UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

# JOODBMS – IMPLEMENTAÇÃO DE UM SISTEMA DE PERSISTÊNCIA E RECUPERAÇÃO DE OBJETOS EM JAVA

**ROBSON PANIAGO DE MIRANDA**

**ORIENTADOR: DR. RAFAEL TIMOTEO DE SOUSA JR.**

**DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA**

**PUBLICAÇÃO: ENE.DM - 269/06**

**BRASÍLIA/DF: Julho – 2006**

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

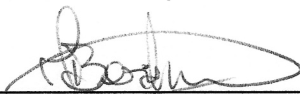
# JOODBMS – IMPLEMENTAÇÃO DE UM SISTEMA DE PERSISTÊNCIA E RECUPERAÇÃO DE OBJETOS EM JAVA

ROBSON PANIAGO DE MIRANDA

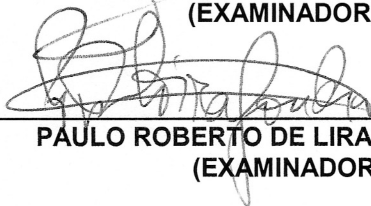
DISSERTAÇÃO DE MESTRADO ACADÊMICO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:

RICARDO STACIARINI PUTTINI, DR., ENE/UnB
(ORIENTADOR)



JACIR LUIZ BORDIM, Dr., CIC/UnB
(EXAMINADOR EXTERNO)



PAULO ROBERTO DE LIRA GONDIM, Dr., ENE/UnB
(EXAMINADOR INTERNO)

BRASÍLIA, 31 DE JULHO DE 2006.

## FICHA CATALOGRÁFICA

MIRANDA, ROBSON PANIAGO.
JOODBMS – Implementação de um Sistema de Persistência e Recuperação de Objetos em Java. [Distrito Federal]
xvi, 130 p., 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2006)
Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Elétrica.

1. Engenharia de Software        2. Orientação a Objetos
3. Bancos de Dados        4. Java
I. ENE/FT/UnB        II. Título (Série)

## REFERÊNCIA BIBLIOGRÁFICA

MIRANDA, R. P. (2006). JOODBMS – Implementação de um Sistema de Persistência e Recuperação de Objetos em Java. Dissertação de Mestrado, Publicação ENE.DM 239/05, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 130 p.

## CESSÃO DE DIREITOS

AUTOR: ROBSON PANIAGO DE MIRANDA
TÍTULO: JOODBMS – Implementação de um Sistema de Persistência e Recuperação de Objetos em Java.
GRAU: Mestre        ANO: 2006

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação de mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. Do mesmo modo, a Universidade de Brasília tem permissão para divulgar esta Dissertação de Mestrado em biblioteca virtual, em formato que permita o acesso via redes de comunicação e a reprodução de cópias, desde que protegida a integridade do conteúdo dessas cópias, inclusive ao que se refere à proibição de acessos a partes isoladas do conteúdo. O autor reserva outros direitos de publicação e nenhuma parte desta dissertação de mestrado pode ser reproduzida sem a autorização por escrito do autor.

________________________________
ROBSON PANIAGO DE MIRANDA
SQN 209 Bloco A Ap. 602. Asa Norte.
CEP: 70854-010 – Brasília/DF – Brasil

Aos meus pais, pela paciência e força

À Vânia, pelo amor e carinho dispensados

A todos aqueles que, direta ou indiretamente,
contribuíram para a conclusão deste trabalho

# AGRADECIMENTOS

Este trabalho não poderia ter sido realizado sem a ajuda de meu orientador, Prof. Dr. Rafael Timóteo, e também sem a colaboração e revisão do Prof. Dr. Ricardo Puttini.

Agradecimentos especiais também a Maristela, pelo apoio dado durante todo o decorrer do trabalho.

À Vânia, pelo apoio e carinho recebidos, por estar sempre ao meu lado nos momentos de maior estresse.

Aos meus colegas e amigos do Ministério Público do DF e Territórios, pela compreensão da extensão deste trabalho de pesquisa.

Aos meus pais, pela paciência que tiveram, e, com certeza, continuarão a ter.

E, sobretudo, a Deus, por permitir a conclusão de mais esta etapa em minha vida.

# RESUMO

## JOODBMS – Implementação de um Banco de Objetos em Java

Este trabalho tem por objetivo mostrar uma implementação de um banco de objetos desenvolvimento completamente na linguagem Java.

Para seu desenvolvimento, foram antes identificadas as características pertinentes aos bancos de dados, com foco principal nos métodos de acesso aos dados, e também nas características para a implementação de um banco de objetos.

O foco principal do desenvolvimento do JOODBMS é a modularização. Através da compartimentalização da funcionalidade em diversos módulos auto-contidos é possível utilizar o JOODBMS como plataforma de estudos, trocando ou acrescentando a implementação dos módulos, mas desde que a interface já definida seja mantida.

O sistema foi completamente desenvolvido na linguagem Java, utilizando-se das bibliotecas CGLIB, para a geração em tempo de execução dos objetos *proxies*, e da Javolution, para o acesso aos dados das páginas de forma estruturada.

Para exemplificar seu funcionamento, uma simples aplicação de cadastro de notas fiscais foi desenvolvida, ilustrando com trechos de código como utilizar o JOODBMS para a persistência de objetos Java.

# ABSTRACT

**JOODBMS – Implementation of an Objectbase in Java**

The purpose of this work is to show an implementation of an objectbase developed in the Java language.

To accomplish that, the first step was to identify the pertinent characteristics of data bases, focusing data access methods, and also the main features needed to implement an objectbase.

The development of JOODBMS is mainly focused in modularization. Through compartimentalization of the functionality in diverse self-contained modules it is possible to use JOODBMS as a platform for research, changing or adding the implementation of the modules, but keeping the already defined interface.

The system was completely developed in the Java language, using the libraries CGLIB to generate the proxy objects at runtime, and Javolution to access the data pages in a structured manner.

To exemplify it's use, a simple billing application was developed as an example of its work, including some lines of code that show how to use JOODBMS to persist java objects.

# SUMÁRIO

1 - Introdução ... 1
2 - Conceitos Teóricos ... 4
2.1 - Princípios de Orientação a Objetos ... 4
2.1.1 - A linguagem Java ... 7
2.2 - Banco de Dados ... 7
2.2.1 - Visão geral de um SGBD ... 8
2.2.2 - Comandos da linguagem de definição de dados (DDL) ... 10
2.2.3 - Processamento de consultas ... 10
2.2.4 - O gerenciador de buffers ... 11
2.2.5 - Processamento de transações ... 11
2.2.6 - O processador de consultas ... 12
2.2.7 - Dados Relacionais e Orientados a Objeto ... 13
2.2.8 - Armazenamento de dados ... 14
2.2.9 - A hierarquia da memória ... 14
2.2.9.1 - Cache ... 15
2.2.9.2 - Memória principal ... 15
2.2.9.3 - Memória virtual ... 15
2.2.10 - Armazenamento secundário ... 16
2.2.11 - Armazenamento volátil e não volátil ... 16
2.2.11.1 - O uso eficiente do espaço de armazenamento secundário ... 17
2.2.12 - O modelo de computação de E/S ... 17
2.2.13 - Classificação de dados em armazenamento secundário ... 18
2.2.14 - Representação de elementos de dados ... 21
2.2.14.1 - *Strings* de caracteres de comprimento fixo ... 21
2.2.14.2 - *Strings* de caracteres de comprimento variável ... 21
2.2.15 - Representação de elementos de bancos de dados relacionais ... 22
2.2.16 - Representação de objetos ... 22
2.2.17 - Registros ... 23
2.2.17.1 - Cabeçalhos de registros ... 23
2.2.17.2 - Armazenamento de registros de tamanho fixo em blocos ... 24
2.2.18 - Representação de endereços de blocos e registros ... 25
2.2.18.1 - Endereços lógicos e estruturados ... 25

2.2.18.2 - Mistura de ponteiros ........ 26
2.2.18.3 - Retorno dos blocos para o disco ........ 29
2.2.18.4 - Registros e blocos fixados ........ 30
2.2.19 - Dados e Registros de Comprimento Variável ........ 30
2.2.19.1 - Registros com campos de comprimento variável ........ 31
2.2.19.2 - Registros com campos repetidos ........ 31
2.2.19.3 - Registros que não se encaixam em um bloco ........ 32
2.2.19.4 - BLOBs ........ 32
2.2.20 - Modificações de Registros ........ 33
2.2.20.1 - Inserção ........ 33
2.2.20.2 - Exclusão ........ 34
2.2.20.3 - Atualização ........ 34
2.3 - Estruturas básicas de arquivo ........ 34
2.3.1 - Buffering de blocos ........ 35
2.3.2 - Alocação de blocos de arquivo em disco ........ 36
2.3.3 - Cabeçalhos de arquivo ........ 37
2.3.4 - Operações em Arquivos ........ 37
2.3.5 - Arquivos de registros desordenados (*heap files*) ........ 38
2.3.6 - Arquivos de Registros Ordenados (*Sorted Files*) ........ 39
2.3.7 - Abordagens mistas ........ 40
2.3.8 - Técnicas de hashing ........ 41
2.3.9 - *Hashing* externo para arquivos em disco ........ 42
2.3.10 - Técnicas de *hashing* que permitem a expansão de arquivos dinâmicos.. 43
2.3.10.1 - *Hashing* extensível ........ 43
2.3.10.2 - *Hashing* linear ........ 44
2.3.11 - Outras organizações primárias ........ 45
2.3.11.1 - Arquivos com Registros Mistos ........ 45
2.4 - Estruturas de Índice ........ 45
2.4.1 - Arquivos Seqüenciais Indexados ........ 46
2.4.1.1 - Arquivos seqüenciais ........ 46
2.4.1.2 - Índices densos ........ 46
2.4.1.3 - Índices esparsos ........ 47
2.4.1.4 - Vários níveis de índice ........ 47
2.4.2 - Índices Secundários ........ 47

2.4.2.1 - Caráter indireto em índices secundários ........ 48
2.4.3 - Operações em um arquivo indexado ........ 48
2.4.3.1 - Adição de registros ........ 49
2.4.3.2 - Exclusão de registros ........ 49
2.4.3.3 - Atualização de registros ........ 49
2.4.4 - Estruturas de arquivo utilizadas em índices ........ 50
2.4.4.1 - Árvore de pesquisa Binária ........ 51
2.4.4.2 - Árvores AVL ........ 52
2.4.4.3 - Árvores Binárias Baseadas em Páginas ........ 53
2.4.4.4 - Árvores B ........ 55
2.4.4.5 - Algoritmo de pesquisa e recuperação ........ 56
2.4.4.6 - Inserção, Divisão e Promoção ........ 57
2.4.4.7 - Exclusão, Redistribuição e Concatenação ........ 58
2.4.4.8 - Registros de tamanho variável ........ 59
2.4.5 - Arquivos seqüenciais indexados ........ 60
2.4.5.1 - Mantendo a seqüência de blocos de forma ordenada ........ 60
2.4.5.2 - Acrescentando um índice à seqüência de blocos ........ 61
2.4.5.3 - As árvores B+ ........ 61
2.5 - Bancos de DADOS ........ 63
2.5.1 - Histórico ........ 63
2.5.1.1 - Bancos Hierárquicos ........ 63
2.5.1.2 - Bancos em rede ........ 64
2.5.1.3 - Bancos Relacionais ........ 64
2.6 - Bancos de Objetos ........ 65
2.6.1 - Perspectiva Histórica ........ 66
2.6.1.1 - Os primeiros sistemas ........ 66
2.6.2 - Características de um banco de objetos ........ 67
2.6.3 - Padronização ........ 68
2.6.4 - Recuperação de dados em bancos de objetos ........ 68
2.7 - Banco de dados Objeto-Relacional ........ 69
2.8 - Modelo de dados orientado a objeto ........ 70
2.8.1 - Identidade de Objeto ........ 70
2.8.2 - Composição de Objetos ........ 71
2.8.3 - Persistência de Objetos ........ 71

2.8.4 - Linguagem Java e Persistência ........ 72
2.8.5 - Armazenamento de Objetos ........ 73
2.8.6 - Coleta de Lixo ........ 75
2.8.7 - Evolução de Esquema ........ 76
3 - JOODBMS – Java Object Oriented Database ........ 78
3.1 - Arquitetura do JOODBMS ........ 83
3.1.1 - O gerenciador de configurações ........ 85
3.1.2 - O gerenciador de armazenamento ........ 86
3.1.3 - O Gerenciador de *Cache* ........ 87
3.1.4 - O Gerenciador de Páginas ........ 89
3.1.5 - O Serializador de Páginas ........ 91
3.1.6 - O Gerenciador de Índices em Árvore ........ 93
3.1.7 - O Serializador de objetos ........ 97
3.1.8 - O Gerenciador de Metadados ........ 99
3.1.9 - A Fábrica de Serializadores ........ 101
3.2 - Testes ........ 102
3.3 - Utilização do JOODBMS ........ 103
3.4 - Um Exemplo de Aplicação utilizando o JOODBMS ........ 105
4 - Conclusões e Trabalhos futuros ........ 111
5 - Referências Bibliográficas ........ 113

# LISTA DE TABELAS

Tabela 3-1. Descrição sumária dos módulos implementados no JOODBMS .................... 79
Tabela 3-2. Quadro comparativo com outros sistemas gerenciadores de bancos de dados 81
Tabela 3-3. Descrição dos métodos de *joodbms.core.DatabaseModule* ............................ 84
Tabela 3-4. Propriedades utilizadas na configuração do JOODBMS ................................ 86
Tabela 3-5. Descrição dos métodos existentes no Gerenciador de Armazenamento .......... 87
Tabela 3-6. Métodos da interface *PageBuffer* ................................................................... 88
Tabela 3-7. Métodos da interface *PageManager* ............................................................... 90
Tabela 3-8. Métodos de *joodbms.core.DataPage* ............................................................... 92
Tabela 3-9 . Documentação dos métods da classe TuplePage ........................................... 92
Tabela 3-10. Documentação dos métodos utilizados pelo módulo de gerenciamento de índices ........................................................................................................... 94
Tabela 3-11. Métodos das classes e interfaces utilizadas no serializador de objetos .......... 98
Tabela 3-12. Documentação dos atributos da clas classes que armazenam os metadados 100

## LISTA DE FIGURAS

Figura 2.1. Modelo completo de um SGBD e as iterações entre os componentes ............... 9
Figura 2.2. Esquema da hierarquia de memória ............................................................ 14
Figura 2.3. Implementação do merge-sort em Java ........................................................ 20
Figura 2.4. Exemplo de mistura de ponteiros por hardware ............................................ 29
Figura 2.5. Exemplo estrutura de uma página. ............................................................... 40
Figura 2.6. Árvore de pesquisa binária ........................................................................... 51
Figura 2.7. Árvore degenerada ........................................................................................ 52
Figura 2.8. Árvore binária paginada ............................................................................... 53
Figura 2.9. Árvore paginada degenerada ......................................................................... 54
Figura 2.10. Exemplo de organização lógica de dados na página .................................... 55
Figura 2.11. Estrutura de um índice em árvore B ........................................................... 56
Figura 2.12. Concatenação de dois blocos ...................................................................... 59
Figura 2.13. Exemplo de modelo hierárquico ................................................................. 64
Figura 3.1. Módulos utilizados no JOODBMS ................................................................ 82
Figura 3.2. Container dos módulos e interface de sessão ................................................ 83
Figura 3.3. Pacotes existentes na implementação do JOODBMS .................................... 84
Figura 3.4. Interface *joodbms.core.ConfigurationManager* ............................................ 85
Figura 3.5. Interface do Gerenciador de Armazenamento ................................................ 86
Figura 3.6. Interface *joodbms.core.CacheManager* e a interface *joodbms.core.PageBuffer* ........................................................................................................................ 88
Figura 3.7. Interface *joodbms.core.PageManager* e a interface *joodbms.core.PageBuffer* 89
Figura 3.8. Interface *joodms.core.PageSerializer* e classse *DataPage*, que contém os dados da página ........................................................................................................... 91
Figura 3.9. Interface *joodbms.core.TreeIndexManager* e classes e interfaces relacionadas93
Figura 3.10. Formato da tupla ......................................................................................... 94
Figura 3.11. Fluxograma da busca da página folha de um elemento ................................ 95
Figura 3.12. Algoritmo recursivo para divisão de páginas ................................................ 96
Figura 3.13. Classes utilizadas no gerenciamento de índices ............................................ 97
Figura 3.14. Interface *joodbms.core.ObjectSerializer* e dependências ............................. 98
Figura 3.15 . Classes responsáveis pelo gerenciador de metadados ................................ 100
Figura 3.16. Classes utilizadas no serializador de tipos simples ..................................... 102
Figura 3.17. Exemplo de utilização do JOODBMS ........................................................ 104

Figura 3.18. Exemplo de classe persistente ..... 105
Figura 3.19. Tela inicial do sistema de controle de notas fiscais ..... 106
Figura 3.20. Código Java para abertura de conexão e configuração do banco de dados .. 106
Figura 3.21. Classes persistentes utilizadas no Controle de Notas Fiscais ..... 107
Figura 3.22. Recuperação de uma nota fiscal pelo seu número ..... 108
Figura 3.23. Criação e modificação de notas ficais ..... 109
Figura 3.24. Tela de escolha de produto ..... 110
Figura 3.25. Obtenção de todos os produtos cadastrados ..... 110

# LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIAÇÕES

SGBD: Sistema de gerenciamento de dados

PARC: *Palo Alto Research Center*

DDL: *Data definition language* , Comandos de linguagem de definição de dados

BLOB: *Binary Large Object*, Objeto Binátrio Extenso

OID: *Object Identifier*, Identificador de objetos

LRU: *Least Recently Used*, Utilizada menos recentemente

OODBMS: *Object-oriented database management system*, Sistema Gerenciador de Banco de dados orientados a objetos

RDBMS: *Relational Database Management System*, Sistema de gerenciamento de banco de dados relacional

ODMG: *Object Database Management Group*

ODL: *Object Definition Language*, Linguagem de Definição de Objetos

OQL: *Object Query Language*, Linguagem de Consulta de Objetos

IDE: *Integrated Development Environment*, Ambiente de desenvolvimento integrado

ORDBMS: *Object-Relational Database Management System*, Sistema Gerenciador de Banco de Dados Objeto-Relacional

JDBC: *Java Database Connectivity*

CAD/CAM: *Computer Aided Design*, Projeto Auxiliado por Computador

UUID: *Universal Unique Identifier*, identificador único Universal

ODBTWG: *Object Database Technology Working Group*, Grupo de Trabalho em Tecnologias de Bancos de Objetos

# 1 - INTRODUÇÃO

Sistemas gerenciadores de banco de dados devem estar cada vez mais adequados ao tipo de informação que precisam fornecer, e com semânticas de acesso e armazenamento mais próximas das linguagens de programação utilizadas.

Estão em amplo uso as linguagens de programação orientadas a objeto, como *Java [20]*, *C++* [15], *Delphi* [16] e *C#* [27]. Tais linguagens fornecem ao desenvolvedor características não encontradas nas linguagens estruturadas, como encapsulamento, herança e polimorfismo. Estas características permitem ao programador maior facilidade de reuso de partes do *software*, levando a um menor tempo de desenvolvimento e maior facilidade na construção de testes automatizados unitários, como o *JUnit* [31].

Como forma de reutilizar os dados existentes em bancos de dados de tecnologias já estabelecidas, como os bancos relacionais, em sistemas desenvolvidos com tecnologias orientadas a objeto, existem ferramentas de mapeamento objeto-relacional [29] [23], as quais utilizam um mapeamento – fornecido pelo programador – entre as classes e seus atributos, e as tabelas e colunas de um banco de dados relacional. Algumas outras plataformas, como os *Entity Beans* [22] são utilizadas para mapeamento não só com bases de dados relacionais, mas também com outras fontes de dados externas, como sistemas de informações gerenciais ou sistemas legados.

Fornecendo uma interface de acesso aos dados que permita ao usuário uma utilização transparente (independente de os objetos serem ou não persistentes) dos dados, aumenta-se a possibilidade de reuso e encapsulamento, uma vez que se torna desnecessário qualquer camada de mapeamento entre o *software* cliente do banco de dados e o próprio banco de dados. Também o “descasamento de impedância” existente entre a linguagem de programação orientada a objetos causa uma perda de desempenho devido à camada intermediária.

Outro ponto que merece destaque é a perda de produtividade do desenvolvedor, já que ele precisará também se preocupar, além de como os dados serão armazenados no banco de dados, também como os mesmos poderão ser representadas nos objetos materializados, e como as consultas e atualizações serão geradas pelo *framework* de mapeamento.

Além da redução dos problemas de desenvolvimento, os bancos de objetos também são bastante utilizados em aplicações científicas, tendo uma boa importância, por exemplo, nas áreas de sistemas aplicados à biotecnologia, medicina, geoprocessamento e telecomunicações. Conforme pesquisa feita pela Objectivity [33], o acesso e a representação transparentes dos objetos favorecem os usos em aplicações científicas, as quais possuem estruturas de dados com relacionamentos complexos.

Os bancos de objetos disponíveis para uso apresentam-se em duas vertentes:

1. Armazenamento apenas dos dados, com os métodos sendo executados no contexto dos clientes;  e

2. Armazenamento de dados e métodos, mas utilizando linguagens proprietárias, sendo necessário que o desenvolvedor utilize ainda uma linguagem para o modelo de dados e outra para a aplicação propriamente dita.

Além destes, alguns outros bancos de dados são desenvolvidos para utilização de forma embarcada, como o DB4O [26], em que tanto a aplicação como o banco de dados são executados no mesmo processo. Na abordagem do DB4O, não há um conceito rígido de esquema, sendo a persistência baseada em um mecanismo proprietário de serialização, e ainda acoplada a um processo de evolução de esquema na qual é responsabilidade do desenvolvedor codificar as regras para a migração dos dados. Uma das características principais do DB4O é sua forma de armazenamento, baseada em árvore de ordem fixa, codificada de forma a utilizar a menor quantidade possível de espaço em armazenamento secundário.

Outras abordagens, como o Ozone [34] são focadas para o desenvolvimento de gerenciadores de armazenamento que possuam embutida uma lógica para manter os objetos que normalmente são acessados juntos de forma agrupada, reduzindo a quantidade de operações de E/S. Entretanto, ele não permite, de forma fácil, a existência de um mecanismo de consulta baseado nos atributos, sendo o acesso puramente através da navegação no grafo de objetos.

O banco de objetos proposto aqui apresenta uma arquitetura modular, capaz de ser estendido, otimizado e aperfeiçoado, fornecendo novas funcionalidades e permitindo novos desenvolvimentos na área.  A estrutura já desenvolvida fornece o armazenamento baseado

em páginas – favorecendo o uso em sistemas cujo acesso ao armazenamento secundário é baseado em blocos de tamanho fixo, como discos rígidos, e a criação de índices para acesso direto aos registros, através da montagem programática de planos de execução.

O trabalho está dividido em tópicos concernentes à revisão bibliográfica sobre os temas abordados, a apresentação do trabalho desenvolvido e resultados, e conclusões e trabalhos futuros.

No capítulo 2 é feita uma revisão bibliográfica sobre programação orientada a objetos, estrutura física de um sistema gerenciador de banco de dados e as estruturas que os compõem, um breve histórico das tecnologias de bancos de dados e as características de um banco de objetos.

No capítulo 3 é delineada a implementação do JOODBMS, o banco de dados implementado no decorrer desta dissertação, onde são discutidas as implementações básicas para os diversos módulos.

No capítulo 4 são apresentadas as conclusões e propostas para o acréscimo de novas funcionalidades.

# 2 - CONCEITOS TEÓRICOS

Neste capítulo serão tratados os conceitos básicos utilizados em sistemas de gerenciamento de bancos de dados orientados a objetos, métodos de acesso e organização de arquivos e as características das linguagens de programação a objetos.

## 2.1 - PRINCÍPIOS DE ORIENTAÇÃO A OBJETOS

O conceito de programação orientada por objetos não é novo. No final da década de 60, a linguagem Sumula67, desenvolvida na Noruega, introduzia conceitos hoje encontrados nas linguagens orientadas a objetos. Em meados de 1970, o Centro de Pesquisa da Xerox (PARC) desenvolveu a linguagem *Smalltalk*, a primeira totalmente orientada a objetos.

Segundo Luiz Maia [37], a programação orientada a objetos tem como principais objetivos reduzir a complexidade no desenvolvimento de *software* e aumentar sua produtividade. A análise, projeto e programação orientados a objetos são as ferramentas utilizadas para resolver para o aumento da complexidade dos ambientes computacionais atuais, que se caracterizam por sistemas heterogêneos, distribuídos em redes, em camadas e baseados em interfaces gráficas. Pode-se considerar que a programação orientada a objetos é uma evolução de práticas que são recomendadas na programação estruturada, mas não formalizadas, como o uso de variáveis locais, visibilidade e escopo. O modelo de objetos permite a criação de bibliotecas que tornam efetivos o compartilhamento e a reutilização de código, reduzindo o tempo de desenvolvimento e, principalmente, simplificando o processo de manutenção das aplicações.

O paradigma da orientação a objetos está fundamentado no encapsulamento dos dados e, associados a eles, também o código necessário para operar com estes dados. Desta forma, pelo menos em nível conceitual, toda a interação de um objeto com o mundo exterior (ou seja, outros objetos) se dá por meio da troca de mensagens.

Um objeto tem associado:

- Um conjunto de variáveis ou propriedades. Este conjunto de variáveis define o estado do objeto;

- Um conjunto de mensagens ao qual o objeto responde. Estas mensagens definem a interface do objeto com o mundo exterior, e cada mensagem pode ter um ou mais parâmetros; e

- Um conjunto de métodos. Estes métodos consistem no código que implementa as mensagens, e os mesmos podem retornar um valor, sendo este valor conhecido como a resposta à mensagem.

Os objetos se comunicam apenas através de mensagens que, no contexto de orientação a objetos, não se refere a uma mensagem física, como ocorre em redes de computadores, e sim à passagem de pedidos entre os objetos. Pode ser utilizada a expressão "chamar um método" para representar o ato de enviar uma mensagem e a execução do método correspondente no objeto destino.

Os métodos podem ser classificados como acessores, ou somente de leitura, e modificadores, ou de execução. Os métodos somente de leitura não afetam o estado do objeto (estes são especificados, na linguagem C++, pelo modificador **const**); os métodos de execução podem ou não afetar o estado do objeto.

O conceito de encapsulamento consiste em manter variáveis e métodos visíveis apenas através de mensagens. A única maneira de um objeto alterar as variáveis de um outro objeto é através da ativação de um de seus métodos por uma mensagem. O encapsulamento funciona como uma proteção para as variáveis e métodos, além de tornar explícita qualquer tipo de comunicação com o objeto.

Uma classe consiste de variáveis e métodos que representam características de um conjunto de objetos semelhantes. O conceito de classe é um dos pilares da programação orientada a objetos, por permitir a reutilização efetiva de código. Uma classe é o gabarito para a definição de um objeto, descreve que propriedades – ou atributos – o objeto terá. A definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, quais os métodos existem nos objetos da classe.

Herança é um mecanismo que permite que características comuns a diversas classes sejam fatoradas em uma classe base, ou superclasse. Cada classe derivada ou subclasse apresenta

as características (estruturas e métodos) da classe base e acrescenta a elas o que for definido de particularidade para ela.

Conforme Ivan Ricarte [40], há várias formas de relacionamentos por herança:

- **Extensão:** A subclasse estende a superclasse, acrescentando novos membros (atributos e/ou métodos): A superclasse permanece inalterada, motivo pelo qual este tipo de relacionamento é normalmente referenciado como **herança estrita.**

- **Especificação:** A superclasse especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade. Apenas a interface (conjunto de especificação dos métodos públicos) da superclasse é herdada pela subclasse.

- **Combinação de extensão e especificação:** A subclasse herda a interface e uma implementação padrão de (pelo menos alguns de) métodos da superclasse. A subclasse pode então redefinir métodos para especializar o comportamento em relação ao que é oferecido pela superclasse.

O mecanismo de herança é recursivo, permitindo criar-se uma hierarquia de classes. Nos níveis mais altos da hierarquia estão características comuns a todos os objetos desta classe, enquanto nos níveis mais inferiores estão especializações das classes superiores. As subclasses herdam as características comuns, além de definirem suas propriedades específicas.

Ainda segundo Luiz Maia [37], outra classificação possível referente ao mecanismo de herança é o fato de existirem dois tipos de mecanismos para sua implementação: simples e múltipla. Na herança simples, a subclasse pode herdar variáveis e métodos apenas de uma classe, enquanto na herança múltipla, a subclasse pode herdar variáveis e métodos de mais de uma classe.

Uma das grandes vantagens da programação orientada a objetos é a utilização de bibliotecas de classes. As bibliotecas de classes permitem uma capacidade muito maior de compartilhamento e reutilização de código, pois é possível criar subclasses para atender novas necessidades, em função das classes já existentes.

O mecanismo de polimorfismo permite tratar objetos semelhantes de uma maneira uniforme, sendo que cada objeto responda de uma forma diferente para uma mesma mensagem. Neste caso, cada classe que implementa um método específico possui sua própria implementação.

O polimorfismo, para ser implementado, exige a utilização do conceito de herança e aplica-se apenas aos métodos da classe. O protocolo de comunicação é estabelecido na classe mais alta da hierarquia, que será herdada por todas as subclasses definidas posteriormente. Este mecanismo cria um protocolo padrão de comunicação com um conjunto de objetos, permitindo uma grande flexibilidade na agregação de objetos semelhantes, mas não idênticos.

Com o conceito de polimorfismo, é possível acrescentar novos métodos a classes já existentes sem a necessidade de recompilar toda a aplicação, apenas a classe modificada. Isto é possível através da técnica de *late binding* ou *dynamic binding*, que permite que novos métodos sejam carregados e ligados (*binding*) à aplicação em tempo de execução.

### 2.1.1 - A linguagem Java

Com a introdução da linguagem Java [20], teve início uma grande revolução no mundo da orientação a objetos. Esta linguagem, que foi construída a partir de experiências obtidas de outras linguagens orientadas a objeto, como C++ e *Smalltalk*, tem grande aceitação no mundo da programação *web*, seja em códigos executados no cliente (*applets*) ou no servidor (*servlets*). Ainda hoje é uma das linguagens orientadas a objeto mais comuns, sendo que outras tecnologias similares, como a DotNet [25] também estão tendo grande aceitação.

A parte fundamental dos programas Java são as classes. A linguagem possui uma grande quantidade de classes já prontas, disponíveis para os programadores reutilizarem em seus próprios programas. Estas classes reutilizáveis fazem parte da biblioteca de classes Java, conhecidas como Java APIs [7].

## 2.2 - BANCO DE DADOS

Segundo Garcia-Molina [12], o poder dos bancos de dados vem de um corpo de conhecimento e tecnologia chamado *sistema de gerenciamento de banco de dados*

(SGBD). Um SGBD é uma ferramenta para criar e gerenciar grandes quantidades de dados de forma eficiente, e permitir que esses dados persistam durante longos espaços de tempo. Os recursos que um SGBD oferece são:

1. **Armazenamento persistente**. Um SGBD permite o armazenamento de grandes quantidades de dados, independente dos processos que estejam armazenando estes dados, e ainda oferece estruturas que permitem o acesso eficiente a estes dados.

2. **Interface de programação**. Um SGBD permite acessar e modificar dados através de uma linguagem de consulta poderosa. A vantagem de um SGBD é flexibilidade para manipular os dados armazenados de formas muito mais complexas que a simples leitura e gravação em arquivos.

3. **Gerenciamento de transações**. Um SGBD admite o acesso concorrente a dados. Para evitar algumas das conseqüências indesejáveis do acesso simultâneo, o SGBD admite o *isolamento*, ou seja, a aparência de que as transações são executadas uma de cada vez, e a *atomicidade*, o requisito de que as transações sejam executadas completamente ou não sejam executadas de forma alguma. Admite também a *resiliência*, ou seja, a capacidade de se recuperar de muitos tipos de falhas ou erros.

### 2.2.1 - Visão geral de um SGBD

Na Figura 2.1 temos o esboço de um SGBD completo. As partes serão estudadas independentemente, devido à complexidade do sistema.

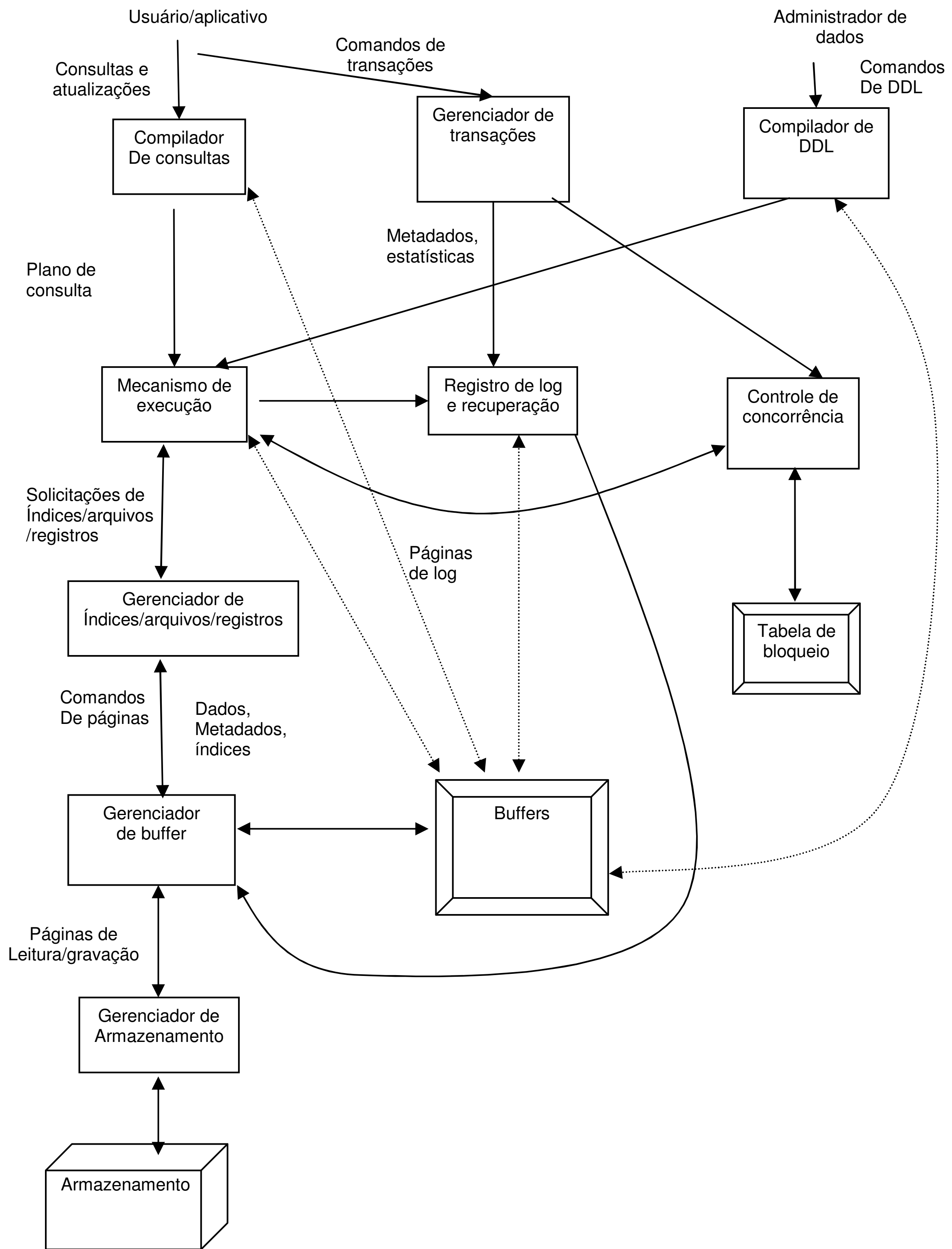



Figura 2.1. Modelo completo de um SGBD e as iterações entre os componentes. Adaptado de Garcia-Molina [12].

### 2.2.2 - Comandos da linguagem de definição de dados (DDL)

Estes tipos de comandos são os mais fáceis de serem processados, pois se aplicam apenas às estruturas e restrições que descrevem o banco de dados como um todo – os *metadados*. Estes comandos são processados por um processador de DDL e repassados ao mecanismo de execução, que percorre o gerenciador de índices/arquivos/registros para alterar os metadados.

### 2.2.3 - Processamento de consultas

As consultas são ações geradas ou por usuários ou por programas aplicativos que não afetam o esquema do banco de dados, mas apenas o conteúdo do banco de dados (para os comandos de modificação), ou apenas extraem os dados do banco de dados (para as operações de consulta). Assim, há dois caminhos que devem ser seguidos para as operações serem processadas:

1. *Responder à consulta*. A consulta é analisada gramaticalmente e aperfeiçoada por um compilador de consultas. O plano de consulta resultante – a seqüência de ações a serem executadas – é repassado ao mecanismo de execução. O mecanismo de execução emite uma seqüência de solicitações de pequenos itens de dados para um gerenciador de recursos que é capaz de acessar os arquivos de dados e os arquivos de índices. As solicitações de dados são convertidas em páginas e essas solicitações são repassadas ao gerenciador de *buffers*. O gerenciador de *buffers* se comunica com o gerenciador de armazenamento para obter dados do disco.

2. *Processamento de transações*. As consultas e outras ações estão agrupadas em transações, que são as unidades que têm que ser executadas atomicamente e em isolamento. A execução de transações também deve ser *durável*, ou seja, o efeito de qualquer transação completada com sucesso deve ser preservado, ainda que haja alguma falha no sistema logo após a transação ser concluída. Há duas partes no gerenciador de transações:

    1. Um gerenciador de controle da concorrência (*scheduler*), responsável por assegurar a atomicidade e o isolamento das transações; e

    2. Um gerenciador de registro de log e recuperação, responsável pela durabilidade das transações.

### 2.2.4 - O gerenciador de buffers

Os dados de um banco de dados em geral residem no armazenamento secundário. Porém, para se executar qualquer operação útil sobre os dados, estes devem ser carregados na memória principal. O gerenciador de *buffers* é responsável por particionar a memória principal em *buffers*, regiões com tamanhos de página normalmente múltiplas do tamanho de uma unidade de armazenamento em disco, para os quais os dados são transferidos de e para o disco. Assim, todos os componentes do SGBD que precisam de informações do disco irão interagir com os *buffers* e com o gerenciador de *buffers*. As seguintes informações podem ser necessárias para os vários componentes:

1. *Dados*: o conteúdo do próprio banco de dados;

2. *Metadados*: o esquema do banco de dados, que descreve sua estrutura e restrições

3. *Estatísticas*: as informações obtidas e armazenadas pelo SGBD sobre propriedades de dados, como os tamanhos de diversas relações ou extensões de classe, ou outros componentes do banco de dados e os valores que eles contêm; e

4. *Índices*: estruturas de dados que facilitam o acesso eficiente aos dados.

### 2.2.5 - Processamento de transações

O gerenciador de transações é o responsável por receber os comandos de transações de um aplicativo, utilizados para demarcar o início e fim das transações, e também as expectativas do aplicativo em relação às mesmas. O gerenciador de transações tem as seguintes responsabilidades:

1. *Registro de log*. Para assegurar durabilidade, toda mudança no banco de dados é registrada pelo menos duas vezes no banco de dados: uma no arquivo de dados, e outra no registro de *log*. O registro de *log* segue uma dentre várias normas projetadas para garantir que, independente de quando ocorre uma falha no sistema, o gerenciador de recuperação possa ler os registros do *log* e retornar o banco de dados a um estado consistente, cancelando as transações que estiverem ainda em aberto.

2. *Controle de concorrência.* As transações devem aparentar ser executadas em isolamento, mesmo ocorrendo muitas transações em execução simultaneamente. O gerenciador de concorrência (*scheduler*) deve assegurar que as ações individuais de várias transações serão executadas em tal ordem que o efeito líquido será igual ao que haveria se todas as transações fossem de fato executadas serialmente. Normalmente este trabalho de escalonamento de operações é feito através de bloqueios sobre os diversos tipos de dados, os quais são mantidos em uma tabela chamada tabela de bloqueios.

3. *Resolução de impasses.* À medida que as transações competem por recursos (bloqueios), pode ocorrer de duas ou mais transações não poderem prosseguir, devido ao fato de uma necessitar de um recurso que outra transação possui. O gerenciador de transações tem, nesse caso, a responsabilidade de intervir em uma das transações e cancelá-la, garantindo o progresso de outras transações.

### 2.2.6 - O processador de consultas

O processador de consultas é, certamente, a parte que mais afeta o desempenho visto pelo usuário. Na Figura 2.1, ele é representado por:

1. O compilador de consultas, responsável por converter a consulta enviada pelo programa aplicativo em uma estrutura de dados chamada plano de consulta, a qual define uma seqüência de operações que serão executadas sobre os dados. As operações deste plano normalmente são implementações da álgebra relacional. Ele é composto de três unidades principais:

    1. O analisador de consultas, que constrói uma estrutura em árvore (*parse tree*) a partir do texto da consulta;

    2. O pré-processador de consultas, responsável pela verificação semântica das consultas (por exemplo, se as relações e colunas existem realmente), e efetua transformações na árvore, transformando-a numa árvore de operações algébricas; e

3. O otimizador de consultas, que, utilizando-se dos metadados e estatísticas, transforma o plano inicial no melhor plano disponível para a execução das operações.

2. O mecanismo de execução, responsável por executar cada um dos passos descritos no plano de consulta. Este mecanismo interage com quase todos os outros módulos do SGBD, seja diretamente, seja através de *buffers*.

### 2.2.7 - Dados Relacionais e Orientados a Objeto

Em um banco de dados relacional, os dados são modelados por tabelas, os itens de dados são tuplas ou linhas da tabela; e as tuplas possuem um número fixo de componentes, cada um dos quais de um tipo fixo determinado pelo esquema da relação.

Existe outro modelo de dados que é usado em sistemas gerenciadores de banco de objetos: dados como objetos. Nesse modelo, o item de dados elementar é um objeto. Os objetos são agrupados em classes e cada classe tem um esquema, formado por propriedades que podem :

1. Algumas dessas propriedades são atributos que podem ser representados como atributos de uma tupla relacional;

2. Outras propriedades são relacionamentos, que conectam um objeto a um ou mais objetos diferentes; e

3. Ainda outras propriedades que são métodos, ou seja, funções que podem ser aplicadas a objetos da classe.

Assim, há também alguns conceitos semelhantes entre os bancos relacionais e os bancos de objetos:

1. Arquivo, relação e extensão, que possuem elementos com um esquema comum (registros, tuplas ou objetos)

2. O esquema de um arquivo ou relação e a definição de uma classe, que descrevem os elementos de um arquivo, relação ou extensão (de uma classe)

3. Registros, tuplas e objetos.

### 2.2.8 - Armazenamento de dados

Um dos aspectos importantes que distinguem os sistemas gerenciadores de bancos de dados de outros sistemas é a habilidade de um SGBD para lidar de forma eficiente com quantidades muito grandes de dados.

Conforme mostrado a seguir, a eficiência de algoritmos envolvendo quantidades muito grandes de dados depende do padrão de movimentação de dados entre a memória principal e o espaço de armazenamento secundário. Devido aos diferentes tipos de memórias, pode-se considerar uma hierarquia entre os diferentes tipos, classificadas por velocidade de acesso e capacidade de armazenamento.

### 2.2.9 - A hierarquia da memória

Um sistema de computador típico tem vários componentes diferentes nos quais os dados podem ser armazenados. Um diagrama esquemático da hierarquia da memória é mostrado na Figura 2.2.

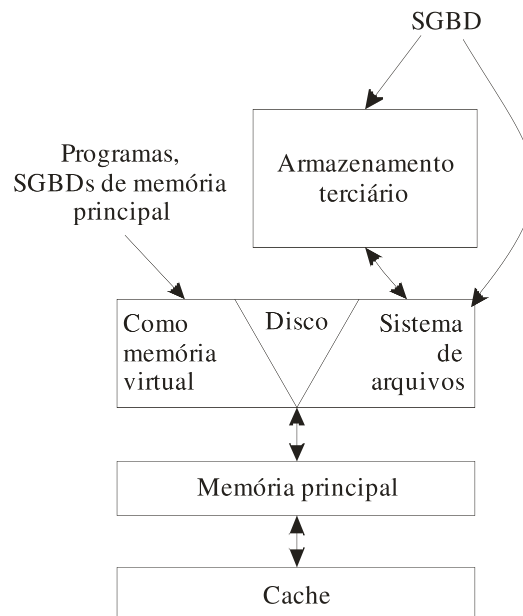



Figura 2.2. Esquema da hierarquia de memória. Adaptado de Garcia-Molina [12]

#### 2.2.9.1 - Cache

No nível mais baixo da hierarquia está um *cache*. Os dados (incluindo as instruções) contidos no cache são uma cópia de certas posições da memória principal. Às vezes, os valores no cache são alterados, mas a mudança correspondente na memória principal é adiada.

Em um sistema multiprocessado, que permite que vários processadores tenham acesso à memória principal e mantenham seus próprios caches privados, muitas vezes é necessário executar atualizações do cache para regravar, isto é, mudar de imediato a posição correspondente na memória principal.

Quando executa instruções, a máquina procura no cache as instruções e também os dados usados por essas instruções. Se não encontrar tais informações no cache, a máquina irá até a memória principal e copiará as instruções ou os dados no cache.

#### 2.2.9.2 - Memória principal

No centro da ação encontra-se a memória principal do computador. Todas as operações que acontecem no computador (execução de instruções e manipulação de dados) atuam sobre as informações que se encontram na memória principal.

As memórias principais são de acesso aleatório, significando que é possível obter qualquer byte aproximadamente no mesmo período de tempo.

#### 2.2.9.3 - Memória virtual

Quando os softwares são executados, os dados utilizados ocupam um espaço de endereços da memória virtual. As instruções de programa também ocupam espaços de endereços próprios para elas.

Normalmente, um espaço de memória virtual é maior que a memória principal. Assim, uma parte do conteúdo da memória virtual ocupada fica, na realidade, armazenada em disco, no chamado arquivo de troca. A memória virtual é deslocada entre o disco e a memória principal em blocos inteiros, os quais costumam ser chamados páginas na memória principal. O hardware da máquina e o sistema operacional permitem que as páginas da memória virtual sejam trazidas para qualquer parte da memória principal, e que

cada byte desse bloco seja identificado de forma apropriada por seu endereço na memória virtual.

Há um interesse crescente em sistemas de bancos de dados de memória principal, que fazem o gerenciamento de seus dados por meio da memória virtual, baseando-se no sistema operacional para trazer os dados necessários até a memória principal, por intermédio de um mecanismo de paginação. Os sistemas de bancos de dados de memória principal são mais úteis quando os dados são pequenos o bastante para permanecerem na memória principal sem serem trocados pelo sistema operacional. Assim, os sistemas de bancos de dados em ampla escala gerenciam geralmente gerenciam os seus dados diretamente no disco.

### 2.2.10 - Armazenamento secundário

Quase todo computador tem algum tipo de armazenamento secundário, uma forma de espaço de armazenamento que é ao mesmo tempo significativamente mais lenta e de maior capacidade que a memória principal, ainda que seja em essência de acesso aleatório.

O disco é considerado como o suporte para a memória virtual e para um sistema de arquivos. Os arquivos são movidos entre o disco e a memória principal em blocos, sob o controle do sistema operacional ou do sistema de banco de dados. Certas partes da memória principal são utilizadas como arquivos de *buffer*, ou seja, para conter fragmentos desses arquivos com dimensões de blocos. É vital que, sempre que possível, um bloco de disco contendo dados que precisamos acessar já esteja em algum buffer na memória principal.

### 2.2.11 - Armazenamento volátil e não volátil

Uma distinção entre dispositivos de armazenamento os define como **voláteis** ou **não voláteis**. Um dispositivo volátil perde o que está armazenado nele quando a energia é desligada. Por outro lado, um dispositivo não volátil deve manter seu conteúdo intacto mesmo por longos períodos quando o dispositivo é desligado ou ocorre uma falha de energia.

Essencialmente todos os dispositivos de armazenamento secundário e terciário são não voláteis. Por outro lado, a memória principal em geral é volátil. Normalmente o armazenamento secundário é quase exclusivamente baseado em discos magnéticos.

Um sistema de banco de dados que funciona em uma máquina com memória principal volátil deve fazer a cópia de toda mudança em disco, antes da mudança poder ser considerada parte do banco de dados. Como conseqüência, as modificações de dados em bancos de dados devem envolver um número considerável de gravações em disco.

#### 2.2.11.1 - O uso eficiente do espaço de armazenamento secundário

Na maioria dos estudos de algoritmos, supõe-se que os dados estão na memória principal e que o acesso a um item de dados demora tanto tempo quanto o acesso a qualquer outro. Esse modelo de computação é chamado “modelo de RAM” ou modelo de acesso aleatório de computação. Porém, ao se implementar um SGBD, deve-se supor que os dados não cabem na memória principal. Assim, deve-se levar em consideração o uso do espaço de armazenamento secundário no projeto eficiente de algoritmos. Os melhores algoritmos para processar quantidades de dados muito grandes difere freqüentemente dos melhores algoritmos de memória principal para o mesmo problema.

Há uma grande vantagem em projetar algoritmos que limitam o número de acessos a disco, ainda que as ações executadas pelo algoritmo sobre os dados na memória principal não sejam aquilo que pode ser considerado ótimo para este tipo de memória.

### 2.2.12 - O modelo de computação de E/S

No modelo de computação de E/S, se um bloco precisar ser movido entre o disco e a memória principal, o tempo necessário para executar a leitura ou gravação será muito maior que o tempo provável para se manipular esses dados na memória principal. Assim, o fator preponderante para os algoritmos que levam em consideração o modelo de computação de E/S é o tempo necessário para as operações de disco, mesmo que o uso de memória principal não seja considerado ótimo.

### 2.2.13 - Classificação de dados em armazenamento secundário

Como um exemplo de como os algoritmos precisam mudar sob o modelo de custo de computação de E/S, será considerado como exemplo o processo de classificação, quando os dados são muito maiores que a memória principal.

Se os dados couberem na memória principal, haverão uma série de algoritmos conhecidos que funcionarão bem. Além disso, usa-se uma estratégia na qual só seriam classificados os campos de chaves com ponteiros associados aos registros completos.

Entretanto, essas técnicas não funcionam muito bem quando a memória secundária é necessária para guardar os dados. As abordagens utilizadas para classificação, quando os dados estão principalmente na memória secundária, envolvem efetuar a movimentação dos blocos entre o armazenamento secundário e a memória principal apenas um pequeno número de vezes, em um padrão regular. Com freqüência, esses algoritmos operam em um pequeno número de passagens; em uma passagem, cada bloco é lido na memória principal uma vez e gravado uma vez para o disco.

Para ordenação em memória, há diversos algoritmos possíveis [19], como o *bubble sort* e o *quicksort*. Entretanto, nestes algoritmos ocorrem várias passagens pelo mesmo registro, tornando o desempenho inaceitável, se for necessário efetuar uma operação de entrada e saída para cada registro lido.

Uma alternativa para o caso da ordenação de grandes arquivos de dados é a ordenação das chaves (*keysorting*) [10], em que apenas as chaves dos registros e os ponteiros para a localização dos registros no arquivo são carregados em memória. Neste algoritmo parte-se do pressuposto que as chaves são pequenas o suficiente para serem totalmente mantidas em memória RAM, e algoritmos tradicionais de ordenação em memória podem ser utilizados. Em seguida, o arquivo de índice completo é reescrito em disco. Após a ordenação, utiliza-se este índice em RAM para reescrever o arquivo de dados, movendo os registros para as posições corretas.

Mesmo uma estrutura como esta ainda apresenta desvantagens, pois ainda pode ocorrer, para arquivos de dados bastante grandes, que as chaves correspondentes aos registros não caibam em memória principal.

Durante a operação de reescrita, os registros originais são lidos em ordem aleatória, o que representa uma grande perda de desempenho, devido às possíveis movimentações da cabeça de leitura/gravação do disco. Como alternativa, pode-se simplesmente ignorar a etapa de reorganização do arquivo, armazenando, ao invés disso, a própria estrutura de dados que contém as chaves e os ponteiros para os registros – ou seja, um arquivo de índice.

Tendo em mente estas limitações para a ordenação de arquivos grandes, vê-se que é necessário outro algoritmo que leve em consideração as características do armazenamento secundário – ou seja, o acesso aleatório é consideravelmente mais lento que o acesso seqüencial. O principal foco é em como utilizar estas características em favor do algoritmo, e técnicas como o processamento **co-sequencial** [10] podem ser utilizadas. Uma abordagem comumente utilizada é a ordenação baseada no algoritmo *merge sort* [10] [12].

O processamento co-sequencial consiste no processamento coordenado de duas ou mais listas seqüenciais, produzindo uma única lista de saída. Dentre as formas de processamento existem a união ou a intersecção entre duas ou mais listas.

O algoritmo *merge sort* consiste na aplicação do princípio de dividir para conquistar. O arquivo não classificado é dividido em vários outros arquivos menores, chamados *corridas*. Cada corrida é obtida lendo-se uma parte do arquivo, de forma a preencher todo um *buffer* em memória, classificando os registros ali carregados e gravando o arquivo de saída. Podem ser utilizados diferentes algoritmos para ordenação, sendo de grande utilidade o *heapsort*, em que os registros são ordenados à medida que os blocos vão sendo lidos a partir do disco. Desta forma é possível executar de forma paralela as operações de E/S e ordenação.

O passo seguinte consiste no processamento co-sequencial dos arquivos resultantes das corridas, através da sincronização e junção dos mesmos. Este processo é dividido nas seguintes fases:

- Inicialização, em que as variáveis e os arquivos são abertos, preparando o processo para ocorrer corretamente;

- Sincronização, garantindo que o registro em uma lista não esteja muito adiante do registro em outras listas;

- Gerenciando condições de fim de arquivo, para controlar o fim do processo. O processo de junção (*merge*) ocorre quando todos os arquivos terminarem de ser lidos.

Na Figura 2.3 está exemplificado a codificação de um algoritmo de *merge sort* utilizando a linguagem Java.

```
public void merge(IndexTreeIterator[] iterators, int keySize,
SerializerFactory factory, HeapWriter heap)  throws JoodbmsException,
JoodbmsFatalException {

  TupleData[] allTuples = new TupleData[iterators.length];
  int i = 0;
  for (IndexTreeIterator iter : iterators) {
   if (iter.hasNext()) {
    allTuples[i] = iter.next();
   }
   i++;
  }
  while (true) {
   // Encontra o mínimo entre todas as tuplas
   ByteBuffer[] minKeys = null;
   AttributeDescriptor[] minKeyDescriptors = null;
   int minTuple = -1;
   for (i = 0; i < allTuples.length; i++) {
    if (allTuples[i] != null) {
     if (minKeys == null || BTreePage.keyCompare(minKeys, factory,
minKeyDescriptors, allTuples[i]) > 0) {
      minTuple = i;
      minKeys = allTuples[minTuple].getColumns(0, keySize);
      minKeyDescriptors = allTuples[minTuple].getDescriptors();
     }
    }
   }
   if (minTuple == -1) {
    heap.close();
    return;
   }
   heap.write(allTuples[minTuple]);
   if (iterators[minTuple].hasNext()) {
    allTuples[minTuple] = iterators[minTuple].next();
    haveMoreNames = true;
   } else {
    allTuples[minTuple] = null;
   }
  }
 }
}
```

Figura 2.3. Implementação do merge-sort em Java

### 2.2.14 - Representação de elementos de dados

Há uma hierarquia no modo como as relações ou conjuntos de objetos são representados no espaço de armazenamento secundário.

- Os atributos precisam ser representados por seqüências de bytes de comprimento fixo ou variável, chamadas “campos”.
- Os campos, por sua vez, são reunidos em coleções de comprimento fixo ou variável, chamadas “registros”, que correspondem a tuplas ou objetos.
- Os registros precisam ser armazenados em blocos físicos. Várias estruturas de dados são úteis, especialmente se blocos de registro precisarem ser reorganizados quando o banco de dados for modificado.
- Uma coleção de registros que forma uma relação ou a extensão de uma classe é armazenada como uma coleção de blocos, chamada *arquivo*. Para dar suporte eficiente a consultas e a modificação dessas coleções, são inseridos uma das diversas estruturas de índice.

Em última instância, todos os dados são representados como uma seqüência de bytes. Para alguns tipos de dados, como números e booleanos, os mesmos são armazenados com um número constante de bytes. Outros, como as *strings* de caracteres, podem necessitar de representações especiais que definam também a quantidade de bytes utilizados.

#### 2.2.14.1 - *Strings* de caracteres de comprimento fixo

O tipo de strings de caracteres mais simples de representar é aquele descrito pelo tipo SQL *CHAR(n)*. Esses strings são seqüências de caracteres de comprimento fixo *n*. O campo para um atributo com esse tipo é um *array* de *n* bytes.

#### 2.2.14.2 - *Strings* de caracteres de comprimento variável

Às vezes, os valores em uma coluna de uma relação são cadeias de caracteres cujo comprimento pode variar amplamente. O tipo de SQL *VARCHAR(n)* é usado com freqüência para uma coluna como esta. Há duas representações comuns para strings *VARCHAR:*

1. Comprimento mais conteúdo. É alocado um *array* de *n + 1* bytes, e o primeiro byte contém, como um inteiro de 8 bits, o número de bytes da string.

2. String terminada em nulo. É alocado também um array de *n + 1* bytes, e o array é preenchido pelo conteúdo da string, e, logo em seguida, o *caractere nulo*. Este terminador nulo torna a representação de strings *VARCHAR* idêntico à representação de *strings* na linguagem C.

### 2.2.15 - Representação de elementos de bancos de dados relacionais

Como uma relação é um conjunto de tuplas, e tuplas são semelhantes a registros, podemos imaginar que cada tupla será armazenada em disco como um registro. O registro ocupará parte de algum bloco de disco, e, dentro do registro, haverá um único campo para cada atributo da relação.

Apesar de a representação parecer simples, há alguns detalhes que precisam ser observados com mais atenção:

1. Como lidar com registros de tamanhos diferentes, ou que não dividam o bloco de tamanho uniforme?

2. O que acontecerá se o tamanho de um registro se alterar porque o campo foi atualizado?

Além disso, é necessário considerar como representar certos tipos de dados que são encontrados em modernos sistemas objeto-relacionais ou orientados a objetos, como identificadores de objetos (ou ponteiro para registros) e *blobs* (objetos binários extensos).

### 2.2.16 - Representação de objetos

Em uma primeira aproximação, um objeto é uma tupla, e seus campos ou "variáveis de instâncias" são atributos. Porém, há duas diferenças importantes:

1. Os objetos podem ter **métodos** ou outras funções de uso especial associadas a eles.

2. Os objetos têm um **identificador de objeto** (OID, *object identifier*), que é um endereço definido em um esquema global de endereçamento. Os objetos também

podem ter relacionamentos com outros objetos, e esses relacionamentos são representados por ponteiros ou lista de ponteiros. Os dados relacionais não têm endereços como valores. A questão de representar endereços é complexa, tanto para grandes relações quanto para classes com grandes extensões.

### 2.2.17 - Registros

Os campos são agrupados em registros. Em geral, cada tipo de registro usado por um sistema de banco de dados deve ter um esquema, o qual é armazenado pelo próprio banco de dados. O esquema inclui os nomes e os tipos de dados de campos no registro e suas posições dentro do registro, e é consultado quando for necessário acessar componentes do registro.

A situação mais simples são os registros com campos de tamanho de tamanho fixo, em que os registros são simplesmente concatenados.

#### 2.2.17.1 - Cabeçalhos de registros

Com freqüência, há informações que devem ser mantidas no registro, mas que não correspondem ao valor de qualquer campo. Por exemplo, podemos querer manter no registro:

1. O esquema ou um ponteiro para o esquema do registro;
2. O comprimento registro; e
3. Timbres de hora (*timestamp*) indicando a hora em que o registro foi modificado ou lido pela última vez.

O sistema de banco de dados mantém as seguintes **informações de esquema**, as quais são acrescentadas quando a relação ou *extent* são criados:

1. Os atributos da relação;
2. Os respectivos tipos;
3. A ordem em que os atributos aparecem na tupla; e

4. Restrições sobre os atributos e sobre a própria relação (declarações de chaves primárias, estrangeiras e restrições sobre faixas de valores)

Ainda que o comprimento da tupla possa ser deduzido desse esquema, muitas vezes é conveniente que no próprio registro também tenha esta informação. Por exemplo, algumas vezes o necessário é apenas encontrar rapidamente o início do próximo registro. O acréscimo do campo de comprimento de registro no próprio registro permite evitar ter que acessar o esquema, o que poderia envolver mais uma operação de E/S de disco.

2.2.17.2 - Armazenamento de registros de tamanho fixo em blocos

Os registros que representam tuplas de uma relação são armazenados em blocos do disco e movidos para a memória principal (juntamente com o bloco inteiro), quando for necessário acessar ou atualizar esses registros.

Há um cabeçalho de bloco que guarda informações como:

1. Ponteiros para um ou mais blocos diferentes que fazem parte de uma rede de blocos;
2. Informações sobre a função desempenhada por esse bloco em tal rede;
3. Informações sobre a relação a qual pertencem as tuplas desse bloco;
4. Um “diretório” fornecendo o deslocamento de cada registro no bloco;
5. Uma “identificação de bloco”; e
6. *Timestamps* indicando a hora da última modificação e/ou do último acesso ao bloco.

Quando o bloco contém tuplas de uma única relação, e estas possuem um tamanho fixo, é o caso mais simples. Simplesmente são empacotados, após o cabeçalho, o máximo de registros que podem ser inseridos no bloco, e o espaço restante é deixado sem uso.

### 2.2.18 - Representação de endereços de blocos e registros

Também pode ser necessário representar endereços, ponteiros ou referências a registros e blocos, pois, em muitos casos, estas estruturas tomam parte em registros complexos.

Quando em memória virtual, o endereço de seu primeiro byte pode ser considerado como o endereço ou ponteiro para o registro. Entretanto, quando o mesmo está no espaço de memória secundário, normalmente é utilizada uma seqüência de bytes que descreve a localização do bloco dentro do sistema de dados.

#### 2.2.18.1 - Endereços lógicos e estruturados

Muitas organizações de dados exigem a movimentação de registros, seja dentro de um bloco ou de um bloco para outro. Se for utilizada uma tabela de mapas, todos os ponteiros para o registro farão referência a essa tabela de mapas, e será necessário, ao mover ou excluir um registro, alterar a entrada para este registro na tabela.

Além disso, é possível montar endereços estruturados, como, por exemplo, usar o endereço físico do bloco e também a chave do registro dentro do bloco. Assim, o endereço é dividido em uma parte física e outra relativa à chave no interior do bloco. Desta forma, utiliza-se a parte física para encontrar o bloco, e, uma vez com ele carregado, a chave é usada para buscar, no bloco, o registro que tem a chave apropriada.

O caso mais simples ocorre quando os registros são de um tipo de comprimento fixo conhecido, com o campo da chave em um deslocamento conhecido. Nesse caso, só é necessário localizar, no cabeçalho, uma contagem de quantos registros existem no bloco, e a partir daí sabe-se exatamente onde encontrar os campos de chave que podem corresponder à chave que faz parte do endereço. Entretanto, existem muitos outros modos nos quais os blocos poderiam estar organizados para tornar possível uma pesquisa no bloco.

É bastante útil manter em cada bloco uma **tabela de deslocamentos** que contém os deslocamentos dos registros dentro do bloco. Nesta estrutura, a tabela de deslocamentos cresce a partir do início do bloco, e os registros crescem a partir do fim. Isso permite que não seja necessário reservar, *a priori*, um espaço fixo para esta tabela.

Esse nível de indireção dentro do bloco fornece muitas das vantagens de endereços lógicos, sem a necessidade de uma tabela de mapas global:

- Pode-se mover o registro dentro do bloco, e esta movimentação torna necessário apenas alterar a entrada do registro na tabela de deslocamentos; os ponteiros para o registro ainda serão capazes de localizá-lo;

- Pode-se permitir que o registro se desloque para outro bloco, se as entradas da tabela de deslocamentos forem grandes o bastante para conter um **endereço de encaminhamento** para o registro; e

- Caso o registro seja excluído, deixar na entrada de deslocamento um valor que informa que o registro foi excluído, chamado **lápide**.

2.2.18.2 - Mistura de ponteiros

Com freqüência, os ponteiros ou endereços fazem parte de registros. Esta situação é bem característica de tuplas que representam objetos ou, em caso de sistemas objeto-relacionais ou relacionais estendidos, de referências para outras tuplas. Também os índices são geralmente compostos por estruturas de arquivo que possuem ponteiros em seu interior.

Para cada registro, bloco, objeto ou qualquer outro tipo de item armazenado em um banco de dados, é possível referenciá-lo através de seu endereço no banco de dados (no armazenamento secundário), ou por seu endereço em memória virtual, quando o bloco foi carregado para um *buffer* em memória.

Quando o item estiver no armazenamento secundário, deve ser utilizado o endereço de banco de dados do item. Porém, quando o item estiver na memória principal, pode-se fazer referência ao item por seu endereço de banco de dados ou por seu endereço de memória.

É mais eficiente utilizar ponteiros de memória, pois seu acesso é feito através de instruções simples da CPU. Para o uso de endereço de banco de dados, quando o item estiver em memória, é necessário o uso de uma tabela de conversão de endereços de banco de dados para endereços de memória.

Entretanto, o uso de tal tabela gera um custo no uso do processador, e, para evitar tais pesquisas constantes, foram desenvolvidas técnicas conhecidas como ***mistura de ponteiros***, que consistem em uma estrutura de dados que contem:

1. Um *flag* indicando se o ponteiro corresponde a um endereço de banco de dados ou a um endereço de memória; e

2. O ponteiro para o banco de dados ou para memória, de acordo com o especificado no *flag*.

São possíveis quatro formas de mistura de ponteiros: a mistura automática, a mistura por demanda, nenhuma mistura, e o controle feito pelo programador:

1. Na mistura automática, assim que um bloco é trazido para memória, todos os seus ponteiros que já estão carregados em memória são atualizados para o ponteiro misturado correspondente. Assim, todos os ponteiros que *apontam* para o bloco recentemente recuperado do armazenamento secundário são automaticamente traduzidos para seu endereço de memória. Ao gravar os blocos que estão em memória, todos os ponteiros são verificados, e, caso algum aponte para um endereço misturado, o mesmo é trocado por seu endereço de banco de dados correspondente;

2. Na mistura por demanda, quando o bloco for trazido para a memória, os ponteiros para ele não serão alterados, e seu endereço de banco de dados será inserido em uma tabela de conversão, juntamente com seu endereço de memória. A conversão ocorrerá apenas quando for seguido algum ponteiro de banco de dados, em que automaticamente será verificado se o mesmo já se encontra na tabela de conversão, e, caso esteja, ele será trocado pelo ponteiro de memória. Também neste caso é necessária a conversão de volta para os endereços de banco de dados, ao se gravar algum bloco;

3. Quando não for utilizada nenhuma mistura, ainda será necessária a tabela de conversão de endereços de bancos de dados para endereços de memória, e todos os acessos serão feitos utilizando o endereço de banco de dados. Ao seguir um endereço de banco de dados, é feita uma busca na tabela de conversão, e o endereço

resultante é utilizado, mas os blocos em memória não terão seus ponteiros modificados; e

4. No controle feito pelo programador, é feita uma escolha em tempo de desenvolvimento sobre quais blocos têm mais chances de serem acessados, e ele pode ser capaz de especificar explicitamente que um bloco carregado na memória deva ter seus ponteiros misturados automaticamente ou que eles sejam misturados somente se necessário, ou também que eles não sejam misturados de forma alguma.

A mistura pode ser feita tanto por *software* como por *hardware*. A mistura por *software* consiste em utilizar-se de mecanismos construídos no próprio SGBD para localizar as referências para outros registros e alterá-las para as referências em memória correspondentes aos registros já carregados. Cuidados especiais devem ser mantidos para atualizar as tabelas correspondentes, para que, quando os blocos forem novamente gravados em disco, atualizar novamente os dados, de forma a que eles apontem novamente para ponteiros de banco de dados.

A mistura por *hardware* utiliza o mecanismo de paginação oferecido pelo sistema operacional e pelo *hardware* para identificar as páginas ainda não carregadas. Como os ponteiros de banco de dados muitas vezes são bastante maiores que os ponteiros de memória, é preciso de alguma tabela já existente no interior das páginas que torne possível o mapeamento entre um endereço de banco de dados e um endereço de memória.

Uma estrutura possível de implementar este tipo de mistura consiste em colocar, nos campos OID do registro, ao invés de incluir o endereço de banco de dados completo, apenas um endereço resumido (curto) e o deslocamento na página, e uma estrutura de dados auxiliar, contendo o mapeamento do endereço curto para o endereço de banco de dados completo. Este endereço curto não precisa possuir todos os bits necessários para o endereçamento em memória, mas apenas o suficiente para ser possível distinguir uma página de outra.

Quando o sistema for tentar acessar ou de-referenciar estes endereços, o mecanismo de falha de página do sistema operacional será acionado, e uma busca nesta tabela será feita. Encontrando o endereço completo de banco de dados a ser carregado, a página é carregada e seus endereços curtos alterados para o endereço real de memória encontrado, juntamente

com a tabela de mapeamento de endereços curtos para endereços reais de bancos de dados. Como tanto a tabela de tradução como os endereços curtos agora estarão contendo os mesmos valores, não é mais necessário desfazer esta tradução quando a página for novamente gravada em disco. Na Figura 2.4 é mostrado um exemplo de tradução de endereços feitos desta forma.

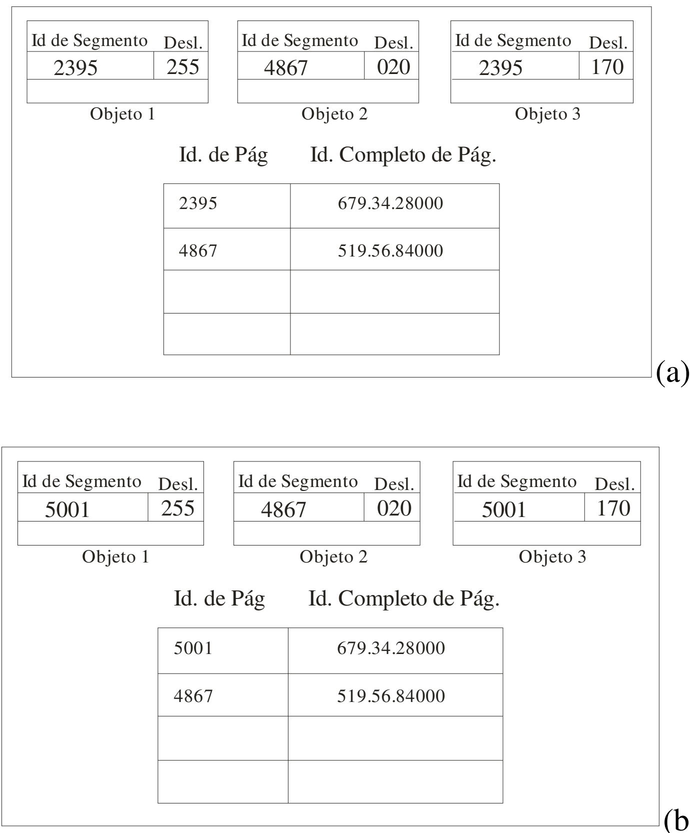



Figura 2.4. Exemplo de mistura de ponteiros por hardware. Em (a) temos a página antes da mistura, incluindo a tabela com os identificadores curtos e identificadores completos. Em (b) a página após a mistura, em que ela foi carregada na localização 5001. Tanto os ponteiros como o mapeamento foram alterados. Adaptado de Silberschatz [42].

#### 2.2.18.3 - Retorno dos blocos para o disco

Quando um bloco é movido da memória de volta para o disco, quaisquer ponteiros dentro desse bloco devem ser "desmisturados"; isto é, seus endereços de memória devem ser substituídos pelos endereços de bancos de dados correspondentes. Para evitar que, para cada pesquisa na tabela de conversão seja necessária uma busca em toda a tabela, é interessante também acrescentar um índice facilitando também pesquisas na forma "dado um endereço de memória, obtenha o endereço de banco de dados correspondente".

#### 2.2.18.4 - Registros e blocos fixados

Um bloco na memória é dito fixado (*pinned*) se ele não pode no momento ser gravado de novo no disco com segurança. Um bit informando se um bloco está fixado ou não pode ser acrescentado no cabeçalho do bloco, ou em outra estrutura de dados auxiliar. Existem muitas razões pelas quais um bloco poderia ser fixado, inclusive requisitos de um sistema de recuperação.

A mistura de ponteiros é uma razão importante para manter os blocos fixados em memória. Um bloco que possua ponteiros misturados em seu interior só pode ser gravado novamente em disco quando os mesmos forem "desmisturados". Para essa operação ocorrer, é necessário também a existência de uma estrutura de dados que informe onde há ponteiros misturados.

### 2.2.19 - Dados e Registros de Comprimento Variável

Um banco de dados cujos registros são basicamente de tamanho fixo muitas vezes não é adequado a aplicações de uso geral. Normalmente é necessário representar, dentro de um banco de dados:

1. **Itens de dados cujo tamanho varia**. Um exemplo notável são as cadeias de caracteres de tamanho variável;

2. **Campos repetidos**. Pode ser necessário armazenar um campo multivalorado, e não se sabe anteriormente qual a quantidade de repetições que o campo terá;

3. **Registros de formato variável**. Às vezes não é possível saber de antemão quais serão os campos de um registro ou quantas ocorrências de cada campo existirão.

4. **Campos com dados de grande tamanho**. Certos dados, como filmes codificados em MPEG, trechos de áudio e até mesmo documentos inteiros também podem ser armazenados em banco de dados, e, muitas vezes, estes dados são maiores que o tamanho de um bloco, e são necessárias estruturas de dados adicionais para representá-los.

#### 2.2.19.1 - Registros com campos de comprimento variável

Se um ou mais campos de um registro têm comprimento variável, então um registro deve conter informações suficientes para permitir localizar qualquer campo do registro.

Uma das abordagens pode ser colocar todos os campos de tamanho fixo no começo do registro, e, após esta região, inserir os campos de comprimento variável. Para localizar facilmente cada campo, no cabeçalho pode ser incluído o comprimento do registro e ponteiros (os deslocamentos a partir do início do registro) para o início de cada campo de comprimento variável.

#### 2.2.19.2 - Registros com campos repetidos

Uma situação semelhante acontece quando um registro contém um número variável de ocorrências de um campo *F*, mas o próprio campo tem comprimento fixo. É suficiente agrupar todas as ocorrências do campo *F* e inserir no cabeçalho do registro um ponteiro para a primeira ocorrência, além do número de ocorrências.

Outra alternativa é acrescentar os campos repetidos e/ou os campos de comprimento variáveis em outro bloco separado, mantendo no registro ponteiros para o lugar em que cada campo repetido inicia-se, e a quantidade de repetições existentes.

Utilizando esta estratégia de indireção, há vantagens e desvantagens:

- Com os registros sendo mantidos com comprimento fixo permite uma maior eficiência na pesquisa, minimizando o *overhead* dos cabeçalhos dos blocos, e permite a movimentação dos registros dentro dos blocos com menos esforço;

- A utilização de um bloco a mais para os campos de tamanho variável aumenta a E/S de disco quando for necessário recuperá-los.

Uma estratégia intermediária consiste em reservar espaço no registro para uma determinada quantidade de repetições, e o que exceder este tamanho é colocado em um bloco à parte.

#### 2.2.19.3 - Registros que não se encaixam em um bloco

Outro problema cuja importância vem aumentando são os valores que com freqüência não cabem em um único bloco, como clipes de áudio e vídeo, imagens e outros tipos de documentos ou arquivos.

A técnica chamada **registros de amplitude** consiste em colocar, no registro, ponteiros para partes do registro que podem estar dispostos em vários blocos. A parte do registro que aparece em um bloco é chamada de fragmento de registro. Um registro com dois ou mais fragmentos é chamado registro com faixas, e os registros que ocupam apenas um bloco são chamados de registro sem faixas.

Assim, para essa estrutura, algumas informações adicionais devem ser colocadas no dicionário, como um *flag* indicando se o registro é ou não um fragmento, e *flags* para informar se é o primeiro ou último fragmento do registro, e ponteiros para o próximo fragmento e para o fragmento anterior.

#### 2.2.19.4 - BLOBs

Os objetos binários extensos ou *BLOBS* (*binary large objects*) são valores extramente grandes, normalmente com tamanhos muito maiores que um bloco do banco de dados. Exemplos comuns são imagens em vários formatos, filmes, áudio e sinais em diversos formatos.

Os BLOBS devem ser armazenados em uma seqüência de blocos, os quais devem ser alocados preferencialmente de forma que a recuperação seja a mais rápida possível. Assim, pode ser interessante que os mesmos sejam armazenados os blocos em seqüência, ou divididos em diversos discos, aumentando a eficiência das operações de E/S.

Ao recuperar um BLOB, nem sempre o usuário irá desejar que todos os dados sejam transferidos inteiramente. É possível que o mesmo queira pequenos pedaços de cada vez, independente do tamanho do registro completo. Assim, são necessárias estruturas de índice que permitam o deslocamento dentro do BLOB sem ter que recuperar todos os blocos que o contém.

### 2.2.20 - Modificações de Registros

Muitas vezes, inserções, exclusões e atualizações de registros geram problemas especiais. Quando os registros possuem comprimento variável, cuidados adicionais devem ser tomados, mas há problemas mesmo com registros de comprimento fixo.

#### 2.2.20.1 - Inserção

Na inserção de novos registros em uma relação, ou de novos objetos na extensão de uma classe, se os registros não são mantidos em nenhuma ordem particular, pode-se simplesmente procurar espaço vazio em algum bloco, ou alocar um novo bloco para este registro.

Quando é necessário que as tuplas estejam em uma ordem específica, antes de tudo deve-se procurar o bloco apropriado para o registro, e, então, encontrar a posição adequada dentro do bloco.

Caso haja espaço disponível dentro do bloco, o registro é simplesmente inserido no mesmo, deslizando os outros registros conforme apropriado. Os novos dados são copiados para o bloco, e uma nova entrada na tabela de deslocamentos é adicionada.

Entretanto, pode ocorrer de não haver espaço disponível no bloco. Neste caso, uma das duas estratégias abaixo pode ser adotada:

1. **Encontrar espaço em um bloco vizinho**. Nesta estratégia, os blocos “vizinhos” são pesquisados, e, caso haja espaço em um deles, registros são movidos para os mesmos, de forma a permitir que o registro caiba no bloco em questão. Caso hajam ponteiros externos para os registros movidos, endereços de encaminhamento podem ser acrescentados na tabela de deslocamentos, informando para quais blocos os registros foram movidos; ou

2. **Criar um bloco de *overflow***. Com esta estratégia, cada bloco possui um ponteiro para um outro bloco de *overflow*, onde são colocados registros adicionais que não couberam no bloco inicial. Neste caso, não há necessidade de um endereço de encaminhamento, uma vez que o bloco de *overflow* é parte do bloco principal. Neste caso, a estrutura resultante apresenta-se como uma lista ligada de blocos.

#### 2.2.20.2 - Exclusão

Ao excluir um registro, o espaço armazenado por ele pode ser recuperado. Utilizando uma tabela de deslocamentos no início do bloco, como sugerido anteriormente, é possível deslocar os registros remanescentes, e um novo espaço em branco se tornará disponível no meio do bloco.

Caso um registro seja excluído e seja utilizada a técnica de blocos de *overflow*, pode-se deslocar registros dentro da cadeia de blocos de *overflow*, reduzindo, assim, a quantidade de blocos através de uma reorganização completa da cadeia.

Uma complicação adicional surge na exclusão, que consiste em, caso haja ponteiros para este registro, evitar que os mesmos apontem para endereços errados do banco de dados. Uma técnica que pode ser utilizada consiste na inserção de uma *lápide* no local do registro excluído.

#### 2.2.20.3 - Atualização

Na atualização de registros de tamanho fixo, não há nenhum efeito colateral sobre o armazenamento dos dados, uma vez que o novo registro ocupará o mesmo espaço de antes da atualização. Entretanto, na atualização de um registro de tamanho variável, temos os mesmos problemas da inserção e exclusão, exceto pelo fato de não ser necessário marcar com uma lápide que o registro foi excluído.

Caso os novos dados do registro sejam menores que o espaço ocupado anteriormente, será necessário deslizar os dados, deixando o espaço vazio no meio do bloco. Em caso do uso de blocos de *overflow*, pode ser que neste momento seja possível uma reorganização da cadeia, reduzindo a quantidade de blocos necessários.

Na situação de o novo registro ocupar mais espaço que o anterior, é necessário verificar se os dados do registro ainda cabem no mesmo bloco, e, caso não caibam, será necessário mover registros ou para blocos vizinhos ou para blocos de *overflow*.

## 2.3 - ESTRUTURAS BÁSICAS DE ARQUIVO

Muitos bancos de dados são armazenados permanentemente em armazenamento secundário. Segundo Elmasri [8], para estas formas de armazenamento, há diversas formas

de organizações de arquivo que tornam eficientes as operações mais utilizadas, sejam elas a inserção, pesquisa ou alteração de registros.

A **organização primária** do arquivo define como os registros de um arquivo são posicionados fisicamente no disco, e, também, como eles podem ser acessados. Um arquivo *heap* (ou desordenado) posiciona os registros sem nenhuma ordem específica, acrescentando os novos registros no final do arquivo. Já um arquivo *sorted* (ordenado ou seqüencial) mantém os registros ordenados segundo o valor de um campo específico, chamado **chave de ordenação**. Um arquivo *hashed* utiliza uma função *hash* aplicada a um campo para determinar a posição do registro no arquivo.

Outras estruturas em árvore também podem ser aplicadas como organização primária, como as árvores-B e suas variações. Estas estruturas são particularmente relevantes em bancos de dados orientados a objetos.

O objetivo de uma boa organização de arquivo é localizar um bloco que contenha um registro desejado com o menor número possível de transferências de bloco

### 2.3.1 - Buffering de blocos

Ao transferir diversos blocos do disco para a memória principal, vários *buffers* de memória são reservados para estes blocos. Assim, enquanto um *buffer* estiver sendo lido ou escrito, a CPU pode trabalhar com os dados em outro *buffer*. O *buffer* é muito útil para processos executados concorrentemente em paralelo, utilizando um processador para E/S (ou E/S assíncrona, utilizando de dispositivos capazes de acesso direto à memória), e outro para o processamento dos dados.

Com esta estratégia, é possível a CPU iniciar o processamento de um bloco que esteja na memória principal. Ao mesmo tempo, o processador de E/S pode ler e transferir o próximo bloco para um *buffer* diferente. Esta técnica é chamada de **buffering duplo**. Utilizando-a, os dados estão sempre disponíveis para o processamento, reduzindo o tempo de espera.

Segundo Folk [10], um esquema de bufferização oferece os melhores resultados se a chance de solicitar uma página que já estiver em um *buffer* for maior que a de uma página que não esteja. Quando uma página é solicitada e a mesma não está na memória principal, ocorre a chamada **falha de página**, e há duas causas principais:

1. A página nunca foi carregada em memória; e
2. A página estava em memória e foi substituída por outra.

No primeiro caso, a falha de página é inevitável, pois o primeiro uso da página com certeza precisará trazê-la a partir do armazenamento secundário. No segundo caso é onde as falhas de página devem ser evitadas utilizando-se técnicas de **gerenciamento de buffer**. Quando todas as páginas de buffer estão cheias, decidir qual das páginas deve ser descartada para ler uma nova a partir do armazenamento secundário é uma decisão crítica.

Uma política de substituição de páginas comumente utilizada é LRU (*Least Recently Used* – Utilizada Menos Recentemente). Este algoritmo consiste em manter um acompanhamento de todas as solicitações de páginas, e, caso seja necessário substituir uma delas, as páginas não solicitadas há mais tempo são escolhidas para descarte.

A principal característica do LRU é baseada na afirmativa da **localidade temporal**, ou seja, há um agrupamento do acesso a algumas páginas em um determinado período de tempo. Algumas estruturas de dados, como as árvores-B, que serão vistas em seguida, por possuírem uma natureza hierárquica, beneficiam-se bastante desta abordagem.

Outras políticas de substituição de páginas são baseadas na estrutura de arquivos sendo utilizadas, como, por exemplo, a utilização da altura de uma página como um fator de escolha da página a ser substituída, para estruturas de dados baseadas em árvore.

### 2.3.2 - Alocação de blocos de arquivo em disco

Há diversas técnicas para alocação dos blocos em disco. Na **alocação consecutiva** (*contiguous allocation*), os blocos são alocados consecutivamente no disco. Nesta técnica, a leitura de blocos consecutivos, utilizando a técnica do buffering duplo, é bastante eficiente. Entretanto, a expansão do arquivo pode ser dificultada por não ser sempre possível encontrar blocos disponíveis em seqüência.

Já na **alocação encadeada** (*linked allocation*), cada bloco do arquivo contém um ponteiro para o próximo bloco do arquivo. Nessa estrutura, a leitura de blocos é mais lenta, a cabeça do disco rígido pode precisar se mover para outras trilhas, mas a expansão do arquivo é mais fácil.

Uma abordagem híbrida consiste em alocar **segmentos de arquivo** ou **extensões** (*clusters*) e encadeá-los. Cada segmento consiste em um conjunto de blocos alocados seqüencialmente, e com um ponteiro para o próximo segmento.

Outra técnica é utilização de **alocação indexada**, em que há um ou mais blocos de índice, que contém ponteiros para os blocos do arquivo.

### 2.3.3 - Cabeçalhos de arquivo

Um **cabeçalho** ou **descritor de arquivo** contém informações sobre o arquivo, que são necessárias ao SGBD. O cabeçalho contém informações para determinar os endereços de disco dos blocos de arquivo, informações de formato e tamanho e o esquema dos campos dos registros deste arquivo.

### 2.3.4 - Operações em Arquivos

As operações em arquivo são geralmente divididas em **operações de recuperação** e em **operações de atualização**. As operações de recuperação não alteram os dados persistentes, enquanto que as de atualização mudam os dados, através da inclusão, exclusão e alteração de registros ou de valores dos campos destes registros. Em ambos os casos é necessário aplicar filtros para encontrar, nos arquivos de dados, os registros que serão recuperados ou atualizados.

As operações de seleção são normalmente simples, e condições complexas precisam ser decomposta pelo SGBD em uma série de operações simples.

Segundo Elmasri e Navathe [8], as seguintes operações podem ser feitas em um arquivo de dados:

- *Open* (abrir): Prepara um arquivo de dados para leitura ou escrita, posicionando o ponteiro de registro corrente no início do arquivo;

- *Reset* (reiniciar): Recoloca o ponteiro de registro corrente para o início do arquivo;

- *Find* ou *Locate* (encontrar ou localizar): Busca o primeiro registro que satisfaz uma condição de pesquisa, trazendo o bloco que o contém para um buffer de memória principal. O registro localizado se torna o registro corrente;

- *Read* ou *Get* (ler ou obter): Copia o registro corrente para uma variável de usuário, podendo ou não avançar para o próximo registro do arquivo;

- *FindNext* (encontra o próximo): Procura o próximo registro que satisfaz a condição de pesquisa, trazendo o bloco que o contém para um buffer. O ponteiro de registro corrente é atualizado para este registro;

- *Delete* (excluir): Exclui o registro atual, alterando também o arquivo em disco;

- *Modify* (modificar): Modifica alguns valores de campo do registro atual e confirma estas alterações em disco;

- *Insert* (incluir): Acrescenta um novo registro no arquivo, localizando o bloco no qual o mesmo deve ser inserido e atualizando o arquivo em disco com o novo bloco; e

- *Close* (fechar): fecha o arquivo de dados, liberando todos os *buffers* utilizados por este arquivo.

Há uma diferença que deve ser levada em consideração entre os termos organização de arquivo e métodos de acesso. A **organização de arquivo** consiste na forma com a qual os dados são distribuídos em registros, blocos e estruturas de acesso, incluindo a forma como os registros e blocos são interligados nos arquivos. Já o **método de acesso** fornece o conjunto de operações citadas acima. Algumas vezes é possível utilizar vários métodos de acesso para a mesma organização de arquivo; outras vezes é necessário que o arquivo esteja organizado de uma certa maneira para a utilização de um método de acesso específico.

### 2.3.5 - Arquivos de registros desordenados (*heap files*)

Esta organização, a mais básica para um arquivo de dados, é chamada ***heap file*** ou ***pile file***. Nesta organização, incluir um novo registro é muito eficiente, bastando copiar o último bloco do arquivo em um *buffer*, acrescentar o registro e reescrever o bloco em disco. Entretanto, a pesquisa por um registro, usando qualquer condição, envolve uma pesquisa seqüencial bloco a bloco do arquivo - um procedimento dispendioso.

Para excluir um registro, um programa deve primeiro encontrar seu bloco, copiá-lo em um buffer, então excluir o registro do *buffer* e, finalmente, reescrever o bloco de volta no disco. Isso deixa espaços sem uso no bloco. A exclusão de um grande número de registros resulta, dessa forma, em desperdício de espaço de armazenamento. Outra técnica utilizada para a exclusão de registros é possuir um byte ou bit extra, chamado marcador de exclusão ou lápide. Programas de pesquisa consideram apenas os registros válidos de um bloco quando executam sua busca. Ambas as técnicas de exclusão exigem a reorganização periódica do arquivo para reaproveitar o espaço sem uso dos registros excluídos. Outra possibilidade é utilizar o espaço dos registros excluídos durante a inclusão de novos registros, embora isso exija controles adicionais para manter informações sobre as localizações vazias.

### 2.3.6 - Arquivos de Registros Ordenados (*Sorted Files*)

Podemos ordenar fisicamente os registros de um arquivo em um disco a partir dos valores de um de seus campos – chamado **campo de classificação.** Isso leva um arquivo **ordenado** ou **seqüencial.** Se o campo de classificação também for um campo chave do arquivo, então o campo é chamado de **chave de classificação.** Primeiro, a leitura dos registros na ordem dos valores da chave de classificação se torna extremamente eficiente porque nenhuma classificação se faz necessária. Segundo, encontrar o próximo registro a partir do atual, na ordem dos valores da chave de classificação, geralmente não requer acessos a outros blocos porque o próximo registro pode estar no mesmo bloco que o atual. Terceiro, o uso de uma condição de pesquisa baseada no valor do campo chave de classificação resulta em um acesso mais rápido quando a técnica de pesquisa binária utilizada, o que constitui uma melhoria em relação à pesquisa linear, embora não seja frequentemente utilizada em arquivos de disco.

Classificar não traz nenhuma vantagem em relação ao acesso aleatório ou ordenado aos registros com base em valores dos demais campos não ordenados do arquivo. Nesses casos, será necessário executar uma pesquisa linear.

A inclusão e a exclusão de registros são operações dispendiosas para um arquivo ordenado, pois os registros deverão permanecer ordenados fisicamente. Para inserir um registro, a primeira etapa consiste em encontrar sua posição correta no arquivo, de acordo com seu valor no campo de classificação e, então abrir espaço no arquivo para inserir o registro

naquela posição. Isso significa que, em média, a metade dos blocos de arquivos deverá ser lida e regravada após os registros serem transferidos entre eles. Para exclusão de um registro, o problema será menos grave se forem utilizados marcadores de exclusão e se forem feitas reorganizações periódicas.

Uma opção para tornar a inclusão mais eficiente é manter alguns espaços não utilizados em cada bloco para novos registros. Entretanto, uma vez que esse espaço seja utilizado, o problema original ressurge. Um outro método freqüentemente utilizado é criar um arquivo desordenado temporário, chamado **arquivo de transação**. Os novos registros são inclusos ao final do arquivo de transação em vez de serem inclusos em suas posições corretas no arquivo principal. Periodicamente o arquivo de transação é classificado é incorporado ao arquivo mestre durante a reorganização do arquivo.

### 2.3.7 - Abordagens mistas

Outra abordagem consiste na utilização de um arquivo em que, no interior de suas páginas, os registros estejam armazenados de forma ordenada, e acrescenta-se ao cabeçalho da página um ponteiro para o próximo registro na seqüência.

Esta técnica permite que as operações de inserção e atualização ocorram com menor custo, no caso do preenchimento e esvaziamento de um bloco, pois envolvem apenas atualizações de ponteiros. Entretanto, estruturas adicionais precisam ser colocadas de forma a facilitar a indexação destes blocos, para permitir a pesquisa de forma eficiente. Neste caso, estruturas de índices esparsos podem ser utilizados, para indicar a primeira chave de classificação de cada um dos blocos. Um exemplo de estrutura de bloco que leva em consideração esta abordagem é mostrada na Figura 2.5.

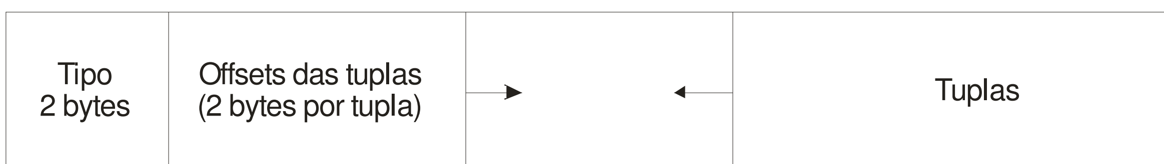



Figura 2.5. Exemplo estrutura de uma página.

### 2.3.8 - Técnicas de hashing

Um outro tipo de organização primária baseia-se em *hashing*, que fornece um acesso muito rápido aos registros sob certas condições de pesquisa. Essa organização é geralmente chamada **arquivo *hash***. A condição de pesquisa precisa ser de igualdade em um único campo; neste caso, o campo é chamado **campo de *hash***. Caso este seja também o campo chave do arquivo, é chamado de **chave de *hash***. A idéia do *hashing* é fornecer uma função *h*, chamada **função *hash*** ou **função espalhamento**, que aplicada ao valor do campo de *hash* de um registro, gere o endereço do bloco de disco no qual o registro está armazenado. Para a maioria dos registros, necessita-se de apenas do acesso a um bloco para recuperá-lo.

Valores de campos de *hash* não inteiros podem ser transformados em inteiros antes que a função *hash* seja aplicada. Para cadeias de caracteres, o código numérico (ASCII*)*, associado aos caracteres, pode ser utilizado na transformação. O problema com a maioria das funções *hash* é que elas não garantem que diferentes valores levarão a diferentes endereços *hash* porque o espaço de variação de campo de *hash* é geralmente muito maior que o espaço de endereços. A função *hash* mapeia o espaço de variação do campo de *hash* no espaço de endereços.

Uma **colisão** ocorrerá quando o valor do campo de *hash* de um registro que está sendo incluído levar a um endereço *hash* que já contiver um registro diferente. O processo para encontrar outra posição é chamado **resolução de colisão**. Há vários métodos para a resolução de colisão:

- ***Open addressing* (endereçamento aberto)**: A partir da posição já ocupada pelo endereço *hash*, o programa prossegue a verificação, pela ordem, das posições subseqüentes, até seja encontrada uma posição vazia.

- ***Chaining* (encadeamento)**: Neste método são mantidas várias posições de *overflow,* geralmente por meio da extensão do vetor por um número de posições de *overflow*. Uma colisão é resolvida posicionando o novo registro em uma localização de *overflow* não utilizada e colocando o endereço de *overflow* no ponteiro do endereço *hash* ocupado.

- ***Hashing* múltiplo**: O programa aplicará uma segunda função *hash* caso a primeira resulte uma colisão. Se novamente ocorrer uma colisão, o programa usará *open*

*addressing* ou aplicará uma terceira função *hash*, usando *open addressing* se necessário.

Cada método de resolução de colisão requer seus próprios algoritmos para inclusão, recuperação e exclusão de registros.

O objetivo de uma boa função *hash* é distribuir os registros uniformemente pelo espaço de endereços de forma a minimizar as colisões e não deixar endereços sem uso.

### 2.3.9 - *Hashing* externo para arquivos em disco

*Hashing* para arquivos em disco são chamados ***hashing* externos**. Considera-se que o espaço de endereço alvo é construído de ***buckets* (baldes)**, cada qual mantendo múltiplos registros. Um *bucket* é um bloco de disco ou um grupo (*cluster*) de blocos consecutivos, contendo **m** registros. A função *hash* mapeia uma chave a um número de *bucket* relativo em vez de atribuir um endereço absoluto de bloco para o *bucket.* Embora as melhores funções *hash* não mantenham registros na ordem dos valores do campo de *hash,* algumas funções – chamadas funções **que preservam a ordem** – o fazem.

O esquema *hashing* descrito é chamado ***hashing* estático** porque o número fixo M de *buckets* será alocado. Isso pode ser um sério inconveniente para arquivos dinâmicos, pois no máximo (m * M) registros caberão no espaço alocado. Se o número de registros for maior que (m * M), haverão colisões, e a recuperação de registros se tornará mais lenta por conta das longas listas de registros de *overflow*. Talvez haja necessidade de trocar o número M de blocos alocados e, então, utilizar uma nova função *hash* (baseado no novo valor de M). Essas reorganizações podem consumir bastante tempo para arquivos grandes. Organizações de arquivos dinâmicas, baseadas em *hashing*, permitem que o número de *buckets* varie dinamicamente utilizando apenas reorganizações localizadas.

A pesquisa de um registro pelo valor de algum campo diferente do campo de *hash* é tão dispendiosa quanto no caso de um arquivo desordenado. A exclusão de registros pode ser implementada por meio da remoção do registro de seu *bucket.* Se o *bucket* possui um encadeamento de *overflow*, podemos transferir um dos registros de *overflow* para o *bucket,* em substituição ao registro excluído.

### 2.3.10 - Técnicas de *hashing* que permitem a expansão de arquivos dinâmicos

A maior desvantagem do esquema de ***hashing* estático** é que o espaço de endereços *hash* é fixo. Desse modo é difícil expandir ou diminuir o arquivo dinamicamente. O ***hashing* extensível** armazena uma estrutura adicional ao arquivo e, por isso, a estrutura de acesso é baseada nos valores resultantes após a aplicação da função *hash* no campo de pesquisa. Outra técnica, chamada ***hashing* linear**, não exige estruturas de acesso adicionais.

Esses esquemas de *hashing* tiram vantagem do fato que o resultado da aplicação de uma função *hash* é um número inteiro não negativo, podendo ser representado como número binário. Os registros são distribuídos entre os *buckets,* tendo como base os valores dos primeiros *bits* de seus valores *hash.*

#### 2.3.10.1 - *Hashing* extensível

No *hashing* extensível, uma estrutura de **diretório –** um vetor de $2^d$ endereços de *bucket* – é mantido, onde *d* é chamado **profundidade global** do diretório. O valor inteiro correspondente aos primeiros (de mais alta ordem) *d* bits de um valor *hash* é utilizado como um índice de vetor para determinar uma entrada no diretório, e o endereço ali armazenado determina o *bucket* no qual os registros correspondentes serão armazenados. Entretanto, não é necessário que haja um *bucket* diferente para cada uma das $2^d$ localizações. Diversas localizações de diretório com os mesmos primeiros d' bits para seus valores *hash* podem conter o mesmo endereço de *buckets* se todos os registros que a função *hash* levar para essas localizações couberem em um único *bucket.* Uma **profundidade local d' –** armazenada em cada *bucket* – especifica o número de bits no qual os conteúdos dos *buckets* são baseados.

O valor de *d* pode ser aumentado ou diminuído uma unidade por vez, assim duplicando ou dividindo ao meio o número de entradas no vetor diretório. A duplicação ocorre quando houver um *overflow* em um *bucket* cuja profundidade local *d'* seja igual à profundidade global *d.* A divisão ao meio ocorre se $d > d'$ para todos os *buckets* após a ocorrência de algumas exclusões. A maioria das recuperações de registros exige dois acessos a blocos: um para o diretório e outro para o *bucket.*

A principal vantagem do *hashing* extensível é que o desempenho do arquivo não degrada conforme o arquivo cresce, em oposição ao *hashing* externo estático, no qual as colisões

aumentam e o encadeamento correspondente causa acessos adicionais. Além disso, nenhum espaço será alocado no *hashing* extensível para crescimento futuro, mas *buckets* adicionais podem ser alocados dinamicamente se necessário. A divisão do *bucket* causará menor número de reorganizações, uma vez que apenas os registros de um *bucket* serão redistribuídos para os dois novos *buckets.* A única reorganização mais dispendiosa será a que for feita quando o diretório tiver de ser duplicado (ou dividido ao meio).

2.3.10.2 - *Hashing* linear

A idéia do *hashing linear* é permitir que um arquivo *hash* expanda ou diminua seu número de *buckets* dinamicamente, sem necessitar de um diretório. Suponha que o arquivo comece com M *buckets* numeradas de 0, 1, ..,M – 1 e use a função *hash mod h(k)=K mod M* como função *hash* inicial, chamada $h_i$. O *overflow* por causa de colisões ainda será necessário e poderá ser tratado por meio da manutenção de cadeias individuais para cada *bucket*. Entretanto, quando uma colisão levar a um registro de *overflow* em qualquer *bucket* do arquivo, o primeiro *bucket* do arquivo – o *bucket* 0 – será dividido em dois *buckets*: o *bucket 0* original e o *bucket M*, no final do arquivo, e os registros no *bucket 0* original serão divididos entre os dois novos buckets, utilizando outra função *hash* $h_{i+1}(K)=mod\ h(k)=K\ mod\ 2M$. A principal propriedade desta função de *hash* é que os valores que anteriormente eram levados ao *bucket* 0 pela função $h_i$, serão agora levados ou ao *bucket 0* ou ao *bucket M* pela função $h_{i+1}$.

À medida que novas colisões forem ocorrendo, novos *buckets* serão criados, dividindo sucessivamente os *buckets* 1, 2, 3, ..., M-1. Ao final, todos os M *buckets* do arquivo terão sido divididos, e os blocos de *overflow* terão se transformado em novos *buckets*. Neste sistema, não há um diretório, e sim apenas um valor *n*, que inicia com 0 e indica quantas divisões já ocorreram – isso é utilizado para determinar quais os *buckets* já foram divididos.

A operação de recuperação de um registro consiste em, inicialmente, calcular o valor *hash* da chave utilizando a função $h_i$. Caso o $h_i(K) < n$, então deve-se aplicar a função $h_{i+1}(K)$, pois sabemos que o *bucket* foi dividido.

Ao incrementar o valor de $n$, caso se obtenha a condição de que $n=M$, sabemos que todos os *buckets* foram divididos, e então seu valor deve ser reiniciado para 0 e a função de hash principal deve ser alterada para $h_{i+1}$.

### 2.3.11 - Outras organizações primárias

#### 2.3.11.1 - Arquivos com Registros Mistos

As organizações de arquivos em SGBDs orientados a objeto, bem como em sistemas legados, tais como SGDBs hierárquicos e SGDBs de rede, freqüentemente implementam relacionamentos entre registros por meio de relacionamentos físicos obtidos por meio da adjacência física (ou *clustering*) dos registros relacionados ou por ponteiros físicos. Essas organizações de arquivo normalmente atribuem uma área em disco para manter os registros de mais de um tipo, de modo que registros de tipos diferentes possam ficar **fisicamente agrupados (*clustered*)** no disco. Se for esperada a utilização freqüente de um determinado relacionamento, a implementação física do relacionamento poderá aumentar a eficiência do sistema na recuperação de registros relacionados.

Para distinguir os registros em um arquivo misto, cada registro possui – além dos valores de seus campos – um campo de **tipo de registro**, que especifica o tipo do registro. Normalmente esse é o primeiro campo de cada registro e é utilizado pelo *software* do sistema para determinar o tipo de registro que ele está prestes a processar. Usando a informação do catálogo, o SGBD pode determinar os campos de cada tipo de registro e seus tamanhos, com o objetivo de interpretar os valores dos dados no registro.

## 2.4 - ESTRUTURAS DE ÍNDICE

Anteriormente foram mostradas as estruturas necessárias para o armazenamento de registros e blocos. Entretanto, também é necessário que tais blocos estejam organizados de forma a facilitar a busca do bloco apropriado para uma inserção, consulta, alteração ou exclusão.

Segundo Garcia-Molina [12], uma das formas de organizar os blocos e registros em uma relação ou extensão de uma classe consiste no uso de estruturas de índice. Um *índice* é uma estrutura de dados que tem como entrada uma propriedade dos registros (normalmente um ou mais campos) e utiliza métodos para encontrar rapidamente esta propriedade.

Um índice consiste, segundo Michael J. Folk e Bill Zoelick [10], em pares de chaves e referências.

Diferentes estruturas de arquivo podem fazer o papel de índices, como os arquivos classificados, os índices secundários para arquivos não classificados, as árvores B e suas variações, e as tabelas de *hash*.

### 2.4.1 - Arquivos Seqüenciais Indexados

A estrutura de índice considerada mais simples é um arquivo classificado associado a um arquivo de índices, os quais contêm pares chave-ponteiro. Estes índices podem ser *densos*, em que todas as chaves existentes no arquivo de dados estão presentes, ou então *esparsos*, onde apenas algumas das chaves estão representadas, normalmente apenas um par chave-ponteiro por cada bloco existente no arquivo.

#### 2.4.1.1 - Arquivos seqüenciais

Este tipo de índice é baseado no fato de que o arquivo de dados é inerentemente classificado pelos atributos correspondentes do índice. Normalmente, neste arquivos, as linhas são classificadas pela chave primária, e as operações de inserção, exclusão e alteração levam em consideração tais características, mantendo o arquivo permanentemente classificado pela chave.

#### 2.4.1.2 - Índices densos

Uma vez que o arquivo de dados esteja classificado, pode-se construir sobre ele um índice denso, composto de uma seqüência de blocos contendo apenas as chaves dos registros e os endereços de banco de dados correspondentes. No índice chamado *denso*, há um ponteiro para cada um dos registros, armazenados na mesma seqüência que no arquivo de dados.

Esta estrutura é vantajosa quando o índice pode ser colocado completamente em memória, enquanto que o arquivo de dados não. Mesmo estando em disco, tal estrutura ainda é vantajosa, pois o número de blocos que precisa ser carregado, utilizando, por exemplo, a pesquisa binária, ainda é menor que se a busca for feita no arquivo de dados, pois os pares chave-ponteiro costumam ser menores que os registros completos.

#### 2.4.1.3 - Índices esparsos

Caso a utilização de um índice denso em uma relação ou extensão mostre-se desvantajosa, por, por exemplo, tal estrutura apresentar-se muito grande, não sendo possível colocá-la toda em memória, pode-se utilizar outra forma chamada de *índice esparso*. Nesta estrutura, há apenas um par chave-ponteiro para cada bloco de dados, correspondente ao primeiro registro do bloco.

Assim, para encontrar o registro com chave *K*, basta pesquisar o índice e encontrar o ponteiro correspondente à maior chave menor que ou igual a *K*. Em seguida, o bloco apontado deve ser carregado, e uma busca dentro do bloco pode ser feita, utilizando as estruturas de dados existentes no cabeçalho do mesmo.

#### 2.4.1.4 - Vários níveis de índice

Utilizando um índice esparso, é possível que uma única entrada possa cobrir vários blocos. Para isso, utiliza-se um índice a mais, acrescentando mais um nível de indireção. Assim, este "índice do índice" aponta para o bloco inicial de outro índice. Esta estrutura de dados permite acrescentar quantos níveis de indireção forem necessários, de forma a manter os níveis mais externos em memória.

É importante observar que, nesta estrutura, a utilização de um índice denso nos últimos níveis retira sua grande vantagem, já que, neste caso, todos os registros do arquivo de dados estariam indexados logo nos níveis mais altos, ocupando exatamente ou mais espaço que um apenas um índice denso utilizado para acessar diretamente os registros.

### 2.4.2 - Índices Secundários

Outro caso simples consiste em manter um arquivo desordenado, em que os registros são postos um em seguida do outro, ou organizados em bloco. No caso de um arquivo seqüencial, caso os registros possuam tamanho variável, mantê-los organizados de forma a utilizar uma pesquisa binária não é suficiente, já que é bastante difícil conhecer exatamente em que bloco ou ponto do arquivo é o ponto central do mesmo.

Assim, pode-se criar um campo chave, de tamanho fixo, e utilizá-lo para indexação. Com este campo, utilizamos um **arquivo de índice**, contendo o campo chave e a referência para o início do registro, que pode ser o *offset* dentro do arquivo de dados, ou mesmo o

endereço de banco de dados do registro em questão. Como a chave possui tamanho fixo, é possível utilizar métodos de busca binária no índice em questão.

Um índice secundário serve também para a localização de registros, mas distinguindo-se do índice principal no sentido em que ele não determina a ordem dos registros no arquivo de dados. O índice secundário é utilizado para informar a localização das tuplas no registro principal, mas a ordem em que os dados ocorrem no índice secundário é dependente do campo escolhido para o mesmo.

Devido a estas características, algumas ressalvas são importantes de serem feitas em um índice secundário:

1. Não há como ter um índice secundário esparso, pois o mesmo não influencia na localização dos registros no arquivo de dados, e, assim, não é possível prever a ordem dos registros no arquivo de dados, em relação ao campo sendo indexado; e

2. No caso da utilização de indexação em vários níveis, é necessário que o primeiro nível seja denso, pelo motivo citado anteriormente.

#### 2.4.2.1 - Caráter indireto em índices secundários

Nos casos em que há repetição de dados no campo sendo indexado, pode ser interessante criar uma estrutura de dados intermediária, como um índice denso, no primeiro nível da indexação. Assim, pode-se poupar espaço nos níveis mais externos. É importante ver que tal estrutura também pode ser vantajosa nos casos em que não ocorra repetição, se for possível manter os índices de níveis maiores em memória, permitindo, por exemplo, pesquisas binárias.

### 2.4.3 - Operações em um arquivo indexado

Deve-se sempre lembrar que uma operação em um arquivo indexado envolve, na verdade, dois arquivos: tanto o arquivo de dados como o arquivo de índice. Assim, cuidados adicionais devem ser tomados para evitar que os dois arquivos fiquem foram de sincronia, e também ferramentas adicionais para a detecção deste tipo de situação, permitindo que o índice seja reconstruído com os dados presentes no arquivo de dados.

#### 2.4.3.1 - Adição de registros

A adição de um novo registro requer também a adição de uma nova entrada no registro de índice, além da entrada no registro de dados.

Ao acrescentar no arquivo de dados, podem ser utilizadas técnicas tradicionais para o acréscimo de dados em arquivos não indexados (por exemplo, acrescentar o registro no primeiro bloco em que haja espaço disponível).

Já no arquivo de índice, inicialmente uma pesquisa por chave deve ser feita, a fim de encontrar a posição correta no mesmo, e, dependendo da estrutura utilizada, deslocar os registros corretamente no índice.

#### 2.4.3.2 - Exclusão de registros

De forma similar à adição de registros, também na exclusão é necessário efetuar a alteração tanto no índice quanto no arquivo de dados. As técnicas explicitadas anteriormente podem ser aplicadas para a reutilização de espaço nos blocos do arquivo de dados. No arquivo de índice pode ser necessário o deslocamento dos registros já existentes, a fim de melhorar a estrutura para pesquisas subseqüentes.

#### 2.4.3.3 - Atualização de registros

Há dois casos para a atualização de registros:

1. *A atualização não altera o valor da chave*. Neste caso, não é necessário reorganizar o índice, apenas o arquivo de dados. Também as técnicas descritas no capítulo anterior se adequam para esta alteração, mas é importante notar que o endereço de banco de dados do registro ou objeto sendo alterado não deve ser modificado. Caso o mesmo seja modificado, esta alteração deve ser refletido no índice

2. *A atualização altera o valor da chave*. Aqui talvez seja necessário reordenar o arquivo de índice, excluindo a antiga entrada e acrescentando a nova. Apesar de estarem combinadas uma exclusão e uma inclusão, para o usuário final a operação é considerada como sendo uma única operação atômica.

### 2.4.4 - Estruturas de arquivo utilizadas em índices

Apesar de estruturas de índice poderem favorecer enormemente o desempenho em uma operação de pesquisa, suas alterações podem ser bastante custosas, em termos de CPU e/ou E/S. Por exemplo, para adicionar ou remover registros em um arquivo de índice com as chaves ordenadas sequencialmente, pode ser necessário, no pior dos casos, reescrever o arquivo inteiramente. Além disso, métodos que possuem boa performance em memória, como a busca binária, não são tão eficientes quando operando em armazenamento secundário.

Assim, quando as estruturas de índice forem maiores do que o possível de ser armazenado em memória, é importante que se utilizem algoritmos adequados para a operação em armazenamento secundário, como:

- Índices em árvore, como as árvores-B e suas variantes, para os casos em que é importante que se mantenham o acesso tanto através de chaves como o acesso seqüencial, em ordem; ou

- Índices *hash*, quando a velocidade for de principal importância.

A utilização de arquivos seqüenciais para a busca por campos chave, utilizando algoritmos como a pesquisa binária, nem sempre é vantajosa, pois há alguns problemas específicos:

- *A pesquisa binária precisa de muitas operações de E/S*. Quando for necessário buscar dados em armazenamento secundário, uma pesquisa que envolva mais de três ou quatro operações de E/S, que possam envolver deslocamento da cabeça do disco, normalmente resultam em um tempo de espera maior do que o desejado.

- *Manter um arquivo indexado é muito caro*. Para se fazer uma pesquisa binária é necessário que o índice esteja ordenado, ou seja, todos os blocos do mesmo estejam em seqüência, para que seja possível encontrar os blocos do meio em cada segmento da pesquisa. Assim, a inserção pode custar em média a movimentação da metade dos registros de um arquivo, e a exclusão, caso não seja utilizada a técnica mostrada anteriormente de colocação de lápides, também pode resultar nesta mesma quantidade de movimentação de dados.

2.4.4.1 - Árvore de pesquisa Binária

Para resolver o segundo problema, pode ser utilizada uma estrutura conhecida como árvore de pesquisa binária. Nesta estrutura, os nós contêm ponteiros para os elementos à esquerda e à direita, e o nó raiz consiste exatamente no elemento do meio, que seria utilizado para o início de uma pesquisa binária. Assim, a árvore é construída como uma estrutura ligada, em que os ponteiros à esquerda e à direita apontam para os nós filhos. Uma árvore de pesquisa binária é mostrada na Figura 2.6.

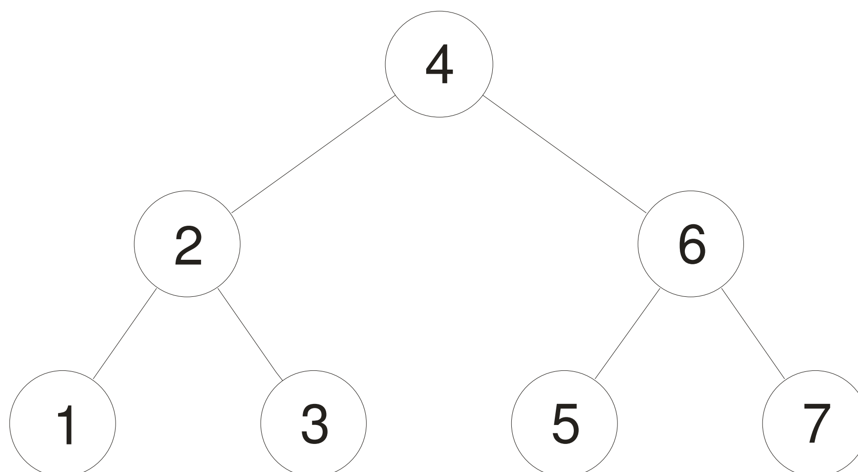



Figura 2.6. Árvore de pesquisa binária

A pesquisa em uma árvore desta forma é feita da mesma forma que uma pesquisa binária, sendo apenas diferente em que a grande desvantagem da pesquisa binária – a necessidade de o arquivo de dados estar ordenado – é eliminada. As operações de inserção e exclusão são bastante eficientes.

Entretanto, apesar destas características, as árvores de pesquisa binária ainda se mostram ineficientes para grandes volumes de dados. O primeiro fato é que é necessário que se façam $log_2 n$ operações de E/S, em média, para recuperar um registro. O segundo problema diz respeito ao *balanceamento* da árvore. Dizemos que uma árvore está *balanceada* quando a menor e a maior altura de um ramo da árvore não diferem por mais de uma unidade. A árvore também pode estar *completamente balanceada*, o que representa que as alturas de todos os ramos são a mesma.

Entretanto, de acordo com a ordem que os valores são acrescentados na árvore, pode ocorrer de a mesma não resultar balanceada, e, neste caso, os custos da pesquisa podem ser

bastante altos, como no caso da árvore degenerada formada quando os elementos são acrescentados em ordem crescente ou decrescente, como mostrado na Figura 2.7.

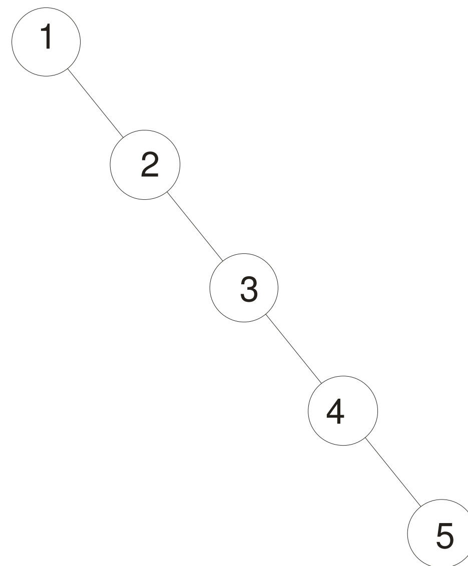



Figura 2.7. Árvore degenerada

2.4.4.2 - Árvores AVL

As árvores AVL provêem uma solução para o caso da árvore degenerada, fornecendo métodos para uma reorganização dos nós, assim que novos elementos são adicionados. As árvores AVL possuem a altura equilibrada (*height-balanced*), ou seja, há um limite de diferença entre a altura de duas sub-árvores que compartilham a mesma raiz. Com o algoritmo proposto nas árvores AVL, este limite é 1.

Com essas características, as árvores AVL possuem bom desempenho de pesquisa, e a manutenção da altura da árvore envolve, durante as operações de inserção, o uso de uma rotação de um conjunto de quatro possíveis. Estas rotações possuem a característica de serem confinadas a regiões locais da árvore.

Entretanto, apesar de possuírem um bom desempenho para leitura e inserção de objetos, as árvores AVL, no pior caso, para uma árvore com N registros, chega a

$$1{,}44\log_2(N+2)$$

níveis. Assim, para índices bastante grandes, ainda podem ser necessárias muitas operações de E/S de disco. Pode-se ver que, em uma árvore AVL, suas características de modo de

operação e desempenho se aproximam bastante de uma pesquisa binária, sem, no entanto, ser necessário manter o arquivo de índice classificado. Ainda assim, a pesquisa binária ainda precisa de bastantes operações em disco, tornando-a inviável para a utilização em armazenamento secundário.

#### 2.4.4.3 - Árvores Binárias Baseadas em Páginas

Ainda que as árvores AVL forneçam um bom desempenho para pesquisa e inclusão, em que não é necessário manter o arquivo de dados permanentemente em ordem seqüencial, ainda são necessárias muitas operações de E/S para encontrar o registro desejado, tornando esta estrutura de organização de dados pouco funcional em armazenamento secundário.

Uma vez que os acessos ao armazenamento secundário normalmente envolvem altas latências para o início da transferência, e a transferência de dados em blocos, tem-se naturalmente a noção de paginação. Utilizando-se desta técnica, a quantidade de operações de E/S para a recuperação de um registro pode ser reduzida, mantendo os registros adjacentes próximos, conforme mostrado na Figura 2.8.

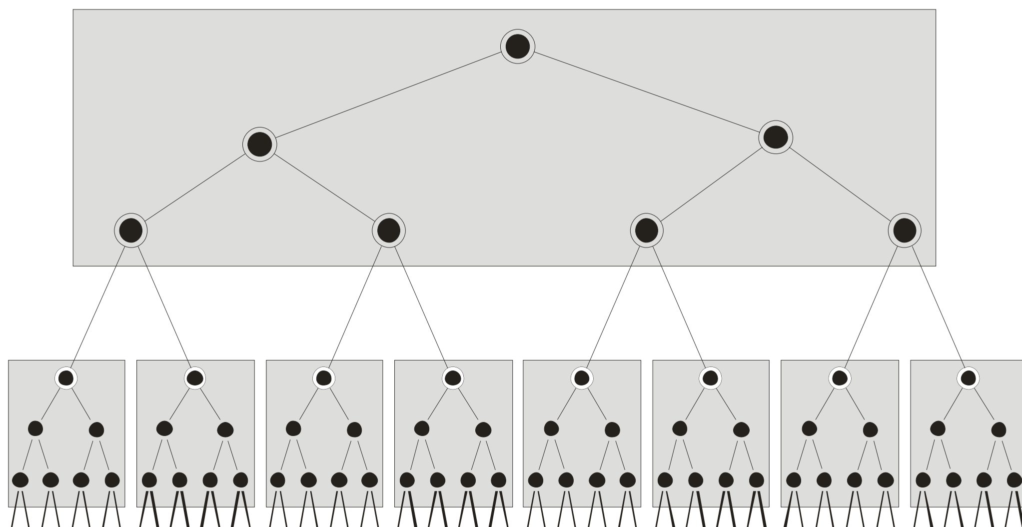


Figura 2.8. Árvore binária paginada, adaptado de Folk [9]

Organizando os dados desta forma, no pior caso de uma árvore paginada balanceada completa, para *k* elementos por página e *N* elementos na árvore, temos

$$\log_{k+1}(N+1)$$

níveis. Esta equação é a generalização do caso das árvores binárias, em que o número de elementos por página é 1. Vê-se que o comportamento logaritmo é obtido através da paginação, reduzindo drasticamente a quantidade de E/S necessárias.

Apesar de quebrar uma árvore em páginas se adapta bem às características dos meios de armazenamento secundário, ainda resta o problema da construção da mesma. Caso a lista de elementos já estiver em ordem antes de serem adicionados, a tarefa se torna fácil, basta iniciar a partir do meio da lista, sendo este o elemento raiz.

Entretanto, sabe-se que normalmente os elementos serão acrescentados na ordem em que os mesmos chegam ao SGBD, e isso normalmente ocorre em ordem aleatória. Assim, caso o elemento raiz não seja adequadamente escolhido, pode-se chegar ao mesmo caso de árvore degenerada, após diversas operações de inclusão e exclusão de objetos. Um exemplo é mostrado na Figura 2.9.

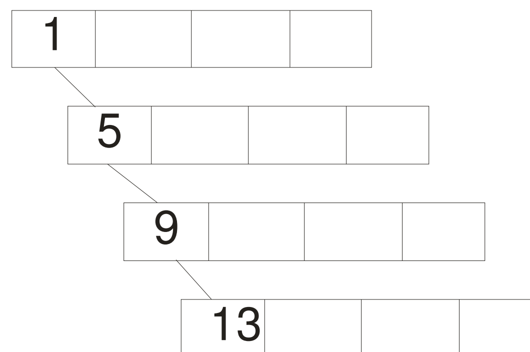



Figura 2.9. Árvore paginada degenerada. Neste caso, com árvore construída a partir do topo, todos os outros elementos ficaram à direita.

Caso opte-se por rotacionar os elementos na árvore, como no caso das árvores AVL, é necessário preocupar-se em que não se pode, neste caso, trabalhar com os elementos individuais de forma eficiente, e sim com páginas inteiras. Pode ocorrer de as páginas precisarem ser divididas, e elementos movidos entre elas. Segundo Folk [10], a tendência para o rearranjo das páginas é que estas operações acabam se espalhando por uma grande parte da árvore.

Desta forma, a melhor forma para construir uma árvore consiste em escolher adequadamente os *separadores*, ou seja, os elementos que dividem o conjunto de chaves

de forma aproximadamente igual, e em conseqüência, como evitar que elementos agrupados (por exemplo, chaves contíguas) acabem por compartilhar a mesma página.

#### 2.4.4.4 - Árvores B

A solução encontrada por Bayer e McCreight [2] para estes problemas consiste exatamente em construir a árvore a partir de seus níveis mais baixos, ao invés do topo.

As árvores B consistem em árvores $\tau(k,h)$, em que ou a altura $h$ é zero (ou seja, a árvore está vazia), ou possui as seguintes propriedades:

1. Cada caminho da raiz até qualquer folha possui o mesmo tamanho $h$, onde $h$ é o número de nós no caminho;

2. cada nó, exceto o nó raiz, e as folhas possui pelo menos $k+1$ filhos. A raiz ou é um nó folha ou possui pelo menos dois filhos; e

3. cada nó possui no máximo $2k+1$ filhos.

Assim, cada nó possui até $2k$ elementos e $2k+1$ ponteiros para os nós filhos. Cada elemento consiste na tupla ($x_i$, $\alpha_i$, $p_i$), onde $x_i$ é a chave de classificação, $\alpha_i$ é uma informação adicional qualquer, associada à chave, e $p_i$ é um ponteiro para o nó filho. Nos nós folhas, o ponteiro $p_i$ não é utilizado. Como simplificação da notação, pode-se omitir o termo $\alpha_i$, sendo o elemento do índice representado na forma ($x_i$, $p_i$). A Figura 2.10 mostra um exemplo de organização de dados na página.

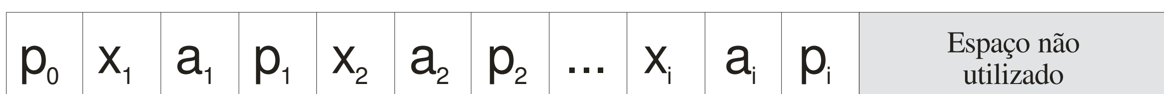



Figura 2.10. Exemplo de organização lógica de dados na página

Como o número mínimo e máximo de elementos por página é definido, então estes devem possuir tamanho fixo. Eles também são armazenados de forma classificada no interior dos blocos, o que permite o uso de algoritmos rápidos para memória principal na inserção e exclusão dos elementos no interior do nó. Assim, as seguintes características também devem ser válidas para que uma árvore seja considerada uma árvore B, considerando $P(p_i)$

a página para a qual $p_i$ aponta, $K(p_i)$ o conjunto de páginas na sub-árvore máxima em que $P(p_i)$ é a página raiz, e $l$ é o número de elementos no nó:

$$(\forall y \in K(p_0)), (y < x_1), \ (1)$$

$$(\forall y \in K(p_i)), (x_i < y < x_{i+1}); i = 1,2,....,l-1 \ (2)$$

$$(\forall y \in K(p_l)), (x_l < y). \ (3)$$

A Figura 2.11 mostra um exemplo de árvore B, em que, para a melhor visualização, estão presentes apenas os dados e os ponteiros para os nós filhos.

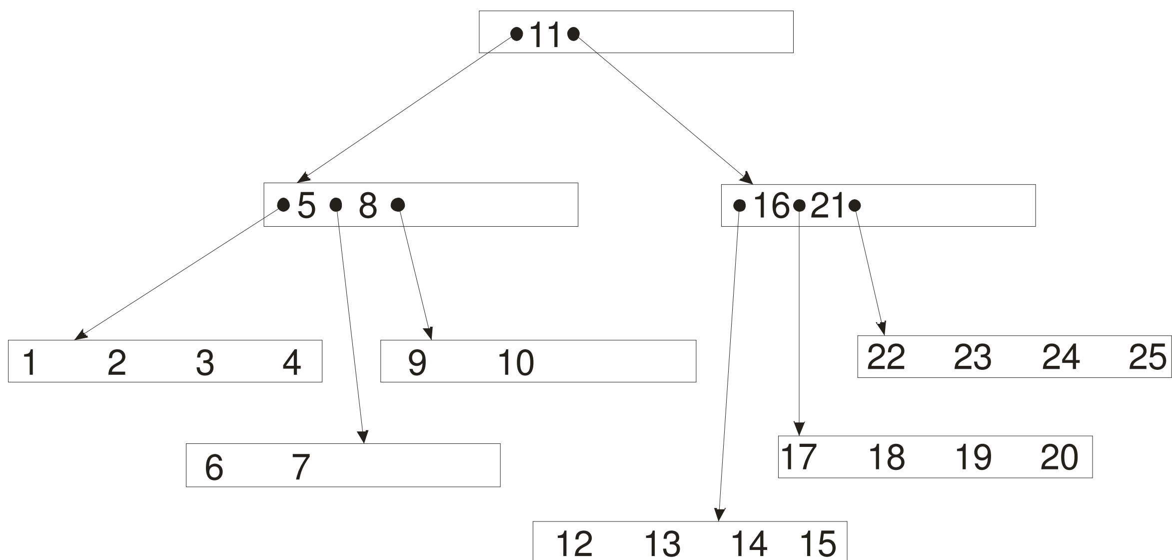



Figura 2.11. Estrutura de um índice em árvore B, adaptado de Bayer [2]

A garantia de que a árvore estará sempre balanceada consiste em fazer com que o elemento raiz apareça a partir de operações de divisão e promoção dos separadores, e esta é uma operação que ocorre de baixo para cima. Quando um registro é excluído, caso a página esteja com $k$ ou menos elementos, podem ser feitas também outras operações de ajuste, como a redistribuição de registros entre nós vizinhos e/ou a concatenação de nós.

2.4.4.5 - Algoritmo de pesquisa e recuperação

O algoritmo de pesquisa consiste em um procedimento com duas características marcantes:

1. Ele é recursivo; e

2. Trabalha em dois estágios, operando alternativamente em páginas inteiras e no interior das páginas.

O algoritmo, apresentado em sua versão iterativa, carrega uma página, iniciando a partir da raiz, e procurando a chave no interior da página, buscando cada vez em níveis mais baixos até que a chave seja encontrada ou que não possa chegar a níveis mais baixos, tendo encontrado um nó folha. Outras implementações, como a apresentada em Folk [10], utilizam uma abordagem recursiva.

Neste algoritmo, caso a chave de pesquisa seja menor que a primeira chave da página sendo examinada, ele prossegue carregando a página apontada por $p_0$, conforme (1). Em seguida, pesquisa dentro da página, utilizando pesquisa binária, para saber se o elemento em questão está nesta página. Caso esteja, retorna um ponteiro para o elemento. Se não estiver, busca qual a posição, dentre as possíveis, de acordo com a (2), e continua seguindo pela sub-árvore que satisfaz esta condição. Caso a chave esteja após o elemento $x_l$, a busca prosseguirá pela direita, seguindo o ponteiro $p_l$. Se o ponteiro a seguir seguido, em qualquer um dos casos, não apontar para uma página válida, é uma indicação que o nó atual é um nó folha, e a pesquisa será interrompida, retornando um ponteiro *null*.

#### 2.4.4.6 - Inserção, Divisão e Promoção

As operações de inserção são sempre feitas nos nós folha. Isso implica que, antes da inserção propriamente dita, deve haver uma fase de pesquisa até a folha onde o registro deve ser inserido.

Após a carga do nó folha onde o elemento de índice será inserido, é necessário verificar se há espaço disponível na página. Caso haja, o elemento colocado na posição correta, deslocando os registros, se necessário, e o processo termina neste ponto.

No caso de não haver espaço na página, ou seja, ela já possui *2k* elementos, é criado uma outra página irmã, e os registros são divididos de forma igual entre as duas páginas. O passo seguinte consiste em inserir o novo elemento na página correta, que pode ser a página original *P* ou a página irmã *P'*. Desta forma, garante-se que cada uma das páginas tem pelo menos *k* elementos.

O passo seguinte consiste na escolha do elemento para promoção, que, no caso, será o primeiro elemento da página *P'*. Retira-se este elemento da página *P'*, e o ponteiro $p_0$ da página *P'* é ajustado para apontar para o bloco anteriormente apontado por $p_1$ (o ponteiro correspondente ao elemento retirado de *P'*). Assim, será inserido na página pai o elemento promovido, e o ponteiro associado a este elemento é a nova página *P'*, retornando ao início do algoritmo.

#### 2.4.4.7 - Exclusão, Redistribuição e Concatenação

Quando um elemento é excluído de um nó, pode ocorrer de o mesmo ficar com uma quantidade de registros menor que o mínimo *k*. Neste caso, pode-se tomar uma das ações corretivas, a redistribuição ou a concatenação.

O processo para a exclusão de um elemento é um pouco mais complexo que o processo de inserção, para o caso de o elemento não estar em uma página folha. Caso esteja em uma folha, basta retirá-lo da página e efetuar a redistribuição ou concatenação, de acordo com o caso.

Em um registro encontrado fora de uma página folha, para excluí-lo, é necessária a substituição do elemento excluído pelo elemento imediatamente seguinte. A obtenção do elemento imediatamente seguinte é feita recuperando o primeiro elemento da folha mais à esquerda da sub-árvore $K_i$ correspondente ao elemento $x_i$ excluído. Em seguida, elimina-se este elemento da folha encontrada, e prossegue-se o processo de redistribuição ou concatenação, se for necessário.

A concatenação consiste no caso de duas páginas *P* e *P'* que sejam irmãos adjacentes, ou seja, possuam o mesmo pai *Q* e são apontadas por ponteiros adjacentes na página *Q*, que, juntas, possuírem menos que *2k* elementos serem transformadas em uma única página. Neste processo, na página *Q*, o elemento ($x_i$', $p_i$') que aponta para a página *P'* é movido para a página *P*, e o ponteiro correspondente deste elemento irá apontar para o $p_0$ da página *P'*. Ainda como conseqüência da exclusão do elemento $x_i$ da página Q, pode ser necessário efetuar outra concatenação ou redistribuição. A título de ilustração, a Figura 2.12 abaixo mostra a situação das páginas e dos elementos antes e depois da concatenação.

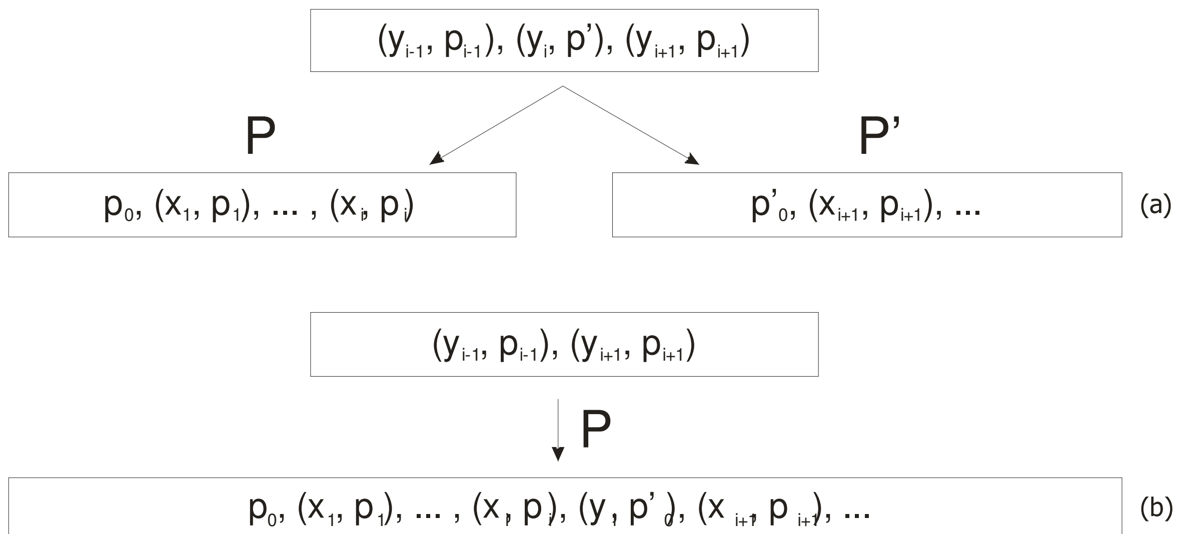



Figura 2.12. Concatenação de dois blocos. Em (a) é mostrada a situação antes da concatenação, e, em (b), após a operação.

A redistribuição ocorre quando o número de chaves em *P* e *P'*, que são irmãos adjacentes, for maior ou igual a *2k* elementos. Neste caso, pode-se distribuir igualmente os elementos entre as páginas *P* e *P'*, alterando também o elemento correspondente na página pai, *Q*. É importante notar que a redistribuição não se propaga para cima, como no caso da concatenação, tendo efeito apenas nas páginas *P*, *P'* e *Q*.

#### 2.4.4.8 - Registros de tamanho variável

A técnica apresentada anteriormente possui uma grande deficiência – trabalha apenas com registros de tamanho fixo. Entretanto, os dados normalmente possuem como característica possuírem um tamanho variável, seja por questões de economia de espaço em armazenamento secundário, ou por características próprias dos dados, com nos casos em que há campos multivalorados.

Sobre este problema, a estratégia apresentada por McCreight [39] consiste em fazer o controle não pelo número de elementos em cada página, mas sim pelo número de bytes ocupados pelos registros. Assim, pode-se fazer a divisão quando, ao inserir um novo registro, o mesmo não couber na página, sendo necessário dividi-la, tentando manter ao máximo possível o mesmo espaço possível em cada página.

Esta abordagem precisa ser cuidadosamente implementada, principalmente em relação ao tamanho máximo de cada registro, de forma a evitar que, durante a inserção de um novo, a

divisão torne-se impossível. Assim, é necessário que o tamanho máximo do registro seja, no máximo, a metade do espaço disponível em uma página.

### 2.4.5 - Arquivos seqüenciais indexados

Como citado anteriormente, é possível termos arquivos seqüenciais, cuja seqüência é dada por uma estrutura de índice externa. Este método de acesso permite que se escolha entre duas visões alternativas de um arquivo:

1. Uma visão *indexada* permite ao SGBD a busca direta por uma chave específica; e

2. Uma visão *seqüencial*, em que o SGBD pode gerenciar o arquivo sequencialmente, retornando os registros em ordem de chave.

Métodos de acesso seqüenciais indexados permitem, em uma única estrutura de acesso, fornecer os serviços necessários para uma busca eficiente através da chave de classificação, que é o padrão de acesso normalmente encontrado em sistemas interativos; e também facilitam o processamento em lote, através da visão seqüencial do arquivo. Além disso, também oferecem maior facilidade durante as operações de inserção e exclusão, já que o índice torna desnecessária qualquer operação de movimentação de grande parte do arquivo de dados, como ocorre em organizações de arquivo puramente seqüenciais.

#### 2.4.5.1 - Mantendo a seqüência de blocos de forma ordenada

O primeiro passo para a definição de uma estrutura de acesso que permita o acesso seqüencial aos registros consiste na formulação de estruturas de dados que garantam que a seqüência de blocos que compõe o arquivo de dados.

Como os blocos não necessariamente serão alocados de forma contígua, a não ser no caso de *clusters* de blocos – os quais podem não ser contíguos –, pode-se colocar campos de *link*, que ligam o bloco predecessor ao bloco sucessor.

Com essa estrutura também é possível inserir novos blocos no interior da seqüência, sem ser necessário reescrever todos os blocos subseqüentes. Isso ocorre quando, durante a inserção de registros em um bloco, ocorre um *overflow*, e o bloco precisa ser dividido ao meio. O novo bloco pode ser acrescentado ao final do arquivo físico, e os ponteiros de *link* ajustados de forma a inseri-lo no meio da seqüência, como uma lista ligada.

Durante o processo de exclusão também pode ocorrer de o bloco ficar vazio, ou com uma quantidade de elementos menor que um determinado mínimo estipulado. Neste caso, ocorre um *underflow*. O *underflow* pode levar a duas condições:

1. Caso seja possível juntar a página com *underflow* com uma de suas vizinhas, deixando apenas um nó, pode-se concatenar os dois ou mais, liberando-os para ser reutilizados; ou

2. Caso não seja possível juntar com as vizinhas, pode-se pelo menos redistribuir os elementos entre as páginas, levando a uma distribuição mais ou menos igual.

O ajuste dos ponteiros no caso de *underflow*, quando uma das páginas é eliminada, é feito da mesma forma que no *overflow*, ajustando o ponteiro da página predecessora. É preciso também manter uma estrutura de dados para controle dos blocos livres, a qual é utilizada tanto durante o processo de inserção de novos blocos, como o de exclusão de blocos.

#### 2.4.5.2 - Acrescentando um índice à seqüência de blocos

Uma vez que se tem o arquivo organizado em forma de *heap*, mas com ponteiros apontando a seqüência correta de blocos a seguir, de forma a manter o arquivo ordenado, é necessário acrescentar uma estrutura para indexá-los.

Esta estrutura possui grande importância nas operações de busca por chave (e, consequentemente, de inserção e exclusão de registros, caso a busca seja feita pela chave de ordenação), já que reduz enormemente a quantidade de E/S necessárias para encontrar um bloco específico. Sem uma estrutura como esta é necessário ler em média a metade de todos os blocos do arquivo para encontrar o registro especificado pela chave.

Mantendo os registros no interior dos blocos de forma ordenada é possível utilizar-se de um índice esparso, conforme definido anteriormente. As entradas no índice teriam apenas a primeira chave existente em cada bloco, associadas a um ponteiro para o bloco.

#### 2.4.5.3 - As árvores B+

Nas seções anteriores foi apresentada a árvore B. Uma variante bastante utilizada é a árvore B+, que difere das árvores B por possuir os dados associados aos registros apenas nos nós folha. Os nós internos contém apenas ponteiros ou para outros nós internos ou para

os nós folha. Nestes nós também não são mantidas as chaves propriamente ditas, e sim uma cópia das mesmas. Quando ocorre uma exclusão em uma folha, cuja chave é igual à existente nos níveis internos, não necessariamente será necessário propagar esta exclusão para os níveis superiores. O mesmo raciocínio também é válido para as operações de inserção.

Esta organização de arquivo permite que se mantenham nos nós folha uma lista classificada de registros, permitindo o acesso seqüencial, e, ao mesmo tempo, provê os serviços necessários de indexação para uma eficiente busca por chave.

A operação de inclusão de novos registros ocorre da mesma forma que nas árvores B, caminhando da raiz até o nó folha em que o registro deva ser inserido, mas a diferença ocorre quando for necessário dividir o bloco. Enquanto na árvore B o primeiro elemento do novo bloco irmão criado é promovido, na variante B+ é promovido apenas uma cópia da chave do primeiro elemento. Assim, o separador que será promovido pode não ser necessariamente o mesmo da chave do primeiro elemento da página folha, após operações de exclusão e atualização. Durante uma operação de exclusão, os separadores nas páginas internas também não são alterados, a não ser que um caso de *underflow* seja detectado nos nós folhas. Neste último caso, pode ocorrer um rearranjo (redistribuição) dos registros na folha, e a conseqüente alteração nos separadores, ou a exclusão de um dos separadores devido à concatenação de páginas no nível folha. Caso também ocorra *underflow* nos nós internos, esta condição também será propagada até os níveis superiores.

Do ponto de vista do desempenho, há vantagens em relação às árvores B, uma vez que, como apenas as chaves estarão nos níveis internos, podem ser colocadas mais chaves por nó, reduzindo, portanto, a altura da árvore. A única desvantagem consiste em ser necessário sempre caminhar até o nó folha para obter o valor real da chave e do registro propriamente dito.

Esta é uma das estruturas de arquivo mais comumente utilizadas em gerenciadores de banco de dados, seja como mecanismo principal de armazenamento, seja como estrutura para um índice auxiliar.

## 2.5 - BANCOS DE DADOS

Segundo Hurson [38], sistemas de gerenciamento de bancos de dados foram largamente usados no ambiente de processamento de dados desde sua introdução no fim da década de 1960. O que permitiu o sucesso dos sistemas gerenciadores de bancos de dados foi sua capacidade de persistência, compartilhamento, independência de dados, consistência e integridade. Estas características não são encontradas em sistemas de arquivo tradicionais.

Para fornecer um arcabouço formal para a representação e manipulação de dados, para o desenvolvimento de um sistema de banco de dados é necessário um modelo de dados, conforme Chaudhri [3]. Antes, os modelos largamente utilizados eram os hierárquicos, em rede e os relacionais. Todos estes possuem o foco nos registros, entretanto as diferenças estão primeiramente em como eles organizam e relacionam os registros.

As primeiras aplicações para banco de dados foram feitas para negócios e administração, normalmente para serviços bancários manterem informações sobre correntistas e contas, e também para sistemas orientados a registro, como um controle de estoque.

### 2.5.1 - Histórico

Anteriormente aos bancos de objetos, diversos sistemas foram propostos e utilizados comercialmente. Eles serão detalhados em seguida.

#### 2.5.1.1 - Bancos Hierárquicos

Conforme Date [6], os bancos de dados hierárquicos compõem-se de um conjunto ordenado de ocorrências múltiplas de um tipo de árvore.

O tipo de árvore é composto de um registro *raiz* e um conjunto ordenado de zero ou mais tipos de sub-árvores dependentes. Um tipo de sub-árvore também é um tipo de registro: compõe-se de um registro raiz – a raiz do tipo sub-árvore – e um ou mais tipos de sub-árvore dependentes de nível inferior. Desta forma, todo tipo de árvore compõe-se de um conjunto hierárquico dos tipos de registro. É fato também que os registros são compostos de campos. Na Figura 2.13 é mostrado um exemplo de um modelo hierárquico. O relacionamento entre os diferentes tipos de registro, em um banco de dados hierárquico, se faz através de relacionamentos pai/filho.

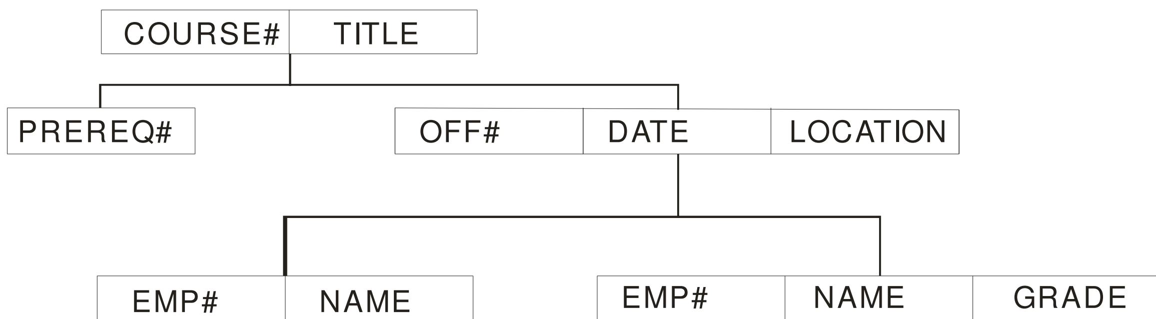



Figura 2.13. Exemplo de modelo hierárquico, adaptado de Date [6]

Às ocorrências também se aplicam esta terminologia raiz/pai/filho. Assim, para cada ocorrência de um registro pai, ocorre também zero ou mais ocorrências de uma sub-árvore característica.

2.5.1.2 - Bancos em rede

Ainda segundo Date [6], os bancos em rede são uma evolução dos bancos hierárquicos. Enquanto que nos bancos hierárquicos, um filho pode ter apenas um pai, nos bancos com estrutura de dados em rede um registro-filho pode ter qualquer número de pais.

Nesta estrutura, um banco de dados compõe-se de dois subconjuntos: um conjunto de ligações (um conjunto de ocorrências múltiplas de cada um dos tipos de registro), e um conjunto de ocorrências múltiplas de cada um dos tipos de ligação. Em cada ligação existem dois registros, um registro *pai* e um registro *filho*. Cada ocorrência de uma ligação corresponde a uma única ocorrência do registro pai e um conjunto ordenado de zero ou mais ocorrências do registro-filho.

2.5.1.3 - Bancos Relacionais

Segundo Date [6], os bancos de dados relacionais são baseados nas teorias dos conjuntos e da álgebra relacional. O armazenamento dos dados se dá em tabelas – que podem ser vistas como conjuntos – e linhas (ou tuplas) nestas tabelas. As linhas podem ser encaradas como os elementos dos conjuntos.

As operações da álgebra relacional – união, intersecção, diferença, produto cartesiano -, além de projeção e seleção, são utilizadas para definir as consultas, e o resultado dessas consultas são também tabelas.

Nestes bancos de dados, normalmente é utilizada a linguagem SQL como ferramenta de consulta, sendo ela padronizada e implementada por grande parte dos sistemas de bancos de dados relacionais.

Os bancos relacionais possuem características que propiciam a não-redundância da informação, e a modelagem de dados através das formas normais definem as características sobre como esta redundância deve ser evitada.

## 2.6 - BANCOS DE OBJETOS

Na década de 1980, com o avanço da tecnologia, novos processos orientados a dados surgiram, onde os sistemas tradicionais de banco de dados relacionais eram inadequados. Como exemplos, podemos citar sistemas de apoio ao projeto e manufatura (*CAD/CAM*, *CIM*), bancos de dados médicos e científicos, sistemas de informações geográficas e bancos de dados multimeios.

Bancos de dados orientados a objeto (*object-oriented database management systems, OODBMS*) foram desenvolvidos para suportar os requerimentos destas aplicações. A tecnologia orientada a objetos provê a flexibilidade necessária, pois não é limitada a tipos de dados pré-definidos. Uma das mais importantes características de bancos de objetos é a possibilidade de especificar tanto a estrutura de dados quanto as operações que irão operar sobre estas estruturas de dados.

O princípio básico do paradigma de orientação a objetos na programação consiste em desenvolver os sistemas como uma coleção de objetos, os quais podem ser agrupados em classes, que se comunicam via passagem de mensagens.

Em um sistema clássico orientado a objetos, os objetos duram apenas enquanto o programa que os criaram está em execução. Por outro lado, em um banco de objetos, os objetos podem ser ou **persistentes** ou **transientes**. Os objetos persistentes são armazenados normalmente em armazenamento secundário, e podem durar mais do que o programa que os criou, e, ainda, serem compartilhados por diversos programas. Já os objetos transientes possuem a duração apenas do programa que os criou. Para um programa desenvolvido em linguagem orientada a objetos possuir as características de persistência, é necessário que seja necessário o acréscimo de características de um SGBD na linguagem, como indexação, controle de concorrência e gerenciamento de transações.

### 2.6.1 - Perspectiva Histórica

Alguns trabalhos experimentais e o desenvolvimento de vários protótipos marcaram o desenvolvimento de bancos orientados a objetos. Apenas após isto os fundamentos teóricos foram investigados e padrões foram desenvolvidos. Os primeiros OODBMSs surgiram no mercado por volta de 1986, e desde então tanto os sistemas como a ciência da computação e os cenários de desenvolvimento se modificaram.

#### 2.6.1.1 - Os primeiros sistemas

O fim da década de 1980 assistiu o nascimento dos primeiros OODBMSs, e, desde então, este tipo de sistema passou por intenso desenvolvimento na indústria de *software*. O que caracterizou esta fase foi a grande quantidade de sistemas produzidos por fabricantes pequenos. Eram revolucionários, no sentido em que não partilhavam nenhuma base de código e nem modelos de dados com os DBMSs existentes na época.

A primeira geração surgiu no ano de 1986, pela companhia francesa Graphael, com o produto G-Base. Em 1987, a companhia americana Servio Corporation, lançou o GemStone. Em 1988, surgiram o Vbase, pela Ontologic (que depois se tornou Ontos, Inc.), e o Statice, pela Symbolics. O interesse particular deste grupo era oferecer persistência às linguagems orientadas a objeto, particularmente aquelas aplicadas ao ramo da Inteligência Artificial. Eram sistemas *stand-alone*, baseados em linguagens proprietárias, e não utilizavam plataformas de padrão industrial. Estes sistemas podiam ser consideradas apenas extensões às linguagens orientadas a objetos, acrescentando características como o gerenciamento de transações.

Em 1988, a companhia Ontos lançou o software de mesmo nome, marcando o início da segunda geração de OODBMSs. Nesta geração estão sistemas como o ObjectStone, Objectivity/DB e Versant ODBMS. O que os diferencia da primeira geração de sistemas é que estes já são voltados para plataformas padrão e arquitetura cliente/servidor. Estes sistemas executavam eram baseados em C++, X-Windows e estações Unix.

O primeiro produto da terceira geração já estava disponível logo em seguida, em agosto de 1990, o Itasca. Em seguida estavam disponíveis o O2 e o Zeitgeist. Estes produtos podem ser definidos como bancos de dados com características avançadas, como o gerenciamento de versões, e linguagens de definição e manipulação de dados orientadas a objetos e

computacionalmente completas. Portanto, eram sistemas bastante avançados, tanto do ponto de vista da tecnologia de banco de dados como também do ambiente de desenvolvimento de *software*.

Como não havia um padrão no desenvolvimento dos bancos orientados a objetos, a comunidade de pesquisa percebeu que havia a necessidade de definir pelo menos o que era um sistema gerenciador de banco de dados orientado a objetos. Assim, foi publicado o Manifesto dos Bancos de Dados Orientados a Objeto [1], que descreve as características básicas que um OODBMS deve ter.

### 2.6.2 - Características de um banco de objetos

Durante o surgimento dos primeiros bancos de dados orientados a objetos, tentou-se definir as características básicas necessárias para os mesmos. Uma das propostas é o Manifesto dos Bancos de Dados Orientados a Objeto [1], que separa as características em três grandes grupos:

- *Características obrigatórias*. São aquelas que devem estar presentes no sistema para que ele possa ser chamado de OODBMS. São estas: objetos complexos, identidade de objetos, encapsulamento, tipos ou classes, herança, polimorfismo ou sobrecarga de funções e ligação tardia (*late binding*), extensibilidade, ser computacionalmente completo, persistência, gerenciamento de armazenamento secundário, concorrência, recuperação em caso de falhas e possibilidade de consultas *ad hoc*;

- *Características opcionais*. São as que podem ser acrescidas para deixar o sistema melhor, mas não são obrigatórias. Dentre elas estão a herança múltipla, checagem de tipo e inferência, distribuição, projeto de transações e versões;

- *Características abertas*. Estas incluem as em que o projetista pode fazer uma escolha. Incluem o paradigma de programação, o sistema de representação, o sistema de tipos e a uniformidade.

### 2.6.3 - Padronização

No início da década de 1990, a característica marcante dos OODBMS era a falta de um modelo comum de dados. Além disso, uma das razões pelas quais os usuários de banco de dados não efetuavam migrações para os novos bancos de dados era o medo de uma nova tecnologia, além de um sentimento dentro da comunidade OODBMS de que o mercado não migraria para esta tecnologia no estado em que estava, sem uma padronização definida. Isso ainda agravado pelo escopo dos bancos de objetos ser maior do que os bancos de dados relacionais (RDBMS), já que eles integram o banco de dados e a linguagem de programação.

Assim, o consórcio ODMG (*Object Database Management Group*) foi formado em 1991, como uma resposta à falta de um grupo de padronização trabalhando na definição de padrões para bancos de objetos. Esta padronização fez também parte de um esforço para difundir os padrões para o mundo de orientação a objeto.

Uma característica importante da padronização dos bancos de objetos é que, como os mesmos também são integrados a uma linguagem de programação orientada a objetos, a natureza da padronização é diferente da padronização dos bancos de dados relacionais. Nos bancos de objetos é necessário também padronizar como é feita a interface e integração entre os bancos de dados e as linguagens de programação.

O padrão ODMG define um modelo de objetos de referência (*ODMG Object Model*), uma linguagem de definição de objetos (*ODMG Object Definition Language – ODL*) e uma linguagem de consulta a objetos (*OQL*). Além disso, define também as ligações com as linguagens de programação, tanto para a definição dos objetos como para a manipulação de dados.

### 2.6.4 - Recuperação de dados em bancos de objetos

Segundo Cook [5], um dos problemas da implementação de bancos de objetos consiste na falta de uma definição de formas de consulta de dados. Boa parte das formas padronizadas envolvem a utilização de consultas baseadas em uma linguagem textual, armazenada em uma cadeia de caracteres, perdendo, assim, as vantagens oferecidas pelas novas ferramentas de desenvolvimento (IDE – *Integrated Development Environment*), como a refatoração de código. Além disso, o projeto dos bancos de objetos prevê que o acesso aos

dados se dá através da navegação dos relacionamentos entre os dados, a partir de um ou mais objetos raiz.

Apesar desta característica de projeto, parte dos bancos de objeto ainda permite o acesso aos dados através de alguma forma de consultas *ad-hoc* e criação de índices sobre os atributos, fornecendo uma visão quase relacional dos dados, chegando, inclusive, a oferecer interface SQL ao banco de dados.

## 2.7 - BANCO DE DADOS OBJETO-RELACIONAL

Os bancos de dados objeto-relacionais (ORDBMSs) provêm uma alternativa para usar objetos em sistemas de banco de dados. Enquanto que os OODBMS representaram um modelo revolucionário, os bancos objeto-relacionais apresentam um contexto evolucionário, integrando os objetos nos bancos relacionais.

A idéia de estender os bancos de dados relacionais para atribuir-lhes algumas das características dos OODBMS data de 1990, quando foi publicado o Manifesto dos Sistemas de Bancos de Dados de Terceira Geração [43]. Segundo esta proposição, características como regras e estruturas complexas de objetos, devem também fazer parte dos bancos de dados de terceira geração, e os mesmos devem também ser compatíveis com os bancos de dados relacionais – os de segunda geração. Além disso, os mesmos devem ser abertos, ou seja, permitir o acesso a partir de uma variedade de aplicativos e linguagens de programação, e serem capazes de cooperar outros bancos de dados de terceira geração em ambientes distribuídos.

Com este foco, propõe-se que os sistemas de bancos de dados de terceira geração implementem características como tipos complexos, herança, encapsulamento, polimorfismo e geração automática de identificadores únicos para os registros. Assim, vê-se uma convergência na conceituação de ORDBMS e OODBMS, em que os primeiros englobam as funcionalidades dos bancos de objetos, e ainda oferecendo as facilidades dos bancos de dados relacionais.

Na década de 1990 houve grande experimentação com bancos de dados objeto-relacionais, e os grandes fabricantes (Oracle, IBM, Sybase) iniciaram a implementação de características objeto-relacionais em seus produtos. Também neste período, surgiu a

padronização SQL-3, que foi implementada pelos fabricantes, refletindo o consenso a respeito destas características.

Também nesta época, as facilidades de consultas declarativas dos OODBMSs e as capacidades de objeto dos ORDBMSs melhoraram, facilitando uma convergência entre estas tecnologias. Desta forma, tínhamos os dois extremos, em que os OODBMSs suportavam dados complexos com consultas simples, e os RDBMSs, com suporte a dados simples com consultas complexas, e os ORDBMSs vieram a suprir o nicho de dados complexos com consultas complexas.

Entretanto, ainda resta uma diferença fundamental entre os OODBMSs e os ORDBMSs: enquanto que os primeiros podem ser vistos como uma extensão das linguagens de programação orientadas a objeto, fornecendo de forma transparente os mecanismos de persistência, os segundos precisam de uma API específica, baseada em SQL, para recuperação e armazenamento de dados persistentes.

## 2.8 - MODELO DE DADOS ORIENTADO A OBJETO

Conforme Silberschatz [42], o modelo de dados orientado a objeto é baseado no paradigma da orientação a objetos. Este paradigma nos mostra as características fundamentais de um modelo de objetos, que consiste na presença de um sistema de tipos (classes), objetos, herança e polimorfismo.

### 2.8.1 - Identidade de Objeto

Em um sistema baseado em objetos, cada objeto tem sua identidade única, e a mesma será mantida mesmo que suas algumas de suas propriedades sejam alteradas com o tempo. Há diversos tipos de identidade, e o conceito utilizado em orientação a objetos é mais forte do que o encontrado em muitas linguagens de programação não orientadas a objeto. Existem os seguintes modelos:

1. *Identidade por valor*. Um valor de dados é utilizado para a identidade. Normalmente este tipo de identidade é utilizado em sistemas relacionais;

2. *Identidade por nome*. Neste caso, um nome fornecido pelo usuário é utilizado para identificar unicamente um conjunto de dados. Normalmente, este tipo de identidade

é utilizado em sistemas de arquivo, em que um nome identifica um e apenas um arquivo, independente de seu conteúdo; e

3. *Embutido*. A noção de identidade está embutida no modelo de dados ou linguagem de programação, e não é necessário que o programador forneça um identificador para os dados. Este é o tipo de identidade utilizado em sistemas orientado a objetos, em que o sistema atribui automaticamente a cada objeto um identificador único.

### 2.8.2 - Composição de Objetos

Os objetos podem também ser modelados por referências a outros objetos, representando, assim, conceitos do mundo real. Os objetos que contém referências para outros objetos são chamados *objetos complexos* ou *compostos*. Esta composição cria uma hierarquia de composição entre os objetos, que formam um grafo, em que os nós correspondem aos objetos e os arcos, às referências.

### 2.8.3 - Persistência de Objetos

Quando os objetos são criados, em uma linguagem de programação orientada a objetos, os mesmos normalmente são *transientes*, ou seja, existem apenas enquanto o programa que os criou está em execução. Caso seja necessário armazenar um objeto, é necessário inicialmente fornecer um mecanismo que permita que o mesmo seja armazenado em meio durável, tornando-os persistentes. Algumas abordagens para oferecer este serviço são:

1. *Persistência por classe*. Esta abordagem consiste em declarar que uma determinada classe é persistente, e, portanto, todas as instâncias desta classe serão armazenadas de forma persistente. A principal desvantagem dessa abordagem é que ela não é flexível, em que os objetos de outras classes não serão armazenados, e todas as instâncias das classes declaradas como persistentes serão armazenadas;

2. *Persistência por criação.* Neste método, utiliza-se uma sintaxe modificada na linguagem de programação, em que a criação de objetos persistentes utiliza uma extensão na forma da criação de objetos transientes. Assim, um objeto torna-se persistente ou transiente de acordo com a forma com a qual ele foi criado;

3. *Persistência por marcação*. É uma variante da abordagem da persistência por criação. Nesta abordagem, todos os objetos são criados como transientes, mas podem-se marcar alguns objetos como persistentes após terem sido criados. O sistema gerenciador orientado a objetos se encarregará de fazer a persistência destes objetos marcados, permitindo aos mesmos serem recuperados em sessões posteriores; e

4. *Persistência por referência*. Neste método, a partir de um objeto raiz persistente, todos os outros objetos por ele referenciado serão também persistidos. Assim, todos os objetos referidos direta ou indiretamente por um objeto persistente serão também persistentes.

### 2.8.4 - Linguagem Java e Persistência

Existem várias tentativas de oferecer serviços de persistência à linguagem Java. O mais simples deles consiste em utilizar o mecanismo padrão de serialização da linguagem, permitindo que grafos de objetos sejam armazenados e lidos como um *stream* de bytes, podendo ser gravados em arquivos. Com estes arquivos, é possível reconstruir em seguida o estado dos objetos que foram serializados. Entretanto, esta ferramenta permite apenas o armazenamento de pequenas quantidades de objetos, e não oferece os serviços quem um gerenciador de banco de dados oferece.

Outras soluções são baseadas na utilização de bancos de dados relacionais para o armazenamento de dados, através das classes da API *Java Database Connectivity* (*JDBC*). Com esta técnica, as classes precisam ser mapeados para relações no banco relacional, e tal mapeamento pode ser feito de forma automática, através de ferramentas já desenvolvidas para este fim, ou através de classes ou métodos desenvolvidos pelo projetista especialmente para este fim. Em qualquer das alternativas, o trabalho é maior do que a utilização de ferramentas de persistência transparente.

Com as características da versão 5 do Java (Java 5), a construção dos mapeamentos se tornou mais fácil, uma vez que as informações referentes ao banco relacional podem ser colocadas no próprio código da classe, através de anotações. O padrão EJB3 define algumas anotações necessárias para este mapeamento, e as ferramentas de mapeamento O/R também podem definir extensões para estas classes. Um exemplo de ferramenta de

mapeamento objeto/relacional bastante difundido é o Hibernate [29], cujo modelo de programação e mapeamento serviu como base para a definição do EJB3/Entity Beans, e também possui uma versão para a plataforma .Net [32]. Outro padrão de mapeamento bastante difundido é o JDO [23], implementado por ferramentas como o Apache OJB [13], SolarMetric Kodo [36], dentre outros.

Outra abordagem para persistência em Java é o armazenamento direto em um OODBMS. Como o impacto da linguagem Java foi bastante considerável, diversas implementações de bancos de objetos já oferecem *bindings* para a linguagem Java, como o POET, o ObjectStore e o Caché.

Além disso, o padrão ODMG 2.0, liberado em 1997, também fornece o padrão de ligação para a linguagem Java, além das linguagens C++ e Smalltalk, definidas na primeira versão. Na versão 3.0 do ODMG, as ligações com a linguagem Java foram ainda mais melhoradas.

### 2.8.5 - Armazenamento de Objetos

Pode-se utilizar quaisquer uma das técnicas descritas anteriormente (arquivos seqüenciais, arquivos *heap*, estruturas de *hashing*) para o armazenamento de objetos. Entretanto, estruturas de arquivo adicionais precisam ser utilizadas para suportar algumas das características específicas de dados orientados a objetos, como campos multivalorados, ponteiros persistentes e objetos realmente grandes (que não cabem em uma página do banco de dados).

Para o armazenamento de objetos, pode-se pensar na mesma forma de armazenamento utilizada para o armazenamento de tuplas de um banco relacional. Para os campos multivalorados, como apontado anteriormente, as alternativas são ou reservar um espaço dentro do registro para algumas repetições, e acrescentar um ponteiro para outra estrutura de armazenamento, ou armazenar apenas o ponteiro para outra estrutura de armazenamento.

Para os campos multivalorados, pode-se, em nível de armazenamento, criar uma outra relação, e, no registro original, acrescentar o ponteiro para a relação. Desta forma, o armazenamento dos dados constitui-se em uma normalização adicional dos dados do objeto. Esta outra relação pode ser armazenada como uma árvore ou *heap*.

Os campos grandes podem ser também armazenados da mesma forma que os campos multivalorados, mas, neste caso, é preciso que a forma de armazenamento suporte o deslocamento dentro do arquivo sem ser necessário ler todos os blocos existentes até o ponto em que se deseja iniciar a recuperação dos dados.

Também podem existir objetos (e não só campos) grandes o suficiente para não ser possível armazená-los em uma página (ou o limite de tamanho máximo para um registro, dependendo da organização de arquivos utilizada). Estes também podem ser armazenados como relações separadas, utilizando alguma estrutura que permita facilmente a recuperação dos campos – utilizando, por exemplo, algum identificador do campo ou seu nome como chave de um índice.

Cada objeto possui também um identificador único (OID). Os OIDs podem ser implementados de duas formas:

- *OID lógico*. Nesta implementação, o identificador de objeto não representa diretamente a localização do registro na estrutura de armazenamento, mas sim é a chave de algum índice que permita a localização rápida do objeto buscado; e

- *OID físico*. No OID já estão codificados explicitamente alguns campos para localização direta do registro no banco de dados, contendo dados como o número do volume, o número da página e o deslocamento dentro da página.

Para auxiliar ainda nos casos em que um objeto é excluído, os OIDs físicos costumam ter também algum campo com um identificador único do objeto, que não é repetido em nenhum outro objeto existente no banco de dados. Isso permite que facilmente sejam detectados ponteiros pendentes (*dangling pointers*). Essa detecção é feita quando, durante a leitura de um objeto, seu OID é diferente do qual foi utilizado para carregá-lo.

Também pode ocorrer a migração de um objeto de uma página para outra. Neste caso, pode ser deixado no local antigo um ponteiro de encaminhamento, apontando para a nova localização do objeto. Esta abordagem possui o inconveniente de ser necessário mais uma operação de E/S para buscar o registro desejado.

Os ponteiros persistentes são implementados em termos dos identificadores de objeto. Quando um objeto é serializado para um registro, as referências para outros objetos

persistentes são armazenadas como um campo contendo o OID do objeto referenciado. Desta forma, é garantido que os objetos persistentes não sejam duplicados quando da serialização de um grafo a partir de um objeto raiz.

Sendo os ponteiros persistentes armazenados em função do OID do objeto referenciado, alguma forma de tradução precisa ser feita para ponteiros de memória, e vice-versa, quando o objeto for lido e gravado. Segundo Silberschatz [42], técnicas de mistura de ponteiros, como as apresentadas anteriormente, são satisfatórias quando se está utilizando linguagens como o C++, em que referências para outros objetos são simplesmente ponteiros em memória.

Quando são utilizadas linguagens como o Java, não é possível fazer o acesso diretamente a endereços de memória de forma segura, devido a diferenças entre as implementações das várias máquinas virtuais existentes, e devido, também a operações que podem ocorrer em segundo plano, como compactação da memória *heap* e coleta de lixo. Nestas linguagens, quando é encontrado um OID, é necessário verificar se o objeto referenciado ainda está em memória e, caso estiver, substituir o OID pela referência a este objeto. Caso o mesmo ainda não tenha sido materializado, pode-se ou materializá-lo neste momento (*eager loading*) ou criar um objeto *proxy* para o mesmo, deixando a materialização para quando ocorrer o primeiro acesso ao objeto (*lazy loading*).

Muitas vezes não é possível e nem desejado que o objeto serializado seja uma cópia fiel da memória ocupada pelo mesmo, pois pode ser necessário que o banco de dados seja compartilhado em máquinas com diferentes arquiteturas, ou que os objetos serializados e materializados sejam utilizados por diferentes versões de compiladores, que geram códigos objeto com *layouts* de ocupação de memória diferentes em cada caso.

### 2.8.6 - Coleta de Lixo

Segundo Cook [4], a característica de os bancos de objeto suportarem a noção de um identificador de objeto independente do valor do mesmo e a definição de tipos complexos de dados trouxeram também um grande problema de performance nestes tipos de bancos de dados: a liberação de espaço de armazenamento na memória secundária dos objetos não mais acessíveis. Nos bancos de dados em rede e hierárquicos percebe-se também que a presença de dados inacessíveis possui grande impacto no desempenho de um SGBD.

Como resultado, os sistemas possuíam operações específicas para permitir ao administrador do banco de dados reduzir o volume de dados armazenados e melhorar a localidade de referências (*clustering*).

Já nos bancos de dados orientados a objetos, há duas formas básicas de excluir um objeto da área de armazenamento: através da **exclusão manual**, em que há uma operação explícita, chamada pela aplicação, para a exclusão do objeto; e a **exclusão automática**, em que o sistema gerenciador de banco de dados automaticamente descobre os objetos não acessíveis.

Para a implementação de exclusão automática há dois caminhos, um baseado em informações locais, como a contagem de referências de um objeto; e outro baseado em informações globais, em que a informação é obtida caminhando pelo grafo de objetos ativos (ou seja, acessíveis).

A exclusão manual dá para o programador o controle completo sobre como e quando um objeto será excluído, mas tem por efeito colateral permitir a presença de ponteiros pendentes, ou seja, apontando para registros inválidos. Também neste caso, em que haja alguma falha na aplicação, pode ocorrer de algum objeto não ser corretamente eliminado, deixando o espaço ocupado por ele inacessível. As técnicas baseadas em contagem de referências são bastante eficientes, contudo falham quando os dados envolvidos possuem alguma dependência cíclica.

Os algoritmos de coleta de lixo são apropriados para todos os casos, mas sua implementação, para sistemas gerenciadores de banco de dados, não é a mesma utilizada para a coleta de objetos inacessíveis em sistemas baseados em memória principal, devido às características adicionais existentes em um banco de objetos, como transações, compartilhamento, distribuição de dados, desempenho da estrutura de armazenamento, e também pelo volume de dados que devem ser tratados.

### 2.8.7 - Evolução de Esquema

Ainda segundo Young-Gook [41], a evolução do esquema é necessária não só por que os modelos de dados são menos estáveis do que o esperado, mas também porque as aplicações típicas de um banco de objetos (CAD/CAM, multimídia e aplicações

científicas) não são completamente compreendidas e ainda passam por freqüentes alterações no esquema.

Apesar de as alterações de esquema serem suportadas por boa parte dos bancos de objetos, estas alterações ainda causam problemas em relação aos bancos compartilhados por diversas aplicações, pois as alterações podem causar efeitos em diversos pontos. Os desenvolvedores, assim, devem consultar entre si para saber se uma determinada alteração de esquema envolverá efeitos colaterais em outros aplicativos que compartilham dos mesmos dados.

Uma forma de aliviar este problema é fornecer visões dos dados armazenados, de acordo com a versão em que os mesmos foram armazenados. Estas visões vão fornecer às aplicações legadas apenas os dados para os quais elas estão preparadas para lidar, e estas visões também fornecem serviços de atualização dos objetos de forma a manter a compatibilidade com as versões mais novas do esquema.

# 3 - JOODBMS – JAVA OBJECT ORIENTED DATABASE

O JOODBMS – *Java Object Oriented Database* – tem dois focos principais: ser utilizado como uma ferramenta de experimentação para pesquisa em otimizações de bancos de objetos, e também fornecer à linguagem Java características de persistência transparente.

A linguagem Java foi escolhida para esta implementação por ter uma grande quantidade de bibliotecas utilitárias já disponíveis, como a *Javolution*, utilizada para o acesso às estruturas de dados armazenadas em disco, e a *CGLIB*, para a geração de *proxies* em tempo de execução. Além disso, esta linguagem possui grande penetração no universo acadêmico, favorecendo seu uso como ferramenta de pesquisa.

Dentre os bancos de dados orientados a objetos com código fonte disponíveis, destacamos o Ozone [34] e o DB4O [26]. O primeiro possui também características de pesquisa, com foco no agrupamento de objetos, de forma a obter um melhor desempenho quando um determinado conjunto de objetos for recuperado do disco. Já o segundo é um banco de objetos de uso geral, disponível em Java e na plataforma .Net. Seu foco é a utilização embutida nas aplicações, apesar de possuir também recursos para acesso cliente/servidor.

Há outros bancos orientados a objetos e objeto-relacionais largamente utilizados, como o *Caché* [30], o *Oracle* [14], e o PostgreSQL [35]. Algumas características desses banco serão apresentadas a seguir.

Destes, o Caché foi desenvolvido com a finalidade de ser um banco de objetos completo, oferecendo uma linguagem orientada a objetos para o desenvolvimento das aplicações, uma IDE (*Integrated Development Environment*) para facilitar o desenvolvimento, depuração e adminsitração do banco de dados, e também um servidor *web* embutido. Oferece também ligações para a linguagem Java, em que a ferramenta cria objetos *proxies* para o acesso ao banco de dados a partir de uma aplicação Java. Entretanto, no servidor a linguagem para programação recomendada é proprietária, o Caché ObjectScript, ou, como alternativa, o Caché Basic. Possui estrutura de armazenamento baseada no MUMPS [17], em que as variáveis podem ser tratadas como um array ou *hash* multidimensional.

Já o *Oracle* e o *PostgreSQL* oferecem características objeto-relacional. O *Oracle* possui o conceito de tipos definidos pelo usuário (custom-types), os quais podem ter atributos e métodos. Como forma de possibilitar a navegação entre os tipos, há o conceito de

referência para objeto, em que é armazenado no banco de dados apenas um ponteiro para a tupla referenciada. Entretanto, no PostgreSQL, a forma de armazenamento ainda é em tabela *heap*, ou seja, com os registros desordenados, e não há suporte direto a uma coluna ser uma referência direta a outro objeto. Estes dois bancos de dados também não possuem como definir classes em Java diretamente no banco de dados – apesar de suportarem a utilização de Java como linguagem de programação para *stored procedures*.

A principal vantagem do JOODBMS em relação às alternativas apresentadas anteriormente é que o mesmo permite a persistência de forma quase totalmente transparente, à exceção dos serviços de coleta de lixo. E apenas uma linguagem de programação pode ser utilizada para o desenvolvimento de toda a aplicação, incluindo a modelagem de dados e a codificação da lógica de negócio. Outro fator que merece consideração é que os métodos associados às classes persistentes são executados no contexto do servidor de banco de dados. Na Tabela 3-2 há uma comparação resumida entre as características dos bancos de objetos disponíveis. Nesta tabela, as características em que o JOODBMS se sobressai em relação aos outros sistemas gerenciadores de bancos de dados estão marcadas em cinza.

O JOODBMS foi desenvolvido utilizando uma arquitetura semelhante à proposta por Garcia-Molina [12], mostrada na Figura 3.1. Nesta figura, os módulos destacados estão implementados e documentados. Na Tabela 3-1 há uma descrição sucinta dos módulos e de quais classes são utilizadas por cada um teles.

Tabela 3-1. Descrição sumária dos módulos implementados no JOODBMS

| Módulo | Classes | Descrição |
|---|---|---|
| Gerenciador de configuração | joodbms.impl.StorageManagerImpl | Responsável por armazenar as configurações do banco de dados em um arquivo de propriedades |
| Gerenciador de armazenamento | joodbms.impl.StorageManagerImpl<br>joodbms.core.PageIdentifier | Responsável por manter abertos os arquivos de dados e fornecer os métodos de leitura/escrita nos mesmos, através de *ByteBuffers* |
| Gerenciador de *cache* | joodbms.impl.CacheManagerImpl<br>joodbms.util.LRUList<br>joodbms.core.PageIdentifier<br>joodbms.core.PageBuffer | Fornece os serviços de *cache* dos dados obtidos e gravados a partir do gerenciador de armazenamento. O gerenciador de páginas utiliza o gerenciador de *cache* para recuperar páginas que estejam em memória |
| Gerenciador de páginas | Joodbms.impl.PageManager<br>joodbms.core.PageIdentifier<br>joodbms.core.PageBuffer | Gerencia a tradução de endereços lógicos (64 bits) para endereços físicos (pares identificador do arquivo/numero do bloco). Também é responsável por gerenciar a alocação de blocos no arquivo, através de uma estrutura de *bitmaps* |

| | | |
|---|---|---|
| Serializador de páginas | joodbms.impl.pages.PageSerializer<br>joodbms.impl.pages.<br>FreeSpaceBitmap<br>joodbms.core.DataPage e subclasses | Encapsula os objetos *PageBuffer* obtidos a partir do gerenciador de cache em objetos de classes descendentes de *joodbms.core.DataPage*, de acordo com o tipo da página. |
| Gerenciador de índices | joodbms.impl.tree.BPlusTree<br>IndexManagerImpl<br>joodbms.impl.tree.<br>BTreeConfigurationPage<br>joodbms.impl.tree.<br>BTreePage<br>joodbms.impl.tree.<br>BTreeInnerPage<br>joodbms.impl.tree.<br>BTreeLeafPage | Gerencia o armazenamento em uma estrutura de árvore paginada. O método de acesso é uma árvore B+, a qual mantém os dados das tuplas apenas nas páginas folha. Utiliza o serializador de páginas para recuperar as páginas que contêm os dados dos índices, e, indiretamente, o gerenciador de *cache* em relação aos serviços de bloqueio de páginas em memória. |
| Serializador de objetos | joodbms.impl.serializer.<br>ObjectSerializer<br>joodbms.core.<br>JoodbmsPersistentProxy | Serializa e materializa os objetos no banco de dados. Utiliza um esquema próprio de serialização, uma vez que o serializador do Java suporta apenas a serialização e materialização de todo um grafo de objetos. As referências para outros objetos e os novos objetos são retornados como *proxies*, que serão carregados ou gravados apenas quando necessário. |
| Fábrica de serializadores | joodbms.impl.serializer.<br>ObjectSerializerImpl<br>Classes que implementam<br>joodbms.core.<br>SimpleTypeSerializer | Serializa, materializa e compara valores armazenados em *ByteBuffers*. É utilizada pelo serializador de objetos para a construção das tuplas, e as classes que implementam SimpleTypeSerializer são utilizadas também pelo gerenciador de índices para a comparação dos valores nas tuplas, durante os processos de inserção, consulta e atualização.. |
| Grenciador de metadados | joodbms.impl.metadata.<br>MetadataManagerImpl<br>joodbms.metadata.<br>AttributeDescriptor<br>ClassMetadata<br>IndexMetadata | Gerencia os metadados referente aos objetos armazenados no banco de dados, através de classes que descrevem as classes e os atributos persistentes, indicando, respectivamente, a página em que a árvore está armazenada, e qual a ordem dos atributos dentro da tupla. Além dos dados sobre as classes, também gerencia os dados sobre os índices secundários |

Esta arquitetura permite que os diversos módulos do sistema possam ser substituídos, sendo que toda a iteração entre eles é definida pelas interfaces no pacote *joodbms.core*. Além disso, temos a classe *joodbms.core.JoodbmsServerSession*, que contém informações sobre a sessão de banco de dados aberta pelo cliente. Estas informações são passadas a todos os métodos que lidam com o estado da sessão – ou seja, os metadados e serializadores. O usuário tem acesso apenas a um subconjunto destes métodos, definidos na interface *joodbms.JoodbmsSession*. Muitos dos métodos de *JoodbmsServerSession* apenas delegam sua funcionalidade para os módulos de gerenciamento de metadados e modificação de dados.

Tabela 3-2. Quadro comparativo com outros sistemas gerenciadores de bancos de dados

| Banco de Dados | Tecnologia | Persistência Transparente | Coleta de lixo | Linguagem de programação | Herança | Polimorfismo | Encapsulamento | Licença | Modular |
|---|---|---|---|---|---|---|---|---|---|
| Postgresql | ORDBMS | Não | Não | Pgsql, Java, C e outras | Sim | Sim | Não | Código aberto | Não |
| Oracle | ORDBMS | Não | Não | PSQL Java | Sim | Sim | Sim | Proprietário | - |
| Ozone | OODBMS | Sim | Sim | Java | Sim | Sim | Sim | Código aberto | Não, exceto no módulo de armazenameento |
| Caché | OODBMS | Não | Não | Caché ObjectScript Caché Basic | Sim | Sim | Sim | Proprietário | - |
| DB4O | OODBMS | Sim | Não | Java e C# | Sim | Sim | Sim | Código aberto | Módulos altamente acoplados |
| eyeDB | OODBMS | Sim | Não | C++/Java | Sim | Sim | Sim | Código aberto | - |
| JOODBMS | OODBMS | Sim | Não | Java | Sim | Sim | Sim | Código aberto | Módulos fracamente acoplados |

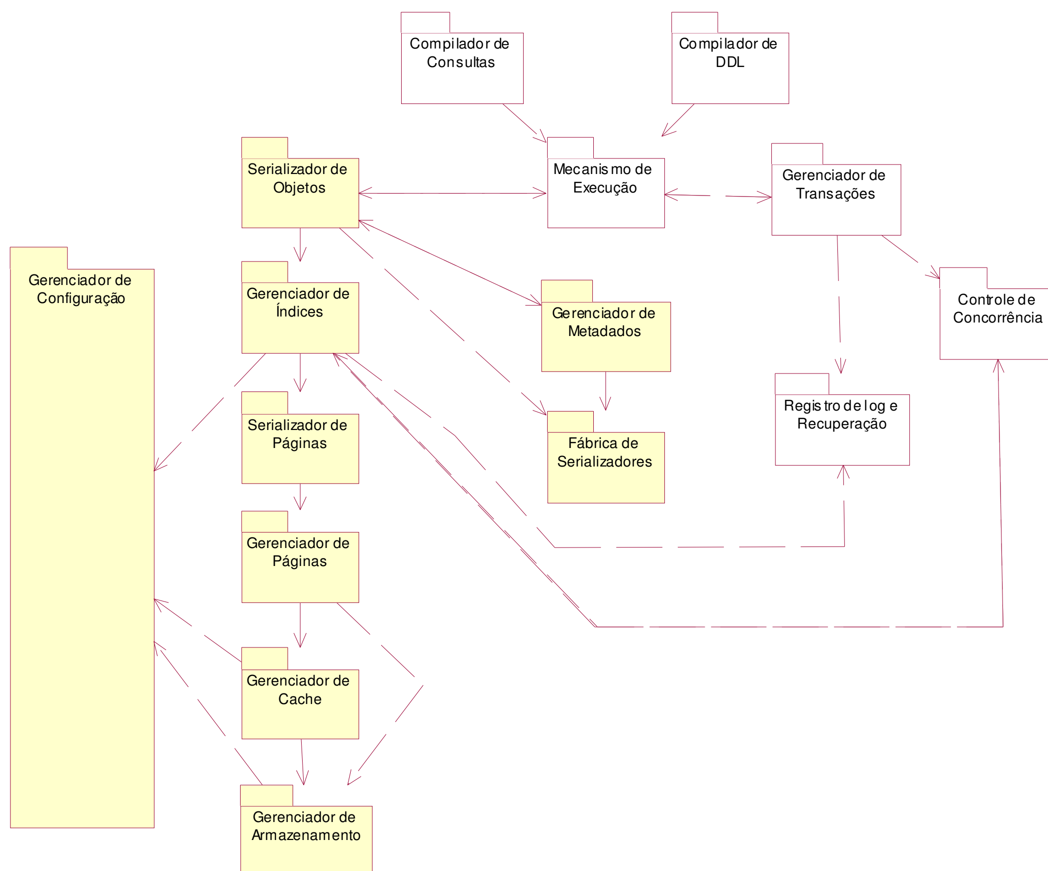



Figura 3.1. Módulos utilizados no JOODBMS. Os módulos já implementados estão marcados em cinza.

O JOODBMS consiste em um gerenciador de estado e *container* para os diversos módulos que compõem o sistema, conforme a Figura 3.2. A interface *joodbms.JoodbmsInstance* é acessada pelo cliente do banco de dados, sendo utilizada para as tarefas de configuração e obtenção da sessão do banco de dados. Internamente, a interface *joodbms.JoodbmsInstanceImplementor* fornece os mecanismos de acesso a todos os módulos necessários para a execução do SGBD.

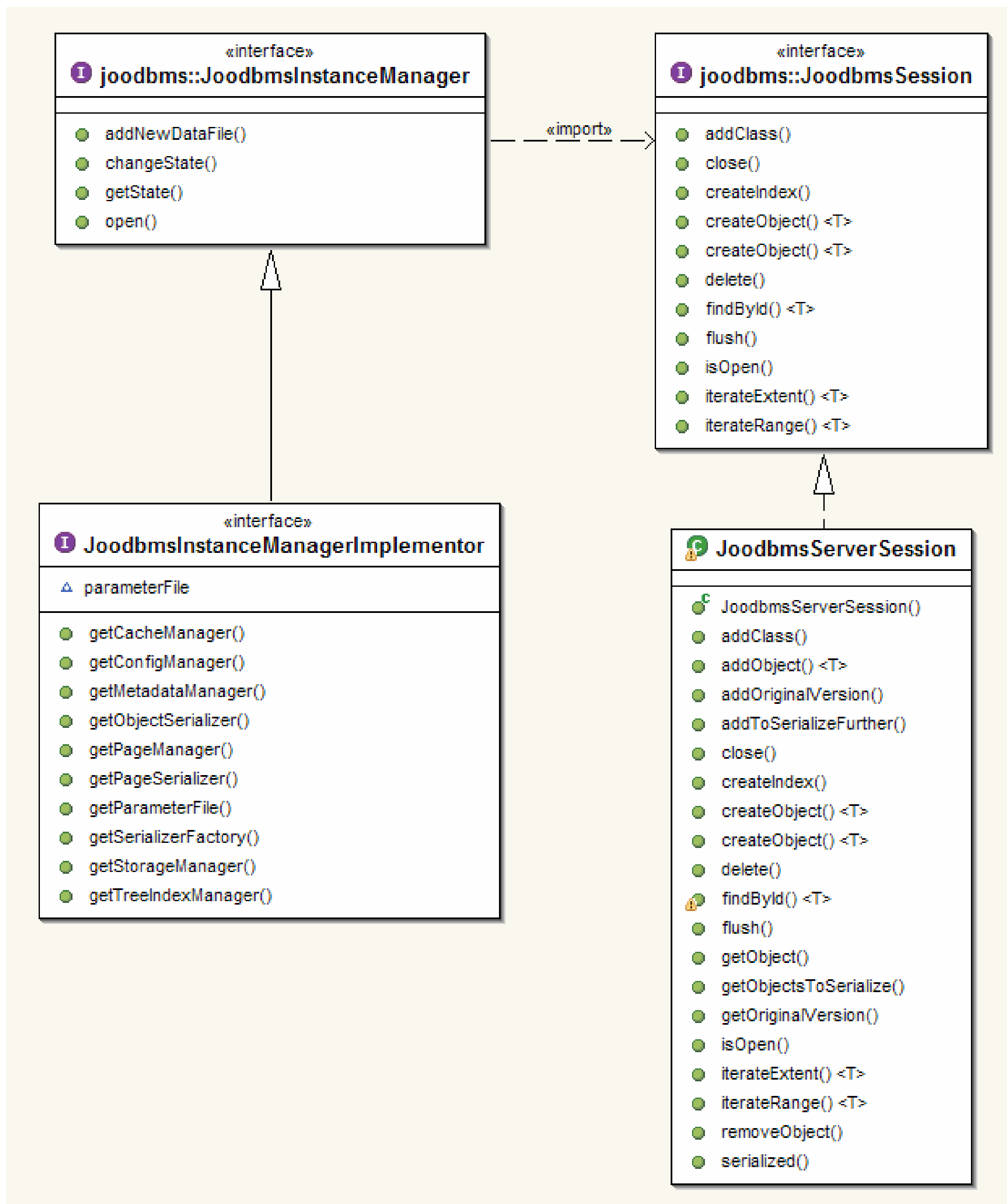



Figura 3.2. Container dos módulos e interface de sessão

## 3.1 - ARQUITETURA DO JOODBMS

A arquitetura desenvolvida permite uma fácil compreensão das responsabilidades de cada um dos módulos, os quais possuem responsabilidades claramente definidas. A implementação utiliza alguns pacotes e sub-pacotes para distinguir os diferentes módulos.

Na Figura 3.3 está o diagrama de pacotes de parte do sistema, incluindo os relacionamentos entre os pacotes. O pacote *joodbms* contém a interface pública do sistema, sendo ela utilizada pelos usuários. No pacote *joodbms.core* estão as interfaces e classes internas do sistema. No pacote *joodbms.impl* e sub-pacotes estão as implementações dos módulos.

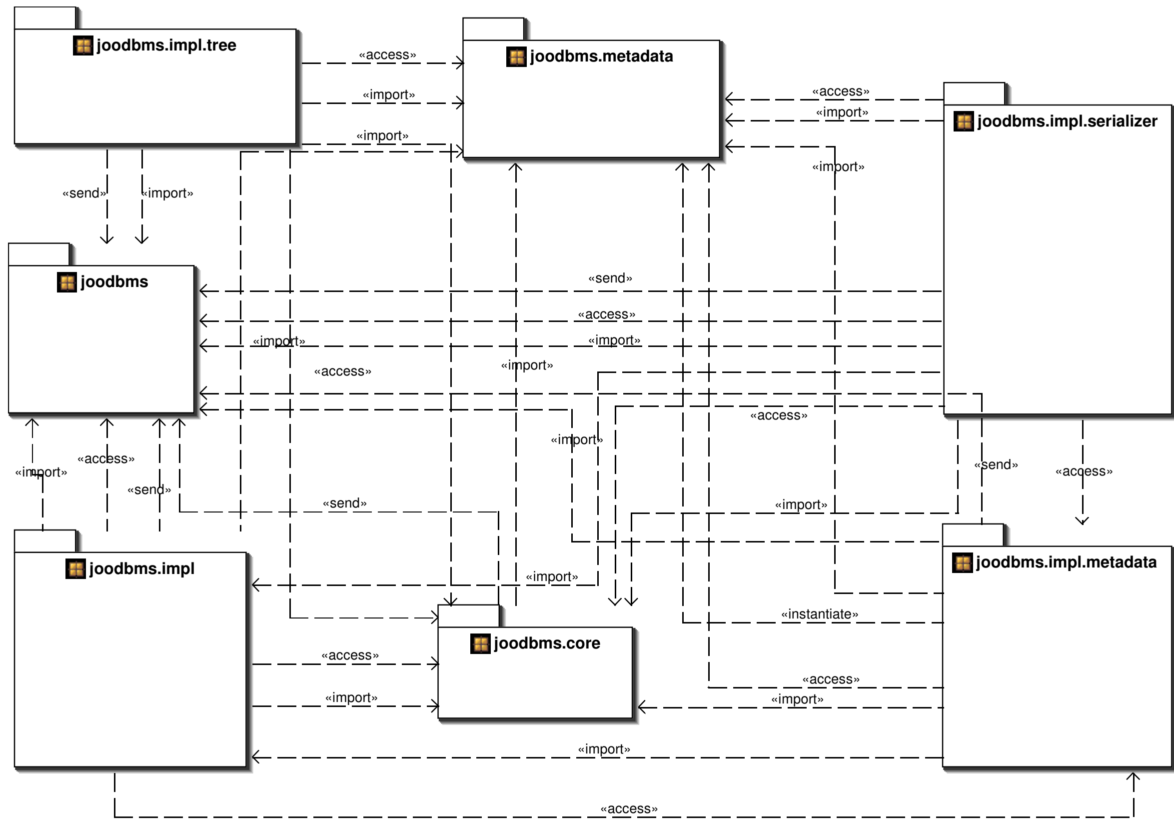



Figura 3.3. Pacotes existentes na implementação do JOODBMS

Todas as implementações devem também utilizar como superclasse a classe *joodbms.core.DatabaseModule*, a qual contém os métodos necessários para o controle do estado do banco de dados. Na Tabela 3-3 estão enumerados os métodos desta classe, com suas respectivas funções.

Tabela 3-3. Descrição dos métodos de *joodbms.core.DatabaseModule*

| Método | Descrição | Uso |
|---|---|---|
| start() | Inicia o módulo, preparando-o para aceitar requisições feitas por outros módulos | Utilizado pela implementação do gerenciador de estados, *joodbms.impl.JoodbmsInstanceManagerImpl* para mudar o estado do banco de dados para *WARM*, *ONLINE* ou *CLOSED*. Deve ser sobrescrita pelas subclasses. |
| stop() | Pára a execução do módulo | Utilizado pela implementação do gerenciador |

| | | de estados para mudar o estado do banco de dados para *WARM*, *ONLINE* ou *CLOSED*. Deve ser sobrescrita pelas subclasses. |
|---|---|---|
| acquireXStateLock() | Obtém um bloqueio exclusivo no módulo, para controle de estado | Utilizado pelas subclasses, quando da alteração de algum parâmetro de configuração. |
| releaseXStateLock() | Libera o bloqueio exclusivo no módulo, após alguma alteração no estado | Utilizado pelas subclasses, quando da alteração de algum parâmetro de configuração. |
| acquireSharedStateLock() | Obtém um bloqueio compartilhado, para a execução dos métodos do módulo | Utilizado pelas subclasses, quando da execução de algum método do módulo, impedindo que ocorra uma alteração de configuração concorrentemente. |
| releaseSharedStateLock() | Libera o bloqueio compartilhado obtido através de acquireSharedStateLock() | Utilizado pelas subclasses, quando da execução de algum método do módulo, impedindo que ocorra uma alteração de configuração concorrentemente. |
| isRunning() | Verifica se o módulo em questão está em execução | Utilizado pelas subclasses, para verificar se os módulos dependentes estão em execução |

### 3.1.1 - O gerenciador de configurações

O gerenciador de configurações, cuja implementação deve utilizar a interface *joodbms.core.ConfigurationManager*, mostrada na Figura 3.4, é um bloco compartilhado por todos os outros módulos. Este módulo define os parâmetros que podem ser alterados no arquivo de configuração. É responsabilidade das implementações deste módulo manter o arquivo de parâmetros de configuração sempre atualizado, de forma que, quando o banco de dados for interrompido, os dados possam ser recuperados sem falha.

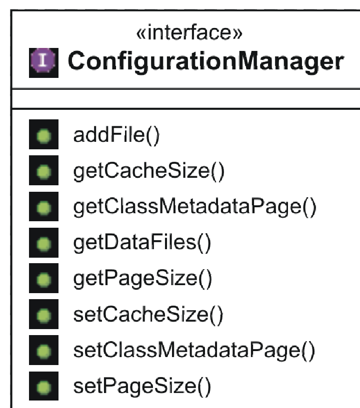



Figura 3.4. Interface *joodbms.core.ConfigurationManager*

Alguns destes parâmetros também são alterados automaticamente pelo SGBD, como, por exemplo, o número da página com o índice dos metadados.

É necessário também que este módulo defina padrões razoáveis para os parâmetros de configuração que porventura não tenham sido informados.

Este módulo foi implementado na classe *joodbms.impl.ConfigurationManagerImpl*, e utiliza o armazenamento da configuração em um arquivo de propriedades padrão da linguagem Java, através da classe *java.util.Properties.* Na figura xx temos um exemplo do arquivo de configuração utilizado, com suas informações detalhadas na Tabela 3-4.

Tabela 3-4. Propriedades utilizadas na configuração do JOODBMS

| Propriedade | Descrição |
|---|---|
| pageSize | Tamanho da página no arquivo de dados, em byte. Também define o tamanho da página de memória utilizada no cachê |
| dataFiles | Lista de arquivos de dados utilizados pelo JOODBMS, separados pelo símbolo "$" |
| classMetadataPage | Página que contém a extensão da classe *joodbms.metadata.ClassMetadata*, utilizada para a obtenção dos metadados das classes armazenadas no JOODBMS |

### 3.1.2 - O gerenciador de armazenamento

Esta camada é responsável pelo gerenciamento dos arquivos que fazem parte do banco de dados. Cada arquivo é identificado por um número único, atribuído quando o mesmo é adicionado no banco de dados. A classe deve implementar a interface *joodbms.core.StorageManager*, conforme a Figura 3.5. Os métodos estão detalhados na Tabela xx.

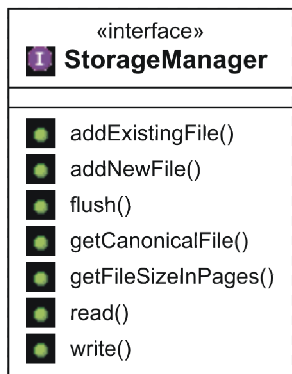



Figura 3.5. Interface do Gerenciador de Armazenamento

Após um arquivo ser adicionado, não é mais possível removê-lo do banco de dados. Extensões futuras neste módulo podem permitir esta operação, criando um mapa de tradução de endereços.

Tabela 3-5. Descrição dos métodos existentes no Gerenciador de Armazenamento

| Método | Descrição |
|---|---|
| addExistingFile() | Adiciona ao sistema gerenciador de banco de dados um arquivo já existente, retornando o número identificador do arquivo nas estruturas de dados deste módulo |
| addNewFile() | Adiciona ao sistema gerenciador de banco de dados um novo arquivo de dados, informando o caminho completo e o tamanho do arquivo. O arquivo não deverá existir anteriormente no sistema de arquivos. Retorna o número identificador do arquivo nas estruturas de dados deste módulo |
| flush() | Confirma quaisquer gravação em arquivos que estejam pendentes nos *buffers* do sistema operacional. |
| getCanonicalFile() | Obtém o caminho canônico para o arquivo especificado. |
| getFileSizeInPage() | Obtém o tamanho do arquivo especificado, em número de páginas |

As operações nos arquivos utilizam com a classe *java.nio.ByteBuffer*, efetuando a transferência de dados entre os arquivos e a memória de forma otimizada, caso os módulos que utilizem este gerenciador utilizem *ByteBuffers* diretos.

As exceções lançadas levam em consideração o tipo de erro que ocorreu. Em caso de erro de E/S, a exceção lançada deve ser *joodbms.JoodbmsFatalException*, que as camadas superiores devem capturar e informar ao gerenciador de instância que houve um erro fatal e que todas as operações a partir deste momento estarão comprometidas.

A implementação padrão utiliza a estrutura fornecida pelo pacote *java.nio*, que é capaz de fazer operações com arquivos diretamente sobre os *ByteBuffers*, quando os mesmos são diretos. Não é suportado o crescimento automático dos arquivos de banco de dados, sendo os mesmos pré-alocados quando de sua criação. Desta forma, tenta-se manter os blocos em disco de forma contígua, reduzindo o número de deslocamentos da cabeça de leitura/gravação durante as operações com o banco de dados.

Os arquivos são acrescentados em uma *array*, e o acesso aos mesmos, pelas camadas superiores, é feito através do identificador do arquivo neste *array*. Este identificador é obtido através dos métodos *addNewFile()* e *addExistingFile*().

### 3.1.3 - O Gerenciador de *Cache*

A interface *joodbms.core.CacheManager*, mostrada na Figura 3.6, tem por finalidade fornecer um sistema de *cache* de páginas físicas. Este *cache* tem o requisito básico de

permitir o acesso por várias *threads* simultaneamente, e serializar estes acessos ao gerenciador de páginas, de forma a apenas uma *thread* de cada vez fazer o acesso a ele.

O gerenciador de *cache* utiliza a interface *joodbms.core.PageBuffer* para representar um *buffer* de página em memória principal, responsável por fornecer às camadas superiores métodos para gerenciar o estado da página.

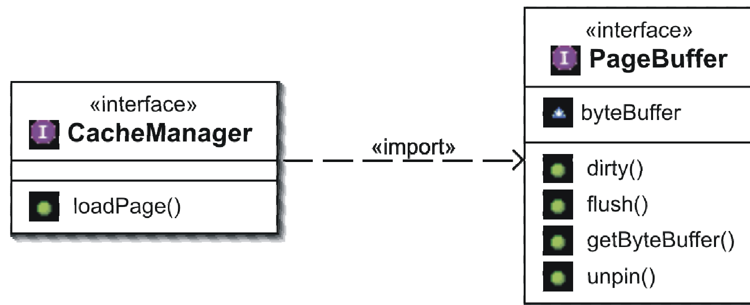



Figura 3.6. Interface *joodbms.core.CacheManager* e a interface *joodbms.core.PageBuffer*

Esta interface possui apenas um método, *loadPage()*, cujo argumento é um objeto do tipo *joodbms.core.PageIdentifer*, contendo o identificador do arquivo, conforme retornado pelo gerenciador de páginas, e o número da página no interior do arquivo. Este método lança as exceções *joodbms.JoodbmsException* se o módulo estiver sendo interrompido durante uma operação de E/S, e *joodbms.JoodbmsFatalException* se ocorrer algum erro de E/S no módulo de gerência do armazenamento.

Os métodos da interface *PageBuffer* estão documentados na Tabela 3-6.

Tabela 3-6. Métodos da interface *PageBuffer*

| **Método** | **Descrição** |
|---|---|
| getByteBuffer() | Obtém o *ByteBuffer* que contém os dados da página. Este *ByteBuffer* é válido enquanto a página ainda estiver fixa (*pinned*) em memória. |
| dirty() | Indica ao gerenciador de *cache* que a página teve seus dados alterados, e é necessário devolvê-los para o disco. |
| flush() | Indica ao gerenciador de *cache* que esta página deve ser gravada em disco, caso seu conteúdo tenha sido modificado. |
| unpin() | Indica ao gerenciador de *cache* que a página pode ser liberada da memória, caso haja necessidade. |

Na implementação padrão, na classe *joodbms.impl.CacheManagerImpl*, um *cache LRU* permite que várias *threads* obtenham páginas simultaneamente. Neste *cache*, caso mais de uma *thread* acesse a mesma página, apenas uma cópia dela será carregada, e os dados serão compartilhados entre as linhas de execução. Foi escolhida a utilização de um *LRU* devido à localidade temporal do acesso às páginas dos índices, em que as páginas mais próximas da raiz tendem a ser mais acessadas que as páginas folha. Assim, o *LRU* tende a manter estas páginas em memória RAM, uma vez que sua freqüência de acesso é alta. Todos os *buffers* são previamente alocados, de forma que pode haver contenção, caso várias páginas estejam sendo solicitadas simultaneamente. É responsabilidade do chamador verificar e ordenar as chamadas, de forma a evitar *deadlock*.

A classe *joodbms.util.LRUList* implementa a estrutura de dados principal utilizada neste módulo, à exceção das características de sincronização, sendo que estas últimas ficam a cargo da própria classe *CacheManagerImpl*. A classe *LRUList* é implementada utilizando uma lista duplamente encadeada, além de um mapa *hash* para a localização rápido dos elementos, quando o mesmo é movido para a frente da lista.

### 3.1.4 - O Gerenciador de Páginas

A implementação da classe *joodbms.core.PageManager*, mostrada na Figura 3.7, é responsável pela tradução entre os números de páginas lógicas para os números de páginas físicas – separando, inclusive, em qual arquivo a página se encontra. Também está dentro de suas responsabilidades o controle de páginas em uso e a alocação de novas páginas.

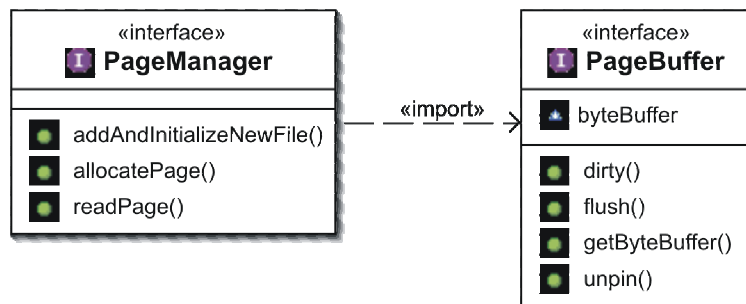



Figura 3.7. Interface *joodbms.core.PageManager* e a interface *joodbms.core.PageBuffer*

Este módulo acessa os dados tanto via gerenciador de *cache* – para a obtenção dos dados dos arquivos –, como também via gerenciador de armazenamento – com a finalidade de acrescentar um novo arquivo de dados. A documentação dos métodos está na Tabela 3-7.

Tabela 3-7. Métodos da interface *PageManager*

| Método | Descrição |
|---|---|
| addAndInitializeNewFile() | Cria um novo arquivo do tamanho especificado, acrescentando no mesmo informações de controle que permitam ao gerenciador de páginas identificar de forma precisa se um determinado arquivo de dados pertence a um banco de dados específico |
| allocatePage() | Aloca uma página nas proximidades da página informada, retornando o número lógico da página alocada |
| readPage() | Lê a página especificada do arquivo de dados, retornando uma estrutura com uma referência para o *ByteBuffer* que contém os dados correspondentes, e mantém a página fixa em memória. |

Para garantir que todos os arquivos de dados adicionados façam parte do mesmo banco de dados, é gerado um UUID quando da criação do primeiro arquivo de dados, e o mesmo é armazenado no cabeçalho de todos os arquivos associados ao banco. Este módulo, ao iniciar, verifica, utilizando o gerenciador de armazenamento, todos os arquivos já associados ao banco, checando se os mesmos possuem o mesmo UUID.

A implementação fornecida utiliza algumas páginas de *bitmap* espalhadas pelo arquivo para o controle da alocação de espaço. O espaçamento entre as páginas depende do tamanho em bytes da página, sendo que cada bit do *bitmap* corresponde a uma página do bloco de páginas subseqüente. Assim, é descontando-se o cabeçalho da página de *bitmap*, de 32 bytes, e os bytes restantes são utilizados para a montagem do mapa de bits. O primeiro *bitmap* é colocado no primeiro bloco lógico do arquivo, e são espaçados de forma igual ao longo do mesmo.

Também é o gerenciador de páginas o responsável por iniciar corretamente os valores no cabeçalho de cada arquivo existente no SGBD, indicando o UUID (*Universal Unique Identifier*) correspondente ao banco, e informações sobre o sequenciamento do arquivo dentro do banco de dados.

O objetivo do alocador de páginas desenvolvido é agrupar as páginas em torno da página fornecida. Assim pode-se tentar, ao criar uma estrutura em árvore, manter todos os seus blocos próximos, minimizando os tempos de *seek* no disco, para a busca de páginas.

### 3.1.5 - O Serializador de Páginas

A função do serializador de páginas é recuperar ou criar as páginas do armazenamento persistente, mas já devolvendo ao chamador o objeto corresponde ao tipo de página correta. Este módulo deve implementar a interface *joodbms.core.PageSerializer*, mostrada na Figura 3.8.

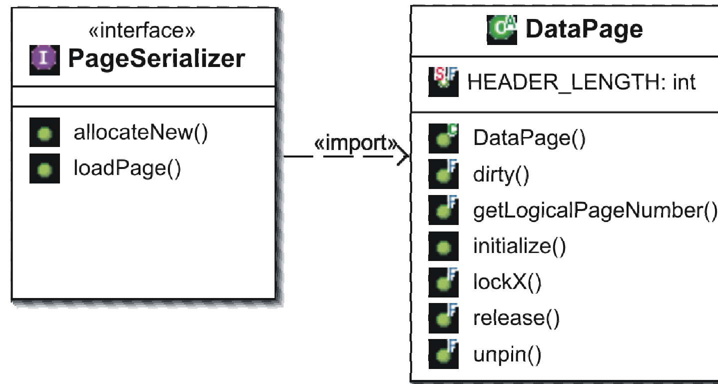



Figura 3.8. Interface *joodms.core.PageSerializer* e classse *DataPage*, que contém os dados da página

Há apenas dois métodos disponíveis no serializador de páginas:

1. *allocateNew()*, utilizado para alocar uma nova página, retornando o objeto correspondente (derivado de DataPage) ao tipo solicitado. Este método utiliza os serviços de alocação de páginas no gerenciador de páginas.

2. *loadPage()*, responsável por carregar, a partir do arquivo de dados ou da memória *cache*, os dados correspondentes à página e encapsular em um objeto derivado de DataPage correspondente ao tipo especificado nos primeiros bytes da página.

A classe *joodbms.core.DataPage*, detalhada na Tabela 3-8, encapsula os dados da página e fornece alguns métodos para o gerenciamento dos dados do cabeçalho. Esta estrutura de página define o tamanho do cabeçalho como 32 bytes e algumas estruturas básicas necessárias, como o tipo de página, armazenado nos dois primeiros bytes da página e informações adicionais para o bloqueio de páginas, com 8 bytes como identificador da *thread* que solicitou bloqueio exclusivo, e mais 4 bytes para um contador de bloqueios compartilhados.

Utilizando o campo com o tipo de página é possível para o serializador identificar qual a classe correspondente e criar o objeto, encapsulando o conjunto de bytes que faz parte do bloco.

Tabela 3-8. Métodos de *joodbms.core.DataPage*

| **Método** | **Descrição** |
|---|---|
| dirty() | Indica que os dados da página foram modificados, e ela precisa ser gravada novamente em disco |
| getLogicalPageNumber() | Obtém o número lógico da página correspondente aos dados carregados em memória |
| initialize() | Coloca os valores padrão na página, chamada pelo método allocateNew() do serializador de páginas, quando durante a inicialização de uma nova página |
| lockX() | Obtém um bloqueio exclusivo na página. Caso a página possua um bloqueio compartilhado, o mesmo será atualizado para um bloqueio exclusivo. Se alguma outra *thread* possuir um bloqueio (exclusivo ou compartilhado), a *thread* atual irá aguardar até todos os outros bloqueios serem liberados. |
| lockShared() | Obtém um bloqueio compartilhado na página. Caso outra *thread* possua um bloqueio exclusivo, a *thread* solicitando o bloqueio compartilhado irá aguardar até o bloqueio exclusivo ser liberado |
| release() | Libera o bloqueio (exclusivo ou compartilhado) na página atual |

O serializador desenvolvido, implementado na classe *joodbms.impl.PageSerializerImpl*, utiliza uma tabela estática de tipos de páginas, mas nada impede que outros serializadores sejam acrescentados, que sejam capazes de um registro dinâmico das classes correspondentes aos tipos de dados. Nesta implementação, a principal diferença entre o método *allocateNew*() e *loadPage()* se dá pelo fato de, em *allocateNew()*, a ser chamado o método *initialize()* da página recém-criada, para deixar seus dados com os valores padrão.

Há já um tipo de página de dados definida, *joodbms.impl.pages.TuplePage*, derivada da classe *DataPage*, citada acima, a qual as páginas que utilizam tuplas para armazenamento de dados podem utilizar como superclasse. Seus métodos e propriedades protegidas estão descritos na Tabela 3-9.

Tabela 3-9 . Documentação dos métods da classe TuplePage

| **Método/Propriedade** | **Descrição** |
|---|---|
| usedKeys | Propriedade que define o número de tuplas que existem nesta página |
| freeSize | Espaço livre na página. Este espaço livre leva em consideração o número de bytes ocupado pelo cabeçalho da página e quaisquer estruturas auxiliares |
| getTuplePosition() | Método protegido utilizado pelas subclasses para obter a posição, no *ByteBuffer*, correspondente ao início da tupla especificada |
| getTupleSize() | Método protegido utilizado pelas subclasses para obter o tamanho, em bytes, |

|  | de uma tupla especificada |
|---|---|
| moveItensTo() | Move um certo número de tuplas, a partir da tupla especificada, para outra tupla |
| deleteEntries() | Exclui um conjunto de tuplas da página atual |
| getNumberOfTuples() | Obtém o número de tuplas existentes na página |
| getTuple() | Obtém a tupla especificada, utilizando os descritores de atributo informados |
| insertTuple() | Adiciona uma tupla na posição especificada |
| updateTuple() | Atualiza a tupla que está armazenada na posição especificada |
| getTotalSize() | Obtém o tamanho total da página |
| getUsedSpace() | Obtém o espaço ocupado, em bytes, na página |
| iterateTuples() | Retorna um iterador para visitar todas as tuplas existentes na página. Este iterador implenta a interface *java.lang.Iterable* |

### 3.1.6 - O Gerenciador de Índices em Árvore

O gerenciador de índices em árvore permite ao SGBD a criação e manutenção de estruturas de dados indexadas. Sua implementação deve utilizar a interface *joodbms.core.TreeIndexManager*, mostrada na Figura 3.9. É responsabilidade deste módulo a criação de índices e o acréscimo, remoção e consulta de tuplas no mesmo, mantendo-as em ordem crescente pela chave.

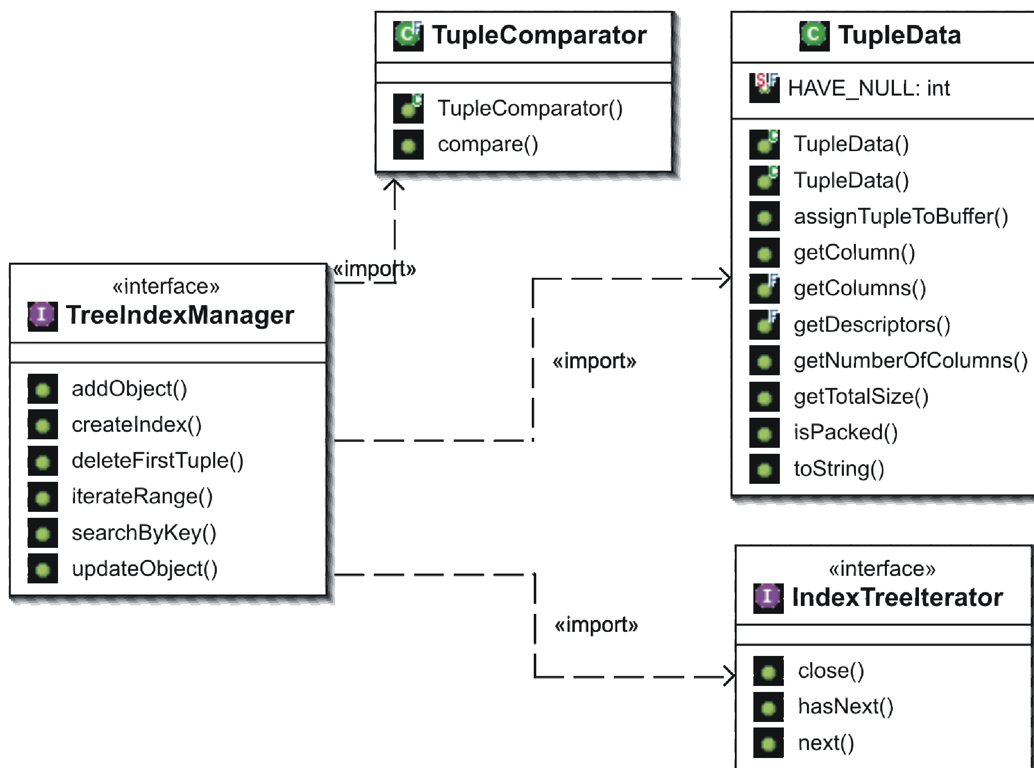



Figura 3.9. Interface *joodbms.core.TreeIndexManager* e classes e interfaces relacionadas

Como se pode observar, este módulo é composto de diversas classes e interfaces associadas. A principal interface é a *joodbms.core.TreeIndexManager*, cujos métodos estão detalhados na Tabela 3-10, que é o ponto de entrada para este módulo. A classe *joodbms.core.TupleComparator* é utilizada pelos métodos do gerenciador de índices para, durante as operações de inserção, pesquisa e atualização de tuplas, localizar a posição correta no índice para a efetivação da operação. A classe *joodbms.core.TupleData* encapsula a região de memória que contém a tupla, cuja estrutura está mostrada na Figura 3.10, e fornece os métodos necessários para acessar e recuperar as colunas que contém a tupla.

| Número de colunas (1 byte) | Atributos da tupla (2 bytes) | Bitmap de atributos nulos (1 bit por atributo, alinhado em 8 bits) | Identificador do objeto (16 bytes) | Dados dos atributos não nulos |
|---|---|---|---|---|

Figura 3.10. Formato da tupla

Tabela 3-10. Documentação dos métodos utilizados pelo módulo de gerenciamento de índices

| **Classe/Interface** | **Método** | **Descrição** |
|---|---|---|
| TreeIndexManager | addObject() | Adiciona a tupla no índice cuja página de configuração é a página especificada. |
| | createIndex() | Cria um novo índice com a configuração especificada, retornando o número da página de configuração, a ser utilizada nas referências futuras |
| | deleteFirstTuple() | Exclui a primeira tupla que satisfaz a condição de pesquisa |
| | iterateRange() | Cria um objeto que implementa a interface *IndexIterator*, para varrer o índice no intervalo de chaves especificada, incluindo os extremos |
| | searchByKey() | Retorna a primeira tupla que satisfaz a condição de consulta informada |
| | updateObject() | Atualiza a tupla especificada |
| TupleData | assignTupleToBuffer() | Copia os dados da tupla para o *ByteBuffer* especificado |
| | getColumn() | Obtém um *ByteBuffer* correspondendo à coluna especificada |
| | getColumns() | Obtém um *array* de *ByteBuffer* correspondente à faixa de colunas especificadas |
| | getDescriptors() | Obtém os descritores (metadados) de cada um dos atributos presentes na tupla |
| | getNumberOfColumns() | Obtém o número de colunas/atributos nesta tupla |
| | getTotalSize() | Obtém o tamanho total (em bytes) ocupado por esta tupla |
| IndexTreeIterator | hasNext() | Retorna ao chamador indicando se há novas tuplas neste iterador |
| | next() | Obtém a próxima tupla |

| | close() | Fecha o iterador, liberando quaisquer bloqueios pendentes nas páginas |
|---|---|---|

A implementação feita foi de uma árvore B+, ou seja, os dados são localizados apenas nos nós folha, facilitando as operações de caminhamento em ordem. A codificação foi feita de forma a utilizar estruturas em pilha, ao invés das versões recursivas dos algoritmos de inclusão e pesquisa de elementos.

O algoritmo de inserção de dados, mostrado na Figura 3.11, inicialmente tenta acrescentar a tupla no bloco folha correspondente à chave. Caso não seja possível esta inclusão, o algoritmo divide a página folha em duas, acrescenta o novo elemento, ou na página da esquerda ou na página da direita, e propaga o separador para as páginas internas, repetindo o procedimento até chegar à página raiz. Caso a página raiz seja dividida também, uma nova página raiz é criada, e a configuração do índice é alterada para refletir esta nova mudança.

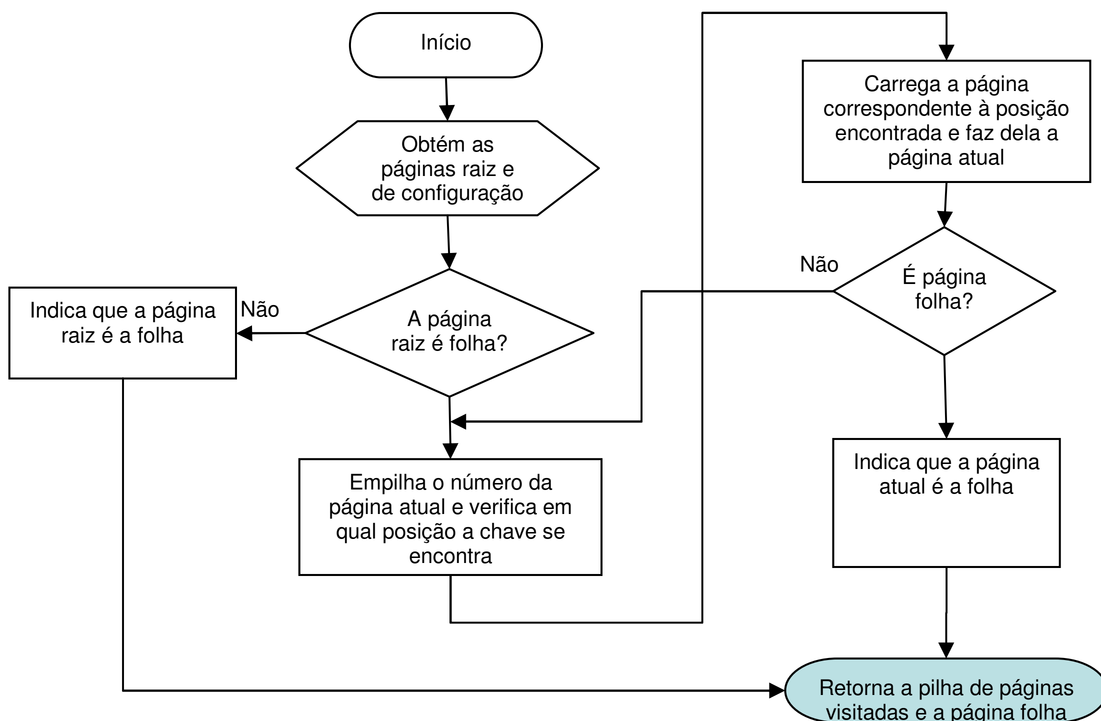



Figura 3.11. Fluxograma da busca da página folha de um elemento

A operação de divisão de páginas pode ser acompanhada pelo fluxograma mostrado na Figura 3.12.

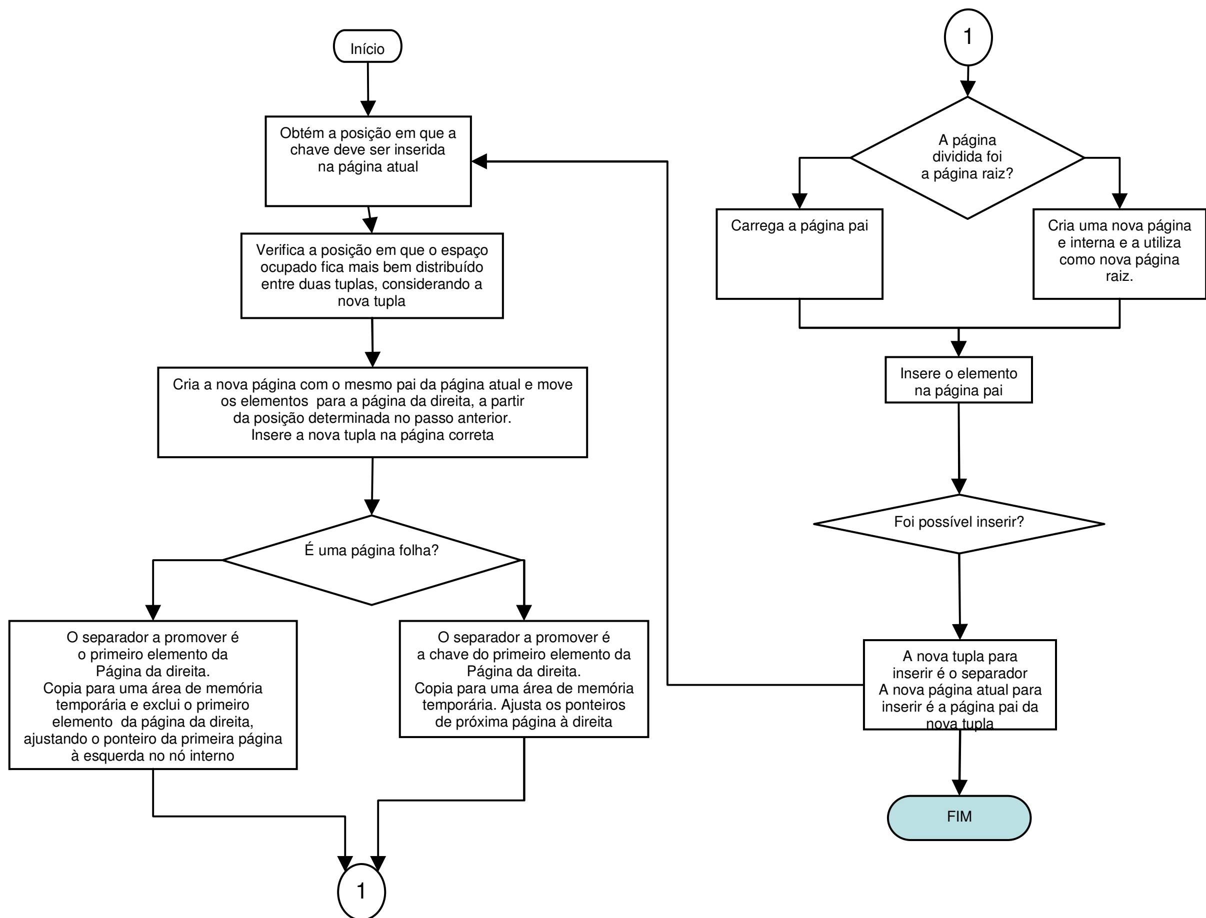



Figura 3.12. Algoritmo recursivo para divisão de páginas

O gerenciador de índices em árvore implementado, presente no pacote *joodbms.impl.tree*, utiliza as páginas de dados e classes mostradas na Figura 3.13. Utilizou-se a possibilidade, fornecida pelo gerenciador de páginas, de se poder ter uma classe específica para cada tipo de página, e assim foi criada a hierarquia de classes, iniciando pela classe raiz *BTreePage*. Assim, as páginas da árvore B+ foram divididas nas classes *BTreeLeafPage*, que contém os objetos serializados, e *BTreeInnerPage*, contendo as folhas internas da árvore, ou seja, os separadores para os objetos filhos.

A página folha, *BTreeLeafPage*, contém um ponteiro para a próxima página folha, permitindo o caminhamento pelo índice na ordem direta. Na página interna, *BTreeInnerPage*, há um ponteiro para a página descendente à esquerda, cujo primeiro valor é menor que o menor valor da página. Também na página interna é armazenado apenas o separador e o ponteiro para a página filha correspondente, que é obtido utilizando o método *getPageForEntry()*. À exceção destas diferenças, as páginas possuem as mesmas características, armazenando as tuplas conforme definido na classe *TuplePage*. Alguns

métodos adicionais são definidos em *BTreePage*, utilizados para a inserção da tupla na posição correta, utilizando a classe *TupleComparator* para identificar qual é esta posição.

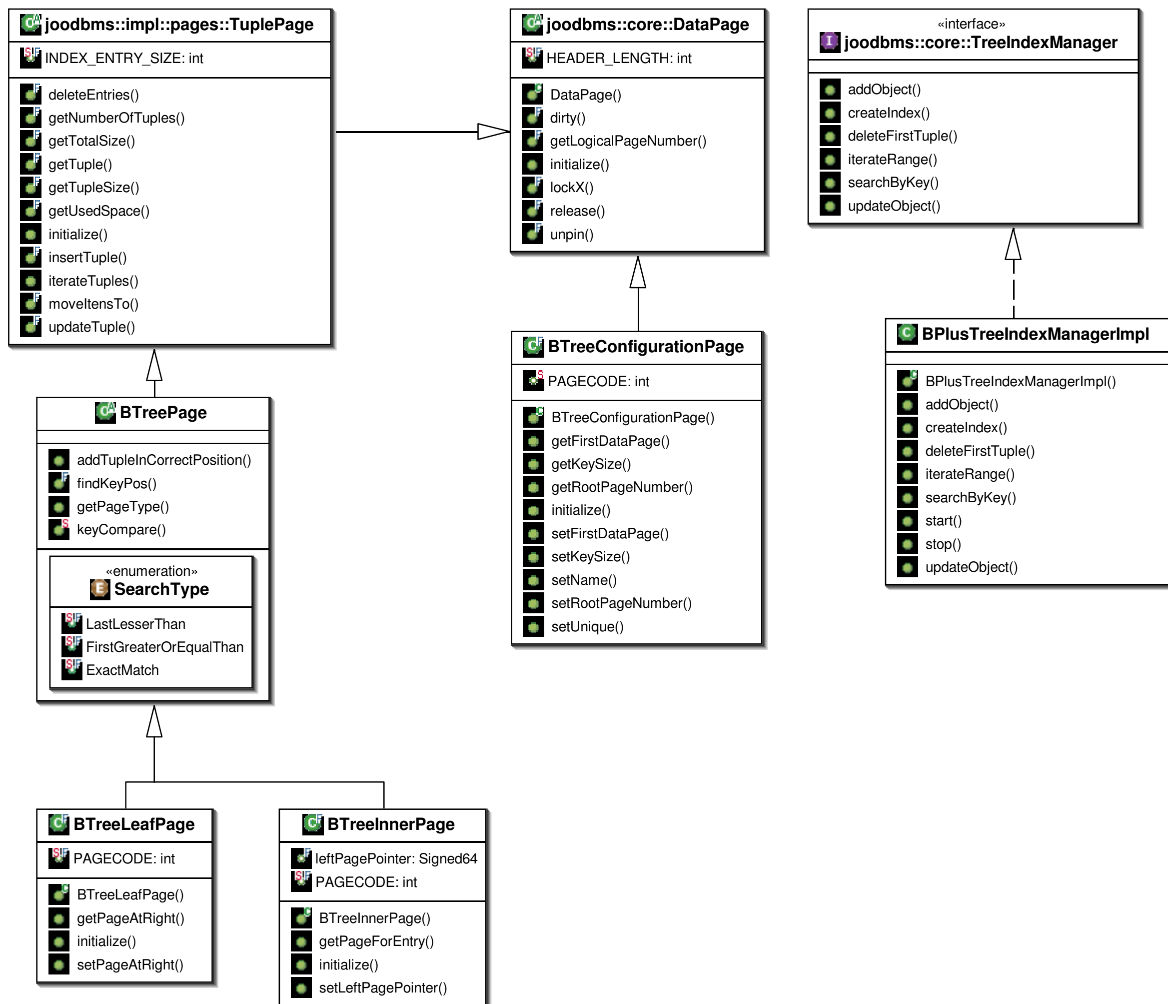



Figura 3.13. Classes utilizadas no gerenciamento de índices

### 3.1.7 - O Serializador de objetos

O serializador de objetos é o módulo responsável por converter os bytes armazenados nas tuplas para os objetos da linguagem Java. A implementação utiliza a interface *joodbms.core.ObjectSerializer*, mostrada na Figura 3.14, juntamente com suas classes dependentes. A documentação dos métodos está na Tabela 3-11.

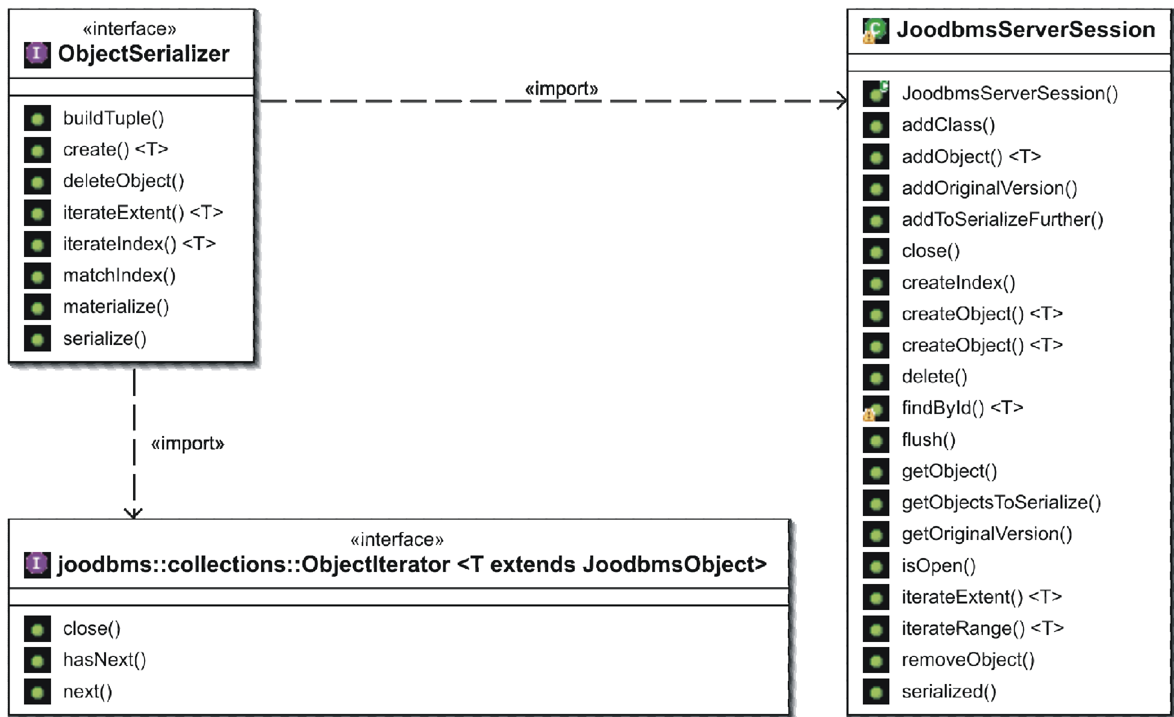



Figura 3.14. Interface *joodbms.core.ObjectSerializer* e dependências

Tabela 3-11. Métodos das classes e interfaces utilizadas no serializador de objetos

| Classe / Interface | Método | Descrição |
|---|---|---|
| ObjectSerializer | buildTuple() | Constrói a tupla correspondente aos atributos especificados, de um objeto. |
| | create() | Cria um *proxy* para um novo objeto da classe especificada, armazenando também uma referência para o mesmo na sessão |
| | deleteObject() | Exclui o objeto especificado do armazenamento persistente e da sessão |
| | iterateExtent() | Itera por todos os objetos de uma classe específica |
| | iterateIndex() | Itera por todos os objetos em uma faixa específica de valores, utilizando um índice |
| | matchIndex() | Busca uma chave específica no índice |
| | materialize() | Materializa o objeto com a chave especificada |
| | serialize() | Serializa o objeto especificado em armazenamento persistente |
| ObjectIterator | hasNext() | Verifica se há um próximo objeto neste iterador |
| | next() | Move o cursor para o próximo objeto |
| | close() | Fecha o iterador, liberando os recursos alocados |

Para retirar do serializador a responsabilidade de conhecer os detalhes de cada tipo primitivo, foi definida a interface *joodbms.core.SimpleTypeSerializer*, cujas classes que a implementam são responsáveis por serializar e materializar os tipos primitivos. Sua descrição será dada juntamente com a fábrica de serializadores.

A implementação utilizada no JOODBMS do serializador de objetos armazena cada atributo em uma coluna da tupla, sendo o primeiro atributo o identificador do objeto, e, em seguida, os atributos das subclasses, começando pelo topo (ou seja, pelos atributos da

classe que descende diretamente de *joodbms.JoodbmsObject*). O serializador padrão fornecido na linguagem Java não foi utilizado por não permitir um controle maior sobre o posicionamento dos atributos dentro do *stream* de bytes resultantes. Além disso, é necessário dar um tratamento especial para o identificador de objeto, e o sistema também precisa de um controle especial sobre quais e quando os objetos são serializados e/ou materializados.

Para reduzir o uso de memória e aumentar o desempenho na recuperação de objetos, são criados *proxies* que efetivamente materializarão um determinado objeto quando for feito o primeiro acesso a seus métodos. Também durante o processo de criação do *proxy* o objeto é associado à sessão que requisitou que o mesmo fosse materializado, e cria-se uma cópia de todos os seus atributos, para verificar, quando do fechamento da sessão, se algum objeto que foi carregado precisa ser atualizado, e também para permitir as consultas nos índices secundários.

Na implementação atual do JOODBMS, este módulo, além do MetadataManager, são os únicos que possuem acesso à sessão do banco de dados. Este acesso é devido às verificações no *cache* de sessão, checando se um determinado objeto já foi carregado nesta sessão, e também para a verificação de, quando for iniciado o processo de serialização, quais objetos ainda precisarão ser serializados. Este caso ocorre quando os objetos são criados na sessão, já que os mesmos são gravados em disco apenas quando do fechamento ou esvaziamento (*flush*) da sessão.

### 3.1.8 - O Gerenciador de Metadados

O gerenciador de metadados é responsável por armazenar e recuperar os metadados das classes e índices. Sua implementação deve seguir as diretrizes da interface *joodbms.core.MetadataManager*, mostrada na Figura 3.15.

É sua responsabilidade fornecer e alterar os metadados das classes armazenadas no JOODBMS. Possui informações já contidas no próprio código, referente às classes *built-in* que descrevem os metadados. Estas informações são construídas de forma estática, pois não é possível obter estas informações a partir do banco de dados se a estrutura dos metadados não for conhecida de antemão.

Figura 3.15 . Classes responsáveis pelo gerenciador de metadados

Também possui a responsabilidade de adicionar classes e índices nos metadados, e efetuar a criação das estruturas de armazenamento necessárias para estas operações. Os metadados consistem nas classes *joodbms.metadata.ClassMetadata*, *joodbms.metadata.AttributeDescriptor* e *joodbms.metadata.IndexMetadata*. A classe *ClassMetadata* é responsável por manter os dados referentes às classes, como o nome e a página de configuração no índice; em *AttributeDescriptor* estão as informações referentes a cada atributo de uma classe; e em *IndexMetadata* os dados para os índices secundários. As propriedades destas classes estão descritas na Tabela 3-12.

Tabela 3-12. Documentação dos atributos da clas classes que armazenam os metadados

| Classe | Propriedade | Descrição |
|---|---|---|
| ClassMetadata | attributes | Array de atributos da classe, em ordem de armazenamento (o primeiro do array será armazenado logo após os atributos da superclasse) |
| | indexPageNumber | Número da página com a configuração do índice |
| | name | Nome da classe representada por este objeto |
| AttributeDescriptor | length | Tamanho máximo do campo (para os campos com tamanho variável) |
| | name | Nome do atributo, ou *null* se o atributo foi excluído |
| | typeCode | Código do tipo do atributo |
| | haveFixedLength | Indica se o atributo possui tamanho fixo ou variável |
| | array | Indica se o atributo é um array do tipo especificado |
| IndexMetadata | indexPageName | Número da página com a configuração do índice |
| | indexedClass | Referência para o objeto ClassMetadata correspondente à classe sendo indexada |
| | keys | Array de referências para objetos AttributeDescriptor que descrevem os atributos indexados |
| | name | Nome do índice |
| | unique | Indica se o índice é único ou não |

Para permitir que os objetos sejam carregados do disco apenas quando de seu primeiro uso, durante a materialização de um objeto, as referências para outros objetos são substituídas por referências pra um objeto *proxy*, que extende a classe do objeto que seria materializado, e também implementa a interface *joodbms.core.JoodbmsPersistentProxy*.

A criação do *proxy* é feita utilizando a biblioteca CGLIB. A classe gerada utiliza o serializador de objetos para, ao primeiro método chamado, recuperar os dados dos atributos a partir do disco. Uma melhoria neste mecanismo pode ser a utilização de modificação de *bytecode*, de forma a manter a hierarquia de classes intacta.

### 3.1.9 - A Fábrica de Serializadores

A responsabilidade da serialização dos tipos primitivos é delegada para as classes que derivam de SimpleTypeSerializer. A interface *joodbms.core.SerializerFactory* define os métodos necessários para a implementação desta fábrica. As classes envolvidas neste módulo estão na Figura 3.16.

Os serializadores são responsáveis por, a partir dos dados brutos contidos em um *ByteBuffer*, construir o objeto do tipo correto, e também o processo inverso, ou seja, a partir do objeto concreto, montar um *stream* de bytes e armazená-los em um ByteBuffer. Este processo deve seguir o contrato definido para as tuplas, em que as colunas com quantidades variáveis de bytes precisam ter, em seus dois primeiros bytes, a quantidade de espaço (em bytes) ocupados por elas, incluindo, nesta conta, estes dois primeiros elementos.

Figura 3.16. Classes utilizadas no serializador de tipos simples

## 3.2 - TESTES

Outra característica obtida pela modularização consiste na possibilidade da criação de testes unitários. Estes testes são feitos utilizando o JUnit [31], contendo pelo menos uma classe de testes para cada módulo.

Os testes implementados permitem a verificação das seguintes funcionalidades:

1. No gerenciador de armazenamento, a criação de um novo arquivo de dados;

2. No gerenciador de cache, a verificação de que o mesmo objeto de página será retornada para *threads* diferentes, caso seja solicitada a mesma página. Também é verificado se o mecanismo de cache libera corretamente as páginas que não estão marcadas em uso, quando o mesmo está cheio;

3. O serializador de páginas é verificado quanto ao tipo de objeto de página que é instanciado, quando uma página específica for carregada.

4. O gerenciador de índices é verificado em relação ao sequenciamento das chaves, às operações de inserção, alteração e exclusão, divisão de páginas e colocação de itens na página folha.

5. O serializador de objetos é testado quanto à serialização e materialização de objetos que contêm de tipos primitivos, tipos de referência, atualizações e iteração por uma faixa de valores. Dentre estes casos ainda são feitos testes de atualização e auto-referência.

6. Para o gerenciador de metadados, o único teste realizado consiste em checar se o *extent* correspondente aos metadados das classes persistentes foi corretamente armazenado; e

7. O gerenciador de sessões, em que são feitos testes para abrir e fechar sessões, verificando se os objetos são corretamente gravados quando do fechamento da sessão.

## 3.3 - UTILIZAÇÃO DO JOODBMS

A versão atual do JOODBMS foi desenvolvida para ser embutida na aplicação usuária. Assim, toda comunicação entre a aplicação e o banco de dados é feita através de chamadas locais, através dos *proxies* criados automaticamente pelo serializador de objetos. A confirmação dos dados em armazenamento persistente se dá após a sessão ser fechada, ou programaticamente.

Os passos para o acesso ao banco de dados são:

1. Obter uma instância de *joodbms.InstanceManager*. Para isto, deve ser utilizado o construtor de *joodbms.impl.JoodbmsInstanceManagerImpl*, passando como parâmetro o arquivo de configuração.

2. Utilizar o método *open()* da interface *joodbms.InstanceManager*, o qual retorna um objeto que implementa *joodbms.JoodbmsSession*. Nesta interface estão os métodos necessários para as operações de criação, recuperação e exclusão de objetos, além de modificação do esquema.

Um exemplo de utilização é mostrado na Figura 3.17, em que um novo tipo é adicionado, acrescentado um índice e alguns objetos são acrescentados e, em seguida, pesquisados pelo índice.

```
import java.io.File;

import joodbms.DatabaseState;
import joodbms.JoodbmsInstanceManager;
import joodbms.JoodbmsSession;
import joodbms.collections.ObjectIterator;
import joodbms.impl.JoodbmsInstanceManagerImpl;

public class JoodbmsExample {
 public static void main(String[] args) throws Exception {
  JoodbmsInstanceManager instance = new
JoodbmsInstanceManagerImpl(File.createTempFile("test", ".config"));
  instance.changeState(DatabaseState.WARM);
  File dataFile = File.createTempFile("test", ".data");
  dataFile.delete();
  instance.addNewDataFile(dataFile, 10 * 1024 * 1024);
  instance.changeState(DatabaseState.ONLINE);

  JoodbmsSession session = instance.open();
  try {
   session.addClass(DataClass.class);
   session.createIndex("DataClass", "idx01", new String[] { "name" }, false);

   DataClass obj = session.createObject("DataClass");
   obj.setName("My Name");
   obj = session.createObject("DataClass");
   obj.setName("Your Name");
   obj = session.createObject("DataClass");
   obj.setName("Her Name");

   session.flush();

   ObjectIterator<DataClass> iter = session.iterateRange("DataClass", "idx01", new
Object[] {"H"}, new Object[] {"N"});
   while (iter.hasNext()) {
    System.out.println(iter.next().getName());
   }
   iter.close();
  } finally {
   session.close();
  }
 }
}
```

Figura 3.17. Exemplo de utilização do JOODBMS

Neste exemplo é possível verificar que a criação de um objeto é feita de forma quase que transparente, sendo a única exceção a utilização da sessão para a criação dos objetos persistentes. Esta criação deve ser feita utilizando a sessão, ao invés da invocação padrão utilizando o operador *new*, pois o objeto retornado é, em realidade, um *proxy* capaz de monitorar seu estado de persistência. Esta característica é realizada através da criação de uma subclasse que extende a classe concreta e implementa, também, a interface *joodbms.impl.JoodbmsPersistentProxy*.

A definição dos atributos persistentes é feita utilizando os recursos de anotação da linguagem Java, conforme mostrado no exemplo da Figura 3.18. A anotação *@PersistentAttribute* indica que este atributo será armazenado no banco de dados, estando disponível também para pesquisas.

```
import joodbms.JoodbmsObject;
import joodbms.PersistentField;

public class DataClass extends JoodbmsObject {

 @PersistentField
 private String name;

 /**
  * @return Returns the name.
  */
 public String getName() {
  return name;
 }

 /**
  * @param name The name to set.
  */
 public void setName(String name) {
  this.name = name;
 }
}
```

Figura 3.18. Exemplo de classe persistente

A pesquisa pode ser feita de duas formas: ou materializando todos os objetos de um *extent* ou utilizando um índice. No primeiro caso, é responsabilidade do cliente verificar se os atributos estão de acordo com o critério. Já no segundo caso, apenas os objetos que estejam na faixa especificada são materializados pelo JOODBMS e retornados ao cliente. No exemplo mostrado, a segunda alternativa foi utilizada.

## 3.4 - UM EXEMPLO DE APLICAÇÃO UTILIZANDO O JOODBMS

Para demonstrar mais claramente o funcionamento do JOODBMS, foi desenvolvida uma aplicação simples de cadastro de notas fiscais, utilizando a tecnologia *Swing* [24]. Nesta aplicação é possível verificar todas as funcionalidades disponibilizadas pelo banco de dados, conforme apresentadas na seção anterior. Exemplos de código serão dados também, nas partes relevantes à manipulação de dados no banco.

A tela inicial da aplicação é ilustrada na Figura 3.19, que permite a consulta e a criação de novas notas fiscais.

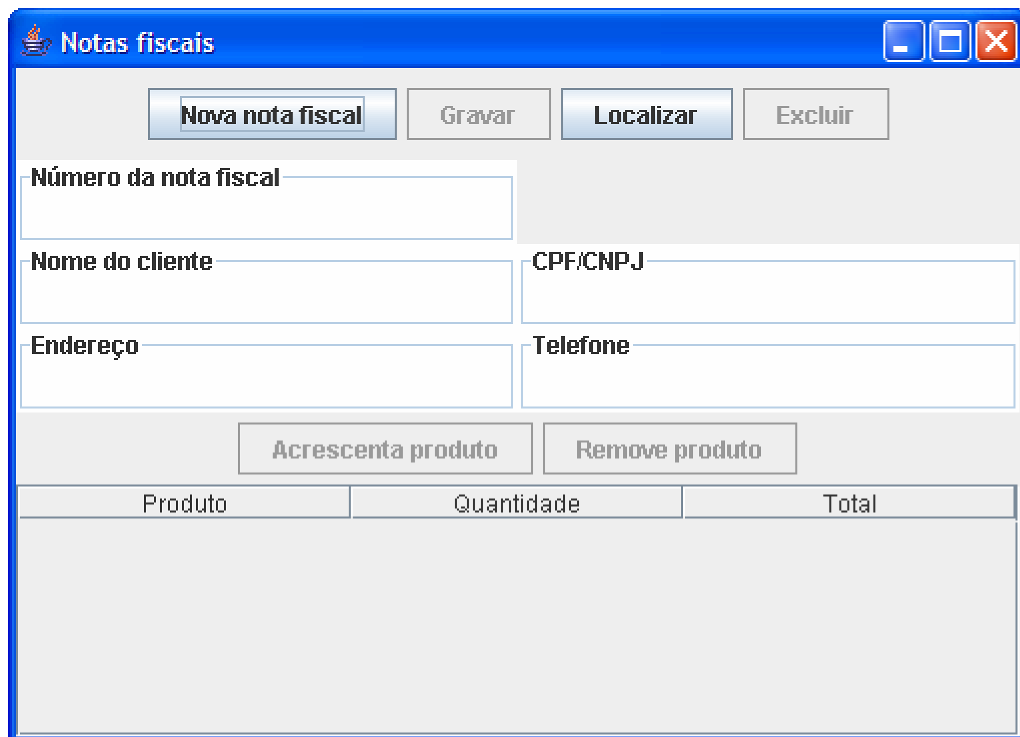



Figura 3.19. Tela inicial do sistema de controle de notas fiscais

A aplicação, ao iniciar, verifica se o banco de dados já foi carregado previamente, e, caso não tenha sido, acrescenta os metadados das classes utilizadas na aplicação. Este processo é feito através do trecho de código apresentado na Figura 3.20, o qual cria o arquivo de dados com 16 megabytes, adiciona as classes mostradas na Figura 3.21 no esquema do banco de dados e cria os índices utilizados nas pesquisas.

```
File propFile = new File("cadastro.config");
boolean newDatabase = !propFile.exists();
instance = new JoodbmsInstanceManagerImpl(propFile);
if (newDatabase) {
 instance.changeState(DatabaseState.WARM);
 instance.addNewDataFile(new File("cadastro.db"), 1024 * 1024 * 16);
 instance.changeState(DatabaseState.ONLINE);
 JoodbmsSession session = instance.open();
 try {
  session.addClass(NotaFiscal.class);
  session.addClass(ItemNota.class);
  session.addClass(Produto.class);
  session.createIndex(NotaFiscal.class.getName(), "numero", new String[] {
"numero" }, true);
  session.createIndex(Produto.class.getName(), "codigo", new String[] {
"codigo" }, true);
 } finally {
  session.close();
 }
} else {
 instance.changeState(DatabaseState.ONLINE);
}
```

Figura 3.20. Código Java para abertura de conexão e configuração do banco de dados

Na Figura 3.23 vê-se também que é possível, utilizando o JOOBMS, armazenar relacionamentos um-para-muitos de forma transparente, utilizando as coleções do Java.

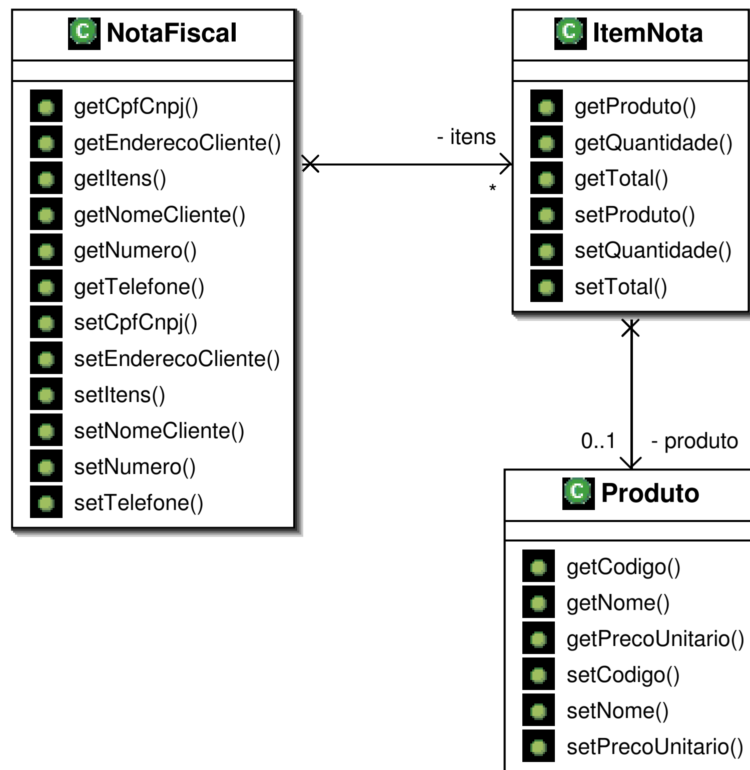



Figura 3.21. Classes persistentes utilizadas no Controle de Notas Fiscais

O passo seguinte é a montagem do formulário mostrado na Figura 3.19. A operação de pesquisa utiliza o número da nota fiscal, e o procedimento é feito através do seguinte fragmento de programa, na Figura 3.22. Este procedimento utiliza os recursos de iteração através de um índice. No caso em questão, durante a criação do banco de dados foi criado o índice **número** sobre o *extent* **example.NotaFiscal**. Este índice é então pesquisado, utilizando como faixa de pesquisa o número da nota fiscal fornecida pelo usuário. É importante observar que a pesquisa pode utilizar o mesmo valor como chave inicial e final, e esta operação irá retornar todos os objetos que contenham como chave o valor informado.

Uma limitação da implementação atual do JOODMBS consiste em não ser possível associar a uma sessão específica um objeto carregado em outra sessão, ou mesmo objetos transientes a uma determinada sessão. Assim, quando é necessário atualizar um objeto persistente a partir de um transiente, é necessário realizar uma cópia dos dados, campo a campo. Caso um dos campos seja referência para outro objeto, é necessário recuperar, a

partir do banco de dados, este objeto persistente e utilizá-lo no lugar do objeto original. Outra característica limitante é não ser possível iniciar os dados dos *proxies* dos objetos já carregados após o fechamento da sessão. Isto implica em ser necessário fazer todas as atualizações da interface com o usuário enquanto a sessão está aberta, ou utilizar padrões de projeto como *Open Session In View* ou *Data Transfer Objects* [11]. Por este motivo, como se vê na Figura 3.22 abaixo, a atualização do formulário da interface com o usuário é feita com a sessão ainda aberta.

```
public static boolean carregaNota(int numero, JPaneNotaFiscal notaFiscal) {
  JoodbmsSession session = instance.open();
  try {
   NotaFiscal nf;
   ObjectIterator<NotaFiscal> iter =
session.iterateRange(NotaFiscal.class.getName(), "numero", new Object[] {
numero },
     new Object[] { numero });
   if (iter.hasNext()) {
    nf = iter.next();
    iter.close();
   } else {
    iter.close();
    return false;
   }
   notaFiscal.mostraFormulario(nf);
   return true;
  } catch (Exception e) {
   e.printStackTrace();
   return false;
  } finally {
   try {
    session.close();
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
 }
}
```

Figura 3.22. Recuperação de uma nota fiscal pelo seu número

O processo de gravação de uma nota fiscal é mais complexo, sendo necessário copiar todos os dados do objeto construído na interface com o usuário. Na Figura 3.23 vê-se que a atualização do objeto precisa recuperar também todos os objetos dependentes, no caso, o os objetos do *extent* **exemplo.Produto**.

```
public static void alteraNota(NotaFiscal novaNota) {
 JoodbmsSession session = instance.open();
 try {
  NotaFiscal nf;
  ObjectIterator<NotaFiscal> iter =
session.iterateRange(NotaFiscal.class.getName(), "numero", new Object[] {
novaNota.getNumero() }, new Object[] { novaNota.getNumero() });
  if (iter.hasNext()) {
   nf = iter.next();
   iter.close();
  } else {
   iter.close();
   nf = session.createObject(NotaFiscal.class.getName());
   nf.setItens((PersistentArrayList<ItemNota>) nf.getJoodbmsSession()
     .createObject(PersistentArrayList.class.getName()));
  }
  nf.setNumero(novaNota.getNumero());
  nf.setEnderecoCliente(novaNota.getEnderecoCliente());
  nf.setNomeCliente(novaNota.getNomeCliente());
  nf.setTelefone(novaNota.getTelefone());
  nf.setCpfCnpj(novaNota.getCpfCnpj());
  if (novaNota.getItens() != null) {
   HashMap<Integer,ItemNota> produtosAntigos=new HashMap<Integer, ItemNota>();
   for (ItemNota item : nf.getItens())
    produtosAntigos.put(item.getProduto().getCodigo(), item);
   for (ItemNota novoItem : novaNota.getItens()) {
    ItemNota item = produtosAntigos.get(novoItem.getProduto().getCodigo());
    if (item != null) {
     item.setQuantidade(novoItem.getQuantidade());
     item.setTotal(novoItem.getTotal());
     produtosAntigos.remove(novoItem.getProduto().getCodigo());
    } else {
     item = session.createObject(ItemNota.class.getName());
     item.setQuantidade(novoItem.getQuantidade());
     item.setTotal(novoItem.getTotal());
     item.setProduto(carregaProduto(session,
novoItem.getProduto().getCodigo()));
     nf.getItens().add(item);
    }
   }
   int i = 0;
   while (i < nf.getItens().size()) {
    ItemNota item = nf.getItens().get(i);
    if (produtosAntigos.containsKey(item.getProduto().getCodigo())) {
     nf.getItens().remove(i);
     session.delete(item);
    } else
     i++;
   }
  } else {
   for (ItemNota item : nf.getItens())
    session.delete(item);
   nf.getItens().clear();
  }
 } catch (Exception e) {
  e.printStackTrace();
 } finally {
  try {
   session.close();
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

Figura 3.23. Criação e modificação de notas ficais

Para a tela de escolha de produtos, mostrada na Figura 3.24, é necessário iterar por todos os elementos do *extent* **exemplo.Produto**. Esta iteração é fácil de ser feita utilizando a classe **ObjectIterator** do JOODBMS. O trecho de código responsável por esta operação está mostrado na Figura 3.25.

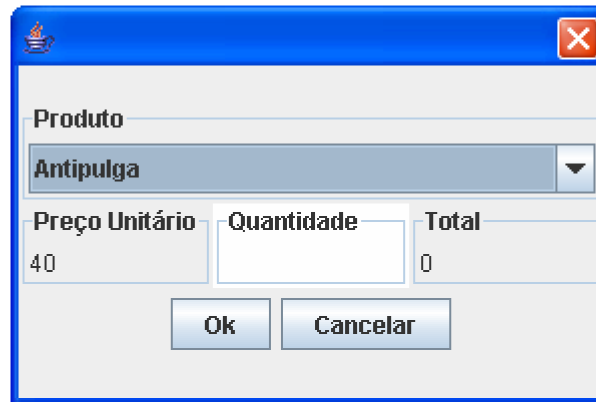



Figura 3.24. Tela de escolha de produto

```
public static void preencheProdutos(ProdutoComboBoxModel produtoModel) {
 JoodbmsSession session = instance.open();
 try {
  ObjectIterator<Produto>iter=session.iterateExtent(Produto.class.getName());
  while (iter.hasNext()) {
   Produto p = iter.next();
   produtoModel.addProduto(p);
  }
  iter.close();
 } catch (Exception e) {
  e.printStackTrace();
 } finally {
  try {
   session.close();
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

Figura 3.25. Obtenção de todos os produtos cadastrados

Através destes exemplos é possível observar que a programação para bancos de objetos é quase totalmente transparente, à exceção, nos exemplos mostrados, na criação de novos objetos e na pesquisa de objetos já cadastrados. Fora estes casos particulares, e a observação de se trabalhar com os objetos persistentes apenas enquanto a sessão estiver aberta, não é necessário utilizar dois “idiomas” de programação – uma linguagem para consultas e outra para a lógica de negócios.

# 4 - CONCLUSÕES E TRABALHOS FUTUROS

Conforme este trabalho demonstra, a área de bancos de objetos apresenta ainda uma área emergente, sendo que grande parte da dificuldade de sua adoção ocorre ainda pela falta de uma padronização e pela quantidade de dados atualmente armazenada em bancos relacionais, hierárquicos ou em rede. A migração para sistemas orientados a objetos pode ocasionar a necessidade de reescrever as aplicações, acrescentando custos maiores do que os potenciais benefícios de seu uso. Além disso, a visão comportamental dos dados pode desfavorecer seu uso para consultas *ad-hoc*.

A estrutura de um banco de objetos permite a manipulação de quantidades relativamente grandes de dados de forma particularmente otimizada, obtendo os objetos diretamente através da navegação em estruturas de dados arbitrariamente complexas, permitindo a navegação de forma eficiente, torna-o uma ferramenta atraente para aplicações científicas e de projeto auxiliado por computador. Também novas pesquisas na área de biotecnologia mostraram grande interesse em bancos de objetos como mecanismo de armazenamento de dados [33] [28].

Existem diversos produtos bancos de objetos disponíveis para uso, mas poucos deles possuem o foco na área acadêmica e científica, sendo mais voltados para uso em sistemas embarcados ou para a pesquisa em focos bastante específicos, como o caso do agrupamento.

Em relação à padronização, recentemente [18], o OMG reiniciou os trabalhos de especificação, baseados no ODMG 3.0, através da formação do *Object Database Technology Working Group* (ODBT WG).

O JOODBMS é um banco de objetos que, devido a sua arquitetura modular, permite a extensão, acrescentando novos elementos ou melhorando os módulos já desenvolvidos. A existência de índices adicionais sobre os atributos facilita a construção de consultas *ad-hoc*, mesmo que programaticamente. A fácil inclusão ou substituição de módulos e serializadores de tipos simples também favorece a extensão do sistema, tornando-o uma boa plataforma para experimentações em persistência de objetos.

Através desta modularização pode-se pensar, como trabalhos futuros, em comparativos de performance entre diversas implementações de índices, serializadores e gerenciadores de armazenamento.

Em relação às outras implementações, vê-se que o JOODBMS apresenta uma boa modularização e documentação de suas interfaces, além de ter por foco a utilização de forma quase transparente para o desenvolvedor, sendo necessário apenas checar os limites da utilização da sessão – a qual deve estar aberta durante todos os instantes em que for necessário utilizar objetos persistentes, e a criação de novos objetos. O foco em criar uma estrutura de página e tupla que permitisse a extensão do esquema e também fácil determinação da posição e valores de atributos sem muito esforço computacional, favorecendo a criação de mecanismos de consulta, de forma a permitir sua utilização como um banco de dados de uso geral.

Entretanto, o JOODBMS ainda precisa de alguns módulos adicionais para ser considerado um banco de objetos capaz de suportar operações em ambiente de produção. A implementação até o momento não suporta transações, bloqueios, nem recuperação em caso de falha, e estas características podem ser adicionadas através da modificação dos módulos correspondentes.

Outras características bastante desejáveis que tornam o processo de utilização do JOODBMS ainda mais transparente para o desenvolvedor, são a coleta de lixo automática, de modo a reduzir a existência de métodos *delete* no código, a utilização de mapas de referência para os objetos carregados na sessão, de modo a gravar os objetos persistentes em disco quando o uso de memória chegar a determinado limite. Outra característica bastante desejável é permitir a utilização de *interfaces* nos objetos persistentes, uma vez que o gerenciador de metadados ainda não é capaz de, ao adicionar um objeto, determinar quais classes implementam interfaces específicas, e utilizar estas interfaces na declaração de atributos persistentes (como, por exemplo, *java.util.List*).

# 5 - REFERÊNCIAS BIBLIOGRÁFICAS


[1] Atkinson. M, DeWitt. D, Maier D. The Object-Oriented Database System Manifesto.

[2] Bayer, R e McCreight, E. Organization and Maintenance of Large Ordered Indexes.

[3] Chaudhri, A. B e Zicari, R. Succeeding with Object Databases. John Wiley & Sons. Estados Unidos, 2001.

[4] Cook, J. E, Wolf, A. L. e Zorn, B. G. A Highly Effective Partition Selection Policy for Object Database Garbage Collection. IEEE Transactions on Knowledge and Data Engineering 10 (1), p. 153, 1998

[5] Cook, W. R. e Rosemberger, C. Native Queries for Persistent Objects – A Design White Paper

[6] Date, C. J. Introdução a sistema de banco de dados. Campus, Rio de Janeiro, 1991

[7] Deitel, H.M e Deitel, P.J. Java como programar. 4ª. Ed. Bookman, Porto Alegre, 2003.

[8] Elmasri, R e Navathe, S,B. Sistemas de Banco de Dados. 4ª .Ed. São Paulo: Pearson Addison Wesley, 2005

[9] Folk, M, J e Zoellick, B. File Structures. 2ª.Ed, Addison-Wesley, United States, 1992

[10] Folk, M. J e Zoellick, B. File Structures. 2ª.Ed, Addison-Wesley, United States, 1992

[11] Fowler, Martin. Patterns Of Enterprise Application Architecture. Addison-Wesley Professional, 2002.

[12] Garcia-Molina, H, Ullman, J. D e Windom J. Implementação de sistemas de banco de dados. Campus, Rio de Janeiro, 2001

[13] http://db.apache.org/ojb/, acessado em 19 de julho de 2006.

[14] http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14220/toc.htm, acessado em 19 de julho de 2006.

[15] http://en.wikipedia.org/wiki/C%2B%2B, acessado em 19 de julho de 2006

[16] http://en.wikipedia.org/wiki/Delphi_programming_language, acessado em 19 de julho de 2006

[17] http://en.wikipedia.org/wiki/M_technology, acessado em 25/6/2006

[18] http://en.wikipedia.org/wiki/Object_database, acessado em 25/6/2006

[19] http://en.wikipedia.org/wiki/Sort_algorithm, acessado em 04/07/2006

[20] http://java.sun.com, acessado em 10/11/2005

[21] http://java.sun.com, acessado em 10/11/2005

[22] http://java.sun.com/products/ejb/, acessado em 19 de julho de 2006.

[23] http://java.sun.com/products/jdo/, acessado em 19 de julho de 2006.

[24] http://java.sun.com/products/jfc/, acessado em 19 de julho de 2006

[25] http://msdn.microsoft.com/netframework/, acessado em 19 de julho de 2006.

[26] http://www.dbo4.com, acessado em 19 de julho de 2006.

[27] http://www.ecma—international.org/publications/standards/Ecma-334.htm, acessado em 26/04/2006

[28] http://www.eyedb.org/history.php, acessado em 25/6/2006

[29] http://www.hibernate.org , acessado em 10/11/2006

[30] http://www.intersystems.com/cache/, acessado em 19 de julho de 2006.

[31] http://www.junit.org/index.htm, acessado em 19 de julho de 2006

[32] http://www.nhibernate.org, acessado em 13/11/2006

[33] http://www.objectivity.com/WhitePapers/BioinformaticsWhitePaper.pdf, acessado em 25/6/2006

[34] http://www.ozone-db.org, acessado em 19 de julho de 2006.

[35] http://www.postgresql.org, acessado em 19 de julho de 2006.

[36] http://www.solarmetric.com, acessado em 3 de novembro de 2005.

[37] http://www.training.com.br/lpmaia/pub_prog_oo.htm, acessado em 19 de julho de 2006.

[38] Hurson.R, Pakzad. H e Cheng.J. Evolution and Performance Issues

[39] McCreight, E, M. Pagination of B*-Trees with Variable-Length Records. Communications of the ACM 20 (9), pp. 670, 1977.

[40] Programação Orientada a Objetos – Uma Abordagem com Java, http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poojava.pdf, acessado em 19 de julho de 2006.

[41] Ra, Young-Gook e Rundensteiner, Elke A. A Transparent Schema-Evolution System Based on Object-Oriented View Technology. IEEE Transactions on Knowledge and Data Engineering 9 (4), p. 600, 1997

[42] Silberschatz, A., Korth, H. F e Sudarshan, S. Sistema de Bancos de Dados. Makron Books, São Paulo, 1999

[43] The Committee for Advanced SGBD Function. Third Generation Data Base System Manifesto. U.C. Berkeley Memorandum No. UCB/ERL M90/28, April 9, 1990.