9. Dessa forma podemos saber qual a dimensão dos possíveis valores do campo C1 da Tabela 9. Apesar do sistema de recuperação não ter sido implementado neste trabalho, foi colocado o bloco de

dados do log para efeito de estudo e implementação das outras partes do sistema relacionadas a ele.

Tabela 9: Campos do Registro de Log

ID	Descrição	Tamanho (bytes)
C1	Tipo do registro de log, conforme a Tabela 10.	1 byte
C2	Tamanho da string do nome do arquivo de destino dos dados.	1
C3	Nome do arquivo físico com extensão para indicar o destino dos dados.	Até 20 bytes
C4	Apontador para o bloco físico de dados.	8
C5	Deslocamento dentro do bloco apontado por C4.	2
C6	Tipo de bloco de dados. Zero(0) para bloco da árvore B+ e um(1) para para bloco de dados.	1
C8	Identificador da transação	4
C9	Indicador se os dados relacionados ao item de dado é velho ou novo. Quando conter valor zero(0), indica valor velho, um(1) indica novo valor. Será considerado como dados para o controle do registro desfazer/refazer o vetor dos dados do objeto, campo C4 e campo C5.	1

Na Tabela 10, cada linha representa um valor que será aplicado ao campo C1 descrito na Tabela 9.

Tabela 10: Tipos de Registro do Log

ID	Valor	Descrição	Tamanho (byte)
A	1	*Start*, inicio de uma transação.	1
B	2	Transação consolidada.	1
C	3	Transação abortada.	1
D	4	Registro desfazer/refazer	1
E	5	Registro inicio do ponto de verificação.	1
F	6	Registro de fim do ponto de verificação.	1

5.4 ESTRUTURAS E SISTEMAS PARA PROCESSAMENTO CONCORRENTE DE TRANSAÇÕES E RECUPERAÇÃO DE DADOS

Conforme foi visto no capítulo 2, iremos nesta seção abordar de forma prática a nível de projeto, descrevendo o que será realizado para que cada um dos conceitos sobre processamento de transação concorrente e sistema de recuperação possam ser implementados. Para as partes mais importantes irei abordar os algoritmos ou padrões de projeto abordados para a codificação do sistema. Demais detalhes serão possíveis identificar no digrama de classes ou no código do sistema.

Até o momento, as estruturas de dados para memória secundária e memória principal, foram abordadas pelos formatos dos blocos e diagrama de classes respectivamente. Esta divisão entre estruturas para memória principal e memória secundária praticamente delimitam também todo o assunto que será tratado em cada capítulo. O processamento de transações concorrentes irá tratar apenas das estruturas em memória principal, já o sistema de *buffer* e recuperação vai abordar questões envolvendo estruturas de dados para o disco. Estas duas áreas de armazenamento definem os algoritmos implementados.

5.5 ESCALONADOR DE TRANSAÇÕES CONCORRENTES

O escalonador de transações concorrentes tem como objetivo definir uma escala válida de ordem de execução das operações entre várias transações, assim como já foi explicado na seção 2.2. Este sistema tem basicamente quatro componentes importantes. Estes componentes foram modelados em uma ou mais classes. Os componentes podem ser divididos em Gerente de Transações, Escalonador de Transações, a Transação e o item de dado que vai estar contido dentro de um objeto da classe Operação.

O Gerente de Transação concorrente é responsável pela criação de objetos Transação. Cada transação é criada quando o usuário da API emitir uma operação sobre a base de dados indicando que uma transação foi iniciada(operação *START*). O Gerente de Transação recebe as solicitações de transação por meio da chamadas de métodos realizadas pelo usuário, ou seja, pelo programa do usuário, que usa a API. Estas chamadas de métodos são representadas pelo objeto Operação. O objeto Transação criado, contem informações como as operações que estão sendo realizadas sobre o dado, um identificador da transação e o estado atual. Depois

que uma transação foi iniciada, a transação acessa concorrentemente o objeto Escalonador, cada objeto Transação é uma linha de execução diferente da linha de execução do escalonador e gerente de *buffer.* Para que isso seja possível um objeto Transação vai herdar da classe *Thread* da API da linguagem de programação Java. O retorno de cada operação será escrito no próprio objeto Operação, podendo ser enviado para o cliente por *Socket*. Quando o cliente receber o retorno em seu *Socket*, a linha de execução do cliente continua a execução e retorna o sucesso ou cancelamento da operação lendo o conteúdo do objeto Operação.

Na Figura 13, está a representação do processo de inserção de dados, conforme o descrevemos.

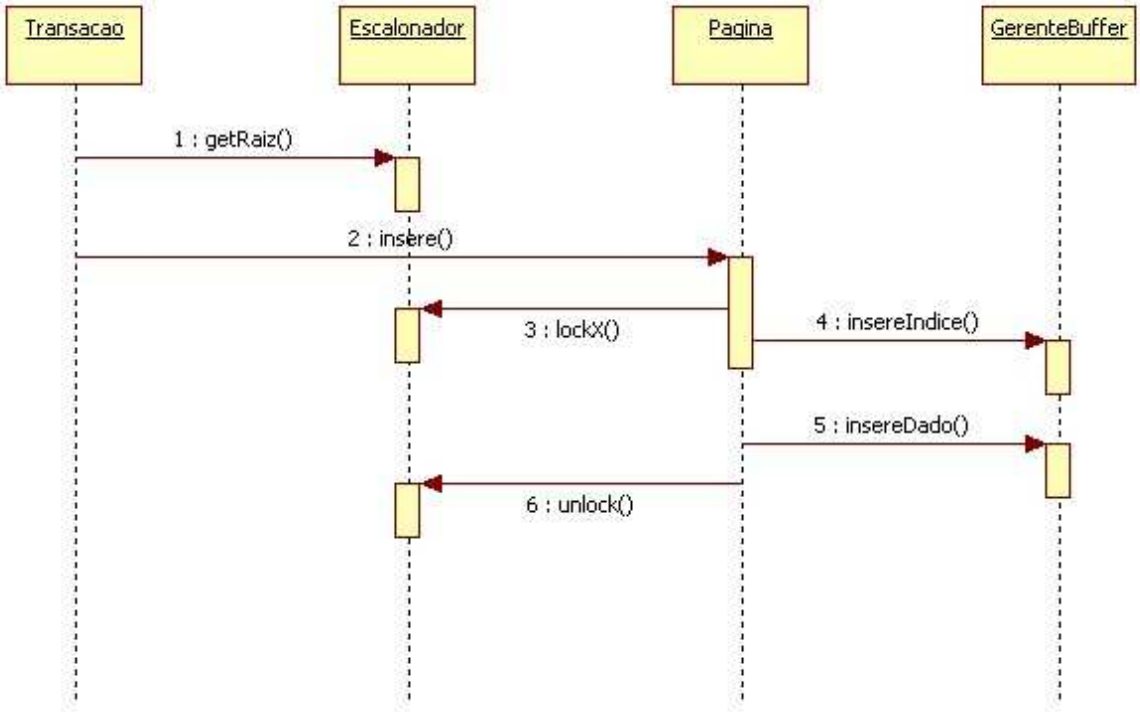

Figura 13: Diagrama de Sequência para Inserir Dados

5.5.1 ESCALONADOR POR BLOQUEIO

Para o escalonamento foi escolhido um protocolo baseado em bloqueios, mas não o bloqueio em duas fases, pois o mesmo tornaria inviável a concorrência sobre a estrutura de dado da árvore B+. Da mesma forma o protocolo em *timestamp* não poderia sempre manter

bloqueada a raiz em uma inserção, pois haveria uma perda de escalonamento de transações concorrentes aos dados. Sendo assim, utilizamos o protocolo de árvore para escalonar as operações das transações. Este protocolo utiliza as características do relacionamento entre os nodos para aplicar regras de bloqueio sobre cada nodo. Ele garante a seriabilidade de conflito, evita *deadlock* e oferece um escalonamento aceitável para acesso concorrente aos dados. Foi provado que o protocolo em árvore evita *deadlock* por Bayer e Schkolnick (1977) (ZAVIANI, 1993, p.184).

Uma transação é uma *Thread*, ou seja, herda da classe *Thread* e assim torna-se uma linha de execução que concorre o processador com outras transações. O controle exclusivo sobre uma estrutura de dados é garantido com a implementação de métodos *synchronized*, que garantem que apenas uma *Thread* poderá tomar um objeto e executar o método solicitado. Um ponto muito importante é que ao ser implementado métodos *synchronized* em um objeto Java, nenhuma outra *Thread* poderá tomar este objeto até que a execução do método atual seja concluída. Baseado nisso, o escalonador deve gastar o mínimo de tempo de processamento para que a execução de seu método seja concluída. Sabemos que a função do escalonador é fornecer uma escala válida para as transações operarem sobre o gerente de *buffer*. Com esta função, o escalonador será um gargalo no sistema, porém, ele é necessário para controle das escalas de processamento das transações. Em um sistema completamente pronto, arquitetura cliente servidor, os serviços de um SGBD, realizam este processamento por meio de um serviço que fica em permanente execução. Realizando um comparativo, um serviços executaria o objeto escalonador e gerente de buffer. Como não foi implementado a comunicação com o cliente, não termos esta parte implementada no trabalho. Basicamente seriam trocas de mensagens entre um *Socket* Cliente e um *Socket* Servidor.

Para implementar o processamento paralelo das transações, foi criado um objeto da classe Escalonador, que vai possuir uma lista, indexada pelos ponteiros dos blocos de dados para o disco, contendo anexado a ele, a informação do tipo de *lock* que a transação necessita conforme as regras de processamento concorrente baseado em bloqueio, ou seja, operações que modificam dados, atribui-se um bloqueio exclusivo para o bloco e nas operações de leitura, um bloqueio compartilhado. O *lock* foi definido sobre o bloco de dados em função da página de índice estar ocupando todo o espaço de um bloco e por haver dependência entre os dados de uma página nas folhas. Um bloco pode ser dividido, criando dois novos blocos, cada um com a metade dos dados do bloco original(algoritmo da arvore B+). Dessa forma, o

bloqueio sobre blocos de dados traz mais beneficio que um bloqueio individual sobre cada item de dado.

A grande questão que fica é, qual o processamento que o escalonador deve realizar. Bem, voltando as notas bibliográficas, temos que a função do escalonador é controlar os acessos concorrentes sobre os blocos de dados e que as transações possuem um *buffer* local para processamento de seus dados. Com isso, praticamente temos as resposta, apesar de não ser tão obvia. Todo acesso e processamento para os dados está sendo realizado através de uma árvore B+. Quando uma transação necessita realizar uma operação qualquer, é solicitado para o escalonador um novo objeto raiz da árvore B+, atribuindo ao seu ponteiro um *lock* correspondente a operação emitida pela transação. Então o único processamento que o objeto Escalonador deve realizar é pedir para o gerente de *buffer* um novo objeto raiz e atribuir a ele um *lock* compatível com a operação solicitada pela transação. Convencionei que o bloco raiz sempre vai estar no bloco zero do arquivo(inicio do arquivo). Assim, o gerente de *buffer* sempre sabe onde buscar a raiz. Estando a transação de posse de uma página raiz, ela tem condições de obter todas as outras páginas e implementar o processamento do algoritmo da árvore B+. Cada página contém os ponteiro físicos para o disco das suas página filhas. Se uma transação necessita acessar um destes blocos, ela pede para o escalonador atribuir-lhe um *lock* e só depois de permitido, pede para o gerente de *buffer* a página. Uma transação pode precisar esperar a obtenção de uma página. Esta espera ocorre no momento que é solicitado para o escalonador a atribuição do *lock* na página interessada. O controle é sempre realizado pelo valor do ponteiro que aponta para o bloco físico dos dados.

Esta solução no inicio não havia sido obtida, pois estava sendo implementado erroneamente todo o processamento da árvore B+ no objeto escalonador. Como os métodos eram declarados com *synchronized*, o objeto escalonador ficava bloqueado até o processamento da árvore B+ ser concluído, tornando-se assim, um processamento sequencial.

A página de uma árvore B+ é sinônimo de nodo. Para cada página da árvore B+ é reservado um bloco inteiro de dados, descrito na Tabela 7. Além dos blocos de índice existem também os blocos de dados, estes irão ocupar o espaço de um bloco de dados da Tabela 7 conforme houver disponibilidade de espaço, ou seja, pode existir, um ou mais itens de dados em um bloco de dado.

A estrutura de dados que mantem a lista de blocos bloqueados no Escalonador, é uma tabela *Hash(java.util.HashMap)*, da API padrão do Java, que gera seu código sobre um objeto

que contem o ponteiro físico para o bloco. A tabela *Hash* tem o objetivo de dar alta *performace* para localização dos itens de dados na memória, situação essa, que é muito interessante para diminuir o tempo que o objeto escalonador fica preso para uma transação, pois, o ponteiro para localização física do bloco possui identidade única, evitando assim, conflitos e perda de *performace*.

Inicialmente imaginou-se utilizar o protocolo em *timestamp*, porém, devido a grande contenção de bloqueio sobre o nó raiz, e assim um grande número de transações revertidas, optou-se pelo protocolo em árvore levando em consideração os seguintes pontos:

1. Processamento de arquivos organizado em estrutura de arvore B+ gera uma grande contenção de bloqueios nos nós ou páginas que ficam no sentido das folhas para a raiz(de baixo para cima) se aplicado bloqueio de duas fazes ou *timestamp*.
2. Protocolo pessimista, que já considera em seu funcionamento, problemas que poderiam abortar uma transação.
3. Simplicidade de funcionamento e por conseguinte de implementação, pois, funciona sobre as características da árvore B+ e garante seriabilidade de conflito.
4. Evita *deadlock*
5. Atrasa transações e não as reverte como no *timestamp*.

Quando uma transação fica de posse da página raiz da árvore B+, esta página tem escopo exclusivo da quela transação, mesmo que outras transações possam estar obtendo a mesma página raiz durante o processamento da transação atual, pois cada uma vai ter um endereço de memória diferente, implementando o *buffer* local dos dados para realizar seu processamento. Todo processamento dos dados é realizado sobre o objeto Pagina, que representa o nodo da árvore B+ e os nodos de itens de dados. O controle para acesso as páginas da árvore B+ é dado pelo objeto Escalonador que é compartilhado para todas as *Threads* do tipo Transação, tornando-se assim a seção crítica da transações. Caso os objetos Página fossem reutilizados entre transações diferentes, poderia ocorrer *deadlock* em função dos métodos *synchronized* que existem no objeto Pagina. Para uma página incluir uma nova Pagina filha ou para ler uma Página filha do disco, a página atual deve solicitar para o gerente de *buffer* a ação desejada. Para que isso seja possível as operações de leitura e escrita do gerente de *buffer* sempre vão receber ponteiros para o disco ou, alocar novos ponteiros para inclusão de nova páginas.

Durante o processamento da árvore B+ foi tomado o cuidado para que as páginas

solicitadas para o gerente de *buffer* fossem processadas em variáveis temporárias sem haver a formação de uma lista encadeada, pois, caso isso ocorresse, o coletor de lixo do Java, não teria condições de desalocar espaço para objetos Pagina já processados, acumulando assim espaço da memória principal. Um bom exemplo disso está no trecho de código do método que percorre a árvore B+ na Figura 14.

```
1034    public Pagina busca(Chave k){
1035        Pagina atual = null, anterior = null;
1036        atual = this;
1037        anterior = null;
1038        lockS();
1039        while(!atual.folha){
1040            anterior = atual;
1041
1042            Entry<Chave,Pagina> e = atual.map.higherEntry(k);
1043            if(e!=null){
1044                atual = e.getValue();
1045            }
1046            else{
1047                atual = atual.maiores;
1048            }
1049
1050            atual = gb.read(atual.bloco, atual.desloca,true);
1051
1052            atual.lockS();
1053            if(!atual.isFolha())
1054                anterior.unlock();
1055        }
1056        atual.unlock();
1057        if(anterior!=null)
1058            anterior.unlock();
1059
1060        Pagina ret = atual.map.get(k);
1061        if(ret == null)
1062            return null;
1063        ret = gb.read(ret.bloco, ret.desloca, true);
1064        return ret;
1065    }
```

Figura 14: Busca na Arvore B+

Note que na Figura 14, gb é o objeto Gerente de *Buffer*. O método *gb.read*, retorna uma página da arvore B+. As variáveis que referenciam o retorno do método *gb.read* são referencias temporária com escopo do método, e não formam uma lista encadeada, pois, não contém referências para outros objetos Pagina. Esta característica do código da Figura 14, fazem que seja possível ler qualquer quantidade de dados sem haver estouro de memória

heap. Caso esta implementação fosse realizada com uma linguagem de programação que não contivesse coletor de lixo, o cuidado seria o mesmo, com um adicional que deveríamos ter no código para liberar memória alocada. Na Figura 15, mostro um caso impróprio. Os retângulo representam os blocos de dados e os círculos as paginas da árvore B+. Caso o gerente de *buffer* remova o bloco de dado B2, todas as referências apontadas por ele iria ser mantidas, conforme a representação pelas linhas que conectam os objetos.

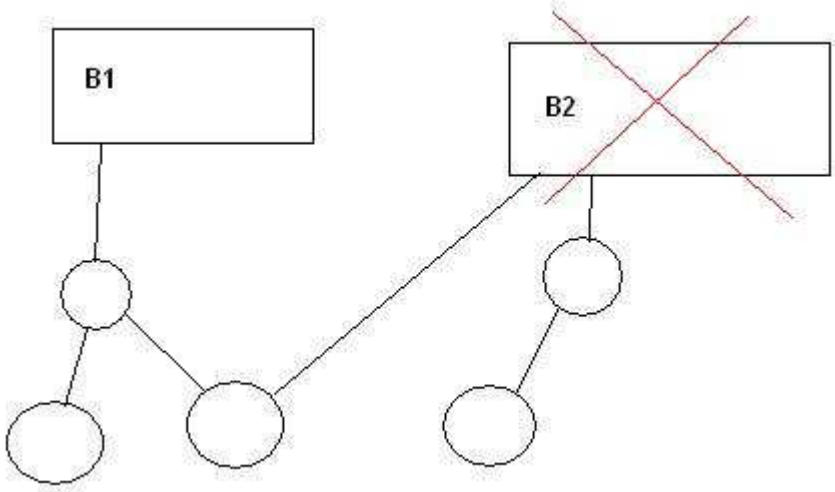

Figura 15: Exemplo Mantendo Referencia

5.5.2 PROBLEMAS COM *DEADLOCK*

Até o momento tudo o que foi descrito tem implementação para evitar o *deadlock*. Durante a implementação é muito fácil cometer erros de programação que gerem *deadlock*. Por exemplo, ocorreu problemas de geração de *deadlock* com a utilização de métodos *synchronized*, caminhamento na árvore B+ e a dupla de métodos *wait* e *notify* das *threads*.

O *deadlock* no caminhamento da árvore B+ ocorreu em uma implementação para adição de dados. Quando uma *thread* descia a árvore B+, a cada aquisição de bloqueio exclusivo em uma nova página, era liberado o bloqueio que havia sido adquirido sobre a página pai. Ficava mantido o bloqueio da página pai apenas quando o sistema atingia a página folha. Essa abordagem não funciona, pois gera *deadlock* facilmente quando uma *thread* T1

desce a árvore e outra T2 sobe a árvore em função da divisão de uma página filha. A *thread* T1 que desce a árvore, tenta adquirir bloqueio sobre a página filha P para realizar o caminhamento e outra *thread* T2 tenta adquirir bloqueio sobre página pai, também P, para inserir uma entrada originada da divisão de uma página filha. Quando uma das duas *threads* adquire bloqueio sobre uma página P, por exemplo a *thread* T1, que as duas tem interesse, a outra *thread* T2 vai ficar eternamente esperando o desbloqueio da página, porém ele não vai ocorrer porque a outra *thread* T1 também espera o desbloqueio da página da *thread* concorrente T2, pois, a página da que a *thread* T2 bloqueou é caminho para a *thread* T1 descer.

Com relação aos métodos *wait* e *notify*, deve-se tomar muito cuidado para que sempre no algoritmo exista para uma chamada a *wait* e outra a *notify*. Caso isso não ocorra, o *deadlock* com certeza irá ocorrer, pois a liberação de um *thread* sempre vai depender do término do processamento de outra. Na Figura 15, está um método válido, em funcionamento, do escalonado para adquirir bloqueio sobre a raiz.

```
54  public synchronized void lockX(Long id){
55      while(!permiteLockX(id)){
56          try{
57              wait(5);
58          }
59          catch(InterruptedException e){
60              e.printStackTrace();
61          }
62      }
63
64      bloqueio.put(id, TipoBloqueio.EXCLUSIVO);
65      notifyAll();
66
67  }
```

Figura 16: Exemplo do LockX

Cabe reforçar que apesar de estar sendo implementado com método *notifyAll*, a regra para não haver *deadlock* é a mesma que foi mencionado com a chamada do método *notify*. Estes dois métodos tem funcionamento diferente, porém, se não houver um *notify* ou *notifyAll* para cada *wait* executado, o *deadlock* vai ocorrer da mesma forma. Na linha 57 da Figura 16, o parâmetro com valor cinco para o método *wait*, é um *timeout* para *thread* que estava esperando tornarem-se pronta.

5.6 GERENTE DE *BUFFER*

O Gerente de *Buffer*, responsável por gravar em disco as estruturas de dados da memória principal e de resgata-las do disco para converter novamente em estruturas de dados de memória principal, tem como objetivo, aumentar a *performace* nas operações de I/O, pois, os discos rígidos possuem uma velocidade de leitura e escrita bem menor que a memória principal.

Dois elementos importantes são considerados, politica de substituição de páginas e organização do arquivo. O politica de substituição de páginas, aplicada aos blocos é a LRU (*Least Recently Used*, usado menos recentemente, ou seja, bloco residente que fora acessado a mais tempo, aquele que está mais obsoleto) pois, como a raiz será o bloco mais acessado, é provável que ela sempre esteja na memória principal. Caso os requisitos do protocolo permitam que um bloco possa ser gravado no disco, será selecionada o bloco mais velho do conjunto permitido. A busca dos blocos no disco será realizada por meio da indexação da árvore B+.

A unidade fundamental do gerente de *buffer* é o bloco de dado. Sua formação foi descrita na Tabela 7. Em Java a leitura e escrita dos dados foi realizada com a classes *RandomAccessFile*. Foram abordados duas formas de converter os objetos em um vetor de dados. A primeira, foi realizada implementando-se a interface *Externalizable* na classe Pagina da arvore B+, a segunda forma foi aplicada ao bloco de dados. No bloco de dados não foi utilizado nenhum mecanismo padrão da API de serialização de objetos do Java para representar em bytes do objeto. A API padrão de serialização de objetos do Java, quando serializa um objeto inclui no cabeçalho do vetor de dados informações que identificam aquele dado como pertencente aos dados que foram serializados. Como o bloco de dados do gerente de *buffer* é a estrutura final para ser gravada no disco, não era meu interesse gravar informações de controle da serialização, pois, iriam ocupar espaço desnecessário. O único objeto que é serializado com a API padrão é objeto de dados do usuário. Apesar do objeto Pagina implementar a interface *Externalizable*, informações de cabeçalho são inseridas da mesma forma, tornando este ponto, candidato para uma melhoria futura.

A implementação da classe Bloco implementa o algoritmo de alocação de espaço denominado *slotted-page.* A carga do vetor de dados foi realizada através das operações de deslocamento de *bits* sobre os tipos de dados primitivos da linguagem de programação Java.

Cada bloco contem em seu cabeçalho uma lista com os deslocamentos para as entradas de dados armazenadas. Quando um item de dado de um bloco é referenciado, são necessárias duas informações, a primeira corresponde ao endereço físico do bloco e a segunda corresponde ao deslocamento para localizar a entrada do dado, que vai conter, mais um deslocamento, agora dentro do bloco para o local do dado. Assim, como acontece com os ponteiros na memória principal, este nível de indireção para os dados dentro de um bloco, torna a utilização do bloco flexível, pois, caso seja necessário mover um bloco para outro local no disco, não será necessário nenhum processamento adicional para poder continuar havendo a localização dos dados.

Quando existe a necessidade de transferir dados entre dois dispositivos com velocidades de leitura e escrita bem distintas, deve-se tomar cuidado com possíveis erros que possam ser gerados durante o processo de transferência dos dados. Para um possível erro, está sendo gravado, junto com o bloco de dados, um código de verificação de erros(*Checksum*). O código é gerado sobre o vetor de dados do bloco, através da classe da API do Java, chamada de Adler32. Caso ocorra uma interrupção inesperada na transferência dos dados, este código vai identificar o problema. Para identificar o problema, a cada leitura é gerado um código de *checksum* sobre o vetor de dados lido, considerando que o campo do código de *checksum* seja zero. Ao ter gerado o código ele é comparado com o código existente no arquivo, se eles forem diferentes uma *exception* será lançada e o sistema interrompe todo o processamento. Se o sistema contivesse a implementação do módulo de recuperação, este erro poderia gerar um evento para desfazer todas as transações ativas. De modo equivalente, quando é gravado um dado é gerado um código de *checksum* sobre o vetor de dados do bloco, considerando novamente, que o campo do código de *checksum* seja zero. Foi convencionado usar zero para o campo de código de *checksum* em função do próprio código estar sendo gravado junto no bloco e assim também estar suscetível a um erro. Durante os meus testes, este erro não ocorreu em nenhuma circunstância.

Na seção 5.6.1, abordamos o funcionamento para politica de substituição de páginas LRU. Esta politica influência na forma como é alocado espaço no disco. A saída de blocos para serem gravadas no disco tem sua ordem determinada pelo algoritmo LRU. É considerado que a ordem de escrita no disco de um bloco novo, pode ser diferente daquela na qual os blocos foram criados para receber novos dados. Para alocação de espaço, foi considerado que a cada novo bloco seria reservado espaço a partir da última posição gravada. No exato

momento em que ele é reservado, nada é gravado em disco, já que isso é papel do LRU definir, porém, este espaço é reservado no sentido lógico e os próximos blocos sempre vão considerar o último bloco alocado independente de ter sido uma alocação lógica ou já física(Alocação física é quando o bloco de dados que teve seu espaço reservado já foi escrito no disco). Quando ocorrer a necessidade de gravar o bloco de dados em disco, será alocado todo o espaço necessário até a posição que o bloco atual está necessitando, caso ainda não tenha gravado. Isso faz com que dependendo da forma como os blocos são utilizados eu possa já reservar espaço para muitos outros que foram alocados antes daquele que o gerente de *buffer* escolheu escrever no disco, poupando assim, um tempo que seria necessário para alocação destes outros blocos, caso fossem alocados individualmente. Na Figura 17, temos o exemplo da implementação para alocação de espaço no disco com o método *output*. A referência raf, aponta para um objeto *RandomAccessFile* da API do Java. A linha 117, faz a realocação de espaço no arquivo de dados e a variável “pos”, é o ponteiro para o bloco de dados.

```
104    private void output(long pos, boolean remove){
105        Bloco saida = null;
106        try{
107
108            if(remove){
109                saida = map.remove(pos);
110                lru.remove(pos);
111            }
112            else
113                saida = map.get(pos);
114
115            //Se for novo aloca no fim do arquivo
116            if(raf.length() < pos)
117                raf.setLength(pos);
118
119            raf.seek(pos);
120
121            saida.writeData(raf);
122        }
123        catch(Exception e){
124            e.printStackTrace();
125            System.exit(1);
126        }
127    }
```

Figura 17: Método *Output*

Na linha 121 da Figura 17, a chamada do método saida.*writeData* escreve o bloco, apontado

por saída, no disco.

Todo bloco de dados, está implementado em uma classe chamada de Bloco. As solicitações de dados para o gerente de *buffer* fazem com que os dados sejam desempacotados deste bloco e empacotadas em um objeto Pagina da arvore B+. Para as transações, o gerente de *buffer* sempre está retornando páginas da arvore B+, pois, é nela que será realizado o processamento para a escolha do próximo bloco de dados conforme algoritmo da arvore B+. Para fazer isso, o gerente de *buffer* só precisa saber o ponteiro para o disco no qual está localizado o bloco de dados desejado.

Como não foi implementado o sistema de recuperação, as operações de *commit* e *rollback* não podem ser fornecidas. Para forçar um gravação de dados no disco, foi criado um operação de *flush*, que obriga o gerente de *buffer* a gravar no disco todos os bloco de dados presentes. Esta gravação é realizada sem a remoção dos blocos de dados da memória principal, já que isso, é decidido com a LRU.

5.6.1 POLITICA DE SUBSTITUIÇÃO DE PÁGINA LRU

A estrutura de dados utilizada para empregar a politica LRU será uma lista encadeada com o seguinte algoritmo aplicado(SILBERSCHATZ, 2000, p.223):

1 Para um novo bloco, adiciono ele ao fim da lista. Se a lista é vazia, o novo bloco representa o inicio e o fim.
2 Para ler um bloco, removo o bloco da lista encadeada e adiciono ele ao fim da lista.

Para localizar um elemento da lista LRU, foi utilizado uma tabela *Hash*, indexada pelo ponteiro do bloco de dados da memória secundária. Ao final da execução do algoritmo, a lista encadeada vai possuir os blocos mais velhos no inicio e os mais novos no fim. Como os bloco serão ponteiros em memória, as operações de adição e remoção vão envolver apenas a atribuição de dois ponteiros, tendo um baixo custo computacional.

5.6.2 ARVORE B+

A árvore B+ foi a estrutura de dados escolhida para realizar a indexação e organização dos dados. A árvore B+ será a estrutura utilizada pelo gerente de *buffer*, e claro, também será

criada pelo próprio gerente de *buffer.* O gerente de *buffer* vai empacotar os dados dentro de objetos Pagina e desempacotar objetos Bloco com os dados de objetos Pagina, da mesma forma que os dados do usuário e blocos de registros de log. As características que levaram a sua escolha são as seguintes:

1. O modelo de gerente de dados da API proposta, é baseado em um par chave-valor. Em função disso as operações de busca serão pelo atributo chave, pelo primeiro registro da chave, ultimo registro da chave, intervalo de valor e conjunto ordenado pela chave. A árvore B+ realiza estes tipos de operação com *performace* logarítmica.
2. Para um bloco com até 4096 bytes, será possível obter chaves 31 chaves com 120 bytes cada uma, possibilitando no máximo quatro acessos a disco para encontra um dado em um conjunto de 1000000 de elementos.
3. Fácil implementação para dados de usuário duplicados.
4. Lista de todos os itens pode ser percorrida a partir do primeiro bloco, apenas com um ponteiro de próximo bloco.
5. Os nós folhas permitem eficiente processamento sequencial do arquivo (SILBERSCHATZ, 2006, p.328).

Os dados do usuário, que serão apontados pelas folhas da árvore B+, serão organizados da mesma forma que as páginas ou nodos da árvore B+. Um objeto de dado de um usuário, será limitado a um tamanho de 4065 bytes, pois, é o tamanho máximo que um bloco de 4096 bytes vai possuir livre. O restante do espaço, está reservado para um área denominada cabeçalho do bloco que contém informações sobre o bloco de dados, como por exemplo, os deslocamentos das entradas de dados.

5.6.3 PROCESSAMENTO

O processamento realizado pelo gerente de *buffer* será realizado através do método produtor consumidor(SILBERSCHATZ, ABRAHAM; 2000, p. 134). Isso significa que uma linha de execução vai realizar a leitura e escrita dos blocos. Outras linhas de execução, originada das transações dos clientes poderão aproveitar o processador para solicitar leitura ao gerente de *buffer* enquanto é realizado IO.

A partir da execução do método *start*(), realizada pelo gerente de transação, o objeto

transação possui sua linha de execução paralela a linha de execução do escalonador e do gerente de *buffer.*

5.7 SUGESTÃO PARA O SISTEMA DE RECUPERAÇÃO(NÃO IMPLEMENTADO)

Um dos primeiros problemas que nos deparamos ao implementar um sistema de recuperação está sobre a *performace* necessária para que a operação de recuperação não seja um grande gargalo de processamento. O arquivo do log deve ser processado sequencialmente, bloco por bloco na ordem em que estão no arquivo. Esta ordem é necessária, pois, utilizamos ela para representar a ordem cronológica dos eventos. Conceitualmente são necessárias duas leituras, uma para refazer e outra para desfazer. Um dos problemas está sobre dependência lógica nos tipos de registros que indicam marcações de inicio e fim para representarem a completude de um número qualquer de operações de transações ou um intervalo de tempo onde várias outras operações de transações diferentes podem ter acontecido. Como exemplo, podemos citar o registro *START* da transação e seu *COMMIT* ou *ABORT* e, registros *START CKPT* e *END CKPT* que delimitam um ponto de verificação. Este problema existe em função do simples fato da memória principal ser finita, ou pior que isso, a memória é muitas vezes menor que o disco. É fundamental esta análise, pois, o gerente de *buffer* existe apenas para tratar esta questão. Sendo assim é imprescindível a *bufferização* do log para leituras e escritas.

Já fica mais claro que entre o intervalo de um registro *START* de uma transação Ti e o seu registro *COMMIT* podem haver muitos registros no log, impedindo que a verificação destes registros dependentes, seja realizada com apenas um acesso a disco. Podem ser necessário muitos acessos a disco, porém, como sempre até o momento, precisamos minimizá-los o máximo possível. Uma solução é aproveitar a estrutura de dado de árvore B+ para indexar o log, deixando na chave, uma informação que identifica uma transação e representa a dependência entre tipos de registro do log. Cria-se um índice, que contém na chave, o identificador da transação e um identificador da operação. Com isso as operações de IO para o gerente de *buffer* diminuem ao tentar localizar um bloco *START* ou *COMMIT,* por exemplo, dada uma transação qualquer. Abaixo ilustramos na Tabela 11, um exemplo de valores de uma chave para descrever as operações de busca que o gerente de log necessita.

Tabela 11: Valores de um Índice para o Log

Valor da Chave	ID da Transação	ID da Operação	Descrição
11	1	1	Transação 1, executou um START (1)
21	2	1	Transação 2, executou um START(1)
22	2	2	Transação 2, executou um COMMIT(2)
31	3	1	Transação 3, executou um START(1)
12	1	2	Transação 1, executou um COMMIT(2)
13	1	3	Transação 1, executou um ROLLBACK

Podemos perceber na Tabela 11, que seriam operações simples para um árvore B+ encontrar, por exemplo, o registro *COMMIT* da transação 1. Considerando que os tipos de registros são valores com representações pré-definidas, basta concatenar ao identificador da transação um código de um tipo de registro e realizar a busca. Tornaríamos o log com algo semelhante as bases de dados do usuário.

O método para modificação de registro será a modificação imediata, pois, oferece uma flexibilidade bem maior para o gerente de *buffer* aplicar o processamento dos blocos de dados. Quando haver necessidade de liberar espaço, ele pode executar a operação e se precisar reverter, basta recuperar o valor antigo a gravar de volta para a base de dados.

Vamos descrever o algoritmo, em alto nível, para processamento de um arquivo de log aplicado para a recuperação de reinicio, que é executado quando ocorre uma queda no sistema e ao reinicia-lo, o processamento de recuperação é executado(SILBERSCHATZ, ABRAHAM; 1999, p. 530). Serão construídas duas listas *redo-list* e *undo-list.* A *redo-list* contém as transações a serem refeitas e a *undo-list,* contém as transações a serem desfeitas. Examinamos registro por registro até encontrar o último registro de *checkpoint*, ou melhor dizendo, ponto de verificação:

1. Para cada registro na forma <*COMMIT* Ti>, adicionamos ele a lista refazer.
2. Para cada registro na forma <*START* Ti>, se a transação Ti, não estiver na lista refazer, então adiciona Ti a lista desfazer.

Estas duas listas podem ser processadas usando uma árvore B+, conforme já foi descrito. Quando o processamento para avaliar ou construir as duas listas estiver concluído, executamos os passos finais da recuperação:

1. Examinar o log em ordem cronológica decrescente, a partir do registro de log mais

recente, o final do arquivo, para cada transação Ti da lista inutilizar, executar a operação de *UNDO* até encontrar o registro *START* da transação. As transações da lista refazer, serão ignoradas neste passo.

2. Examinar o log em ordem cronológica crescente até o registro de *checkpoint* mais recente, e para cada transação Ti a partir deste registro que pertence a lista refazer, executar a operação *REDO*(refazer). Ignorar os registro de log das transações da lista desfazer.

É evidente que estas duas listas podem não caber na memória principal, e como já foi mencionado, será necessário submeter o arquivo de log ao gerente de *buffer* para tratar volumes grandes de dados.

6 TESTES

Os testes foram realizados aplicando sobre um número X de transações concorrentes para um mesmo conjunto de dados Y, operações de leitura ou escrita de forma aleatória. Para cada operação em cada transação foi totalizado um tempo e calculado uma média aritmética em relação ao número de operações que foram executadas. No final foi realizado uma média aritmética final apresentada na Tabela 12. Foram utilizados para teste um valor numérico para chave e *string* para o dado. Em Java existe objetos do tipo *String*, implementam na classe *String* a interface *Serializable*, tornado possível a geração do vetor de dados. Desta tabela, podemos perceber o incremento do tempo quando aumentado o nível de concorrência. É evidente que muitos fatores influenciam nestas medidas, desde a máquina utilizada como o comportamento do sistema, principalmente o gerente de *buffer* e o escalonador.

Tabela 12: Tempos de Resposta das Operações

Operação	Quantidade de Transações	Quantidade de Itens	Tempo Médio de Resposta(ms)
Escrita	10	1000	21,53
Leitura	10	1000	5,36
Escrita	20	1000	66,58
Leitura	20	1000	13,83
Escrita	30	1000	71,24
Leitura	30	1000	13,28
Escrita	40	1000	107,76
Leitura	40	1000	25,96
Escrita	50	1000	220,45
Leitura	50	1000	44,52

Na Figura 18, temos um gráfico representando os dados da Tabela 12. No eixo X, existe o número de transações em uma escala de 1 x 10 e no eixo Y, os tempos para escrita em uma escala 1 x 1. Percebe-se que o tempo para a escrita aumenta com o aumento do número de transações de forma não linear.

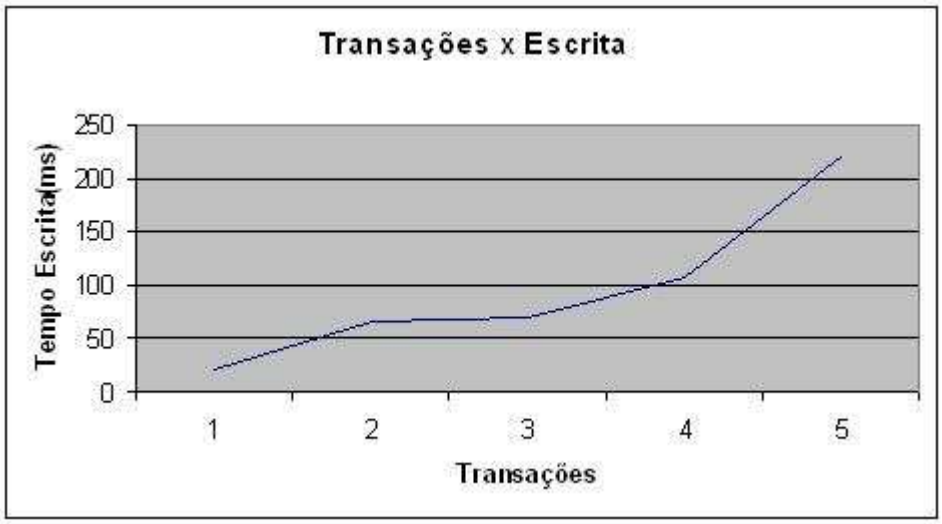

Figura 18: Transações x Escritas

Na Figura 19, existe a representação dos tempos de leitura com o mesmo número de transações. No eixo X o número de transações em uma escala de 1 x 10 e no eixo Y os tempos de leitura em uma escala 1 x 1. Os tempos de leitura também sobem com o número de transações, porém, em quantidades de tempo bem menor que a escrita. Isso pode ser concluído em função da escrita precisar bloquear todos a paginas desde a raiz para realizar a operação e na leitura isso não é necessário.

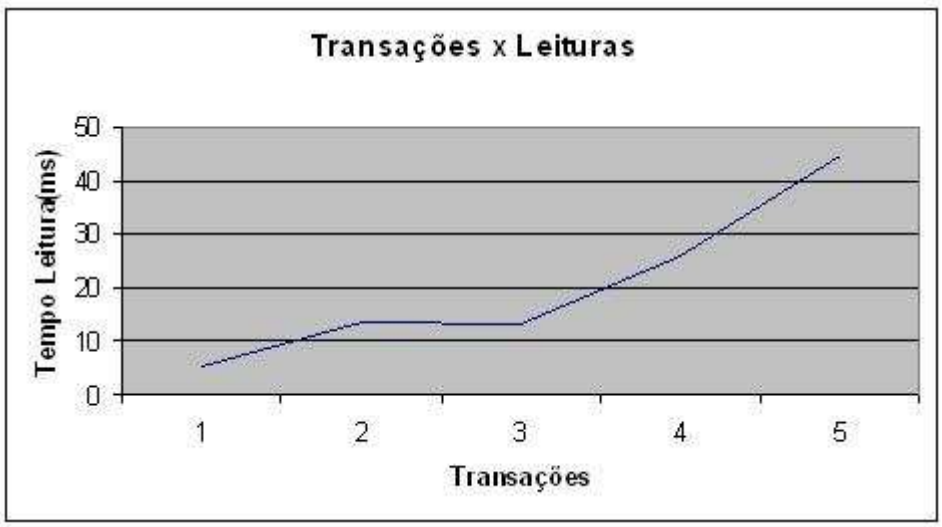

Figura 19: Transações x Leituras

7 CONCLUSÃO

Pode-se concluir que uma API no padrão chave-valor, provê uma base de conhecimento muito interessante para aplicar os conceitos das áreas de banco de dados e sistemas operacionais. Outro ponto importante e que fica evidente é que alguns métodos para escalonamento de transações apresentam dificuldade para realização de seus testes.

Baseado em alguns testes que haviam sido realizados antes da implementação, esperava-se uma performace melhor para a API, mas, como existem muitos itens que podem ser melhorados, seus tempos se justificam. Como exemplo, temos a diferença entre os tempos de leitura e escrita bem como a forma distinta de bloqueio que causa os tempos distintos.

Uma otimização nas formas de bloqueio podem trazer melhorias significativas, visto a discrepância que já existe para leitura e escrita.

A codificação do sistema de bloqueios de árvore trata os *deadlocks*, porém, deve se tomar muito cuidado com questões da codificação para não cometer erros que possam também gerar *deadlock*.

É muito importante projetar o sistema pensando nas estruturas de dados para serem aplicadas em arquivo, pois, uma implementação para memória principal facilmente gera listas encadeadas que não possuem viabilidade para tratar grandes quantidades de dados.

É possível que algum determinado protocolo de bloqueio seja mais interessante em algum contexto de utilização que dependa de questões do ambiente para utilização do sistema, como por exemplo, o número de usuários concorrentes, tamanho de transações, tamanho dos itens de dados, da chave, hardware e sistema operacional.

O modelo de API chave-valor, economiza muito recurso computacional, pois, ele é basicamente composto de apenas uma API para manipular os dados, não precisando de um processador para um linguagem de consulta bem como as conversões que são necessárias para os dados.

O sistema de recuperação e comunicação com o cliente via TCP/IP usando Sockets, poderiam ser outros dois trabalhos de conclusão de curso. Com certeza, com mais alguns trabalhos seria possível obter um sistema de armazenamento completo. No inicio, imaginei que poderia ser implementado parte do sistema de recuperação, porém, não é possível, pois, uma operação de recuperação de falha ou *abort* explicito usam a mesma estrutura de

recuperação, não tendo diferença entre elas em termos de implementação. O único sistema que está relacionado ao sistema de recuperação que foi implementado é o Gerente de *Buffer.*

Seria muito interessante avaliar todos os algoritmos, calculando a complexidade de tempo e espaço de cada um deles, isso incluindo os algoritmos de API externas, como neste caso, a API do Java, principalmente o pacote das coleções. Com esta avaliação poderia-se identificar pontos falhos e melhorar a performace do sistema.

A serialização do Java e os formatos dos dados inteiros de tipos primitivos, consomem grande tempo de processamento. A serialização varre toda a estrutura do objeto, se ele for grande, maior será o tempo para serializa-lo. Imagine que eu tenha um objeto grande e modifique apenas uma variável de todo o objeto, o custo computacional será o mesmo de serializar todo o objeto. O tipos de dado primitivo possuem os bits mais significativos a esquerda(Big-Edian). Se o processador tem esta ordem contrária, ele precisa converter os dados gerados pelo Java. Isso consome tempo de processamento. A implementação de um sistema de armazenamento seria bem mais interessante com uma linguagem de programação como C, C++ ou Pascal, gerando código compilado, código para a máquina física e não para uma máquina virtual.

8 REFERÊNCIAS

BERNSTEIN, Philip A.; HADZILACOS, Vassos; GOODMAN, Nathan. Concurrency Control and Recovery in Database Systems. California: Addison Wesley, 1987.

MOLINA, Garcia H.; ULLMANN, J. D.; WIDOM, J. Implementação de Sistemas de Bancos de Dados. Rio de Janeiro: Campus, 2001.

HORSTMANN, C. S; CORNELL, Gary. Core Java Volume 1. São Paulo: Pearson, 2003.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN S. Sistema de Banco de Dados. São Paulo: Pearson Makron Books, 1999.

ZIVIANI, Nivio. Projeto de Algoritmos. São Paulo: Pioneira, 2002.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN S. Sistema de Banco de Dados. Rio de Janeiro: Elsevier, 2006.

DEITEL, H. M.; DEITEL P. J. Java Como Programar 4º Ed. Porto Alegre: Bookman, 2003.

SILBERSCHATZ, Abraham; GALVIN, Peter; GAGNE, Greg. Sistemas Operacionais. Rio de Janeiro: Campus, 2000.