UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E INFORMÁTICA

"NÚCLEO GERENCIADOR DE OBJETOS COMPATIBILIZANDO EFICIÊNCIA E FLEXIBILIDADE"

CARLOS ROBERTO VALÊNCIO

Tese apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para obtenção do título de Doutor em Ciências "Física Aplicada - Opção Física Computacional"

USP/IFSC/SBI

8-2-001360

Orientador:
Prof. Dr. Caetano Traina Júnior

Comissão Julgadora:
Prof. Dr. Caetano Traina Júnior (ICMC-USP)
Profa. Dra. Rosely Sanches (ICMC-USP)
Prof. Dr. Gonzalo Travieso (IFSC-USP)
Prof. Dr. Mauro Biajiz (UFSCar)
Prof. Dr. Marcelo Finger (IME-USP)

São Carlos - São Paulo
2000

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

Valêncio, Carlos Roberto

"Núcleo gerenciador de objetos compatibilizando eficiência e flexibilidade"/ Carlos Roberto Valêncio – São Carlos, 2000

Tese (Doutorado) – Instituto de Física de São Carlos da Universidade de São Paulo, 2000 – páginas: 143
Área: Física Aplicada-Opção Física Computacional
Prof. Dr. Caetano Traina Júnior

1. Eficiência e flexibilidade, 2. Gerenciamento de dados, 3. Objetos

I. Título

UNIVERSIDADE DE SÃO PAULO
Instituto de Física de São Carlos

Av. Dr. Carlos Botelho, 1465
CEP 13560-250 - São Carlos - SP
Brasil

Fone (016) 273-9333
Fax (016) 272-2218

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE CARLOS ROBERTO VALÊNCIO APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, DA UNIVERSIDADE DE SÃO PAULO, EM 06 DE SETEMBRO DE 2000.

COMISSÃO JULGADORA:

Prof. Dr. Caetano Traina Júnior/ICMC-SCE-USP

Profa. Dra. Rosely Sanches/ICMC-SCE-USP

Prof. Dr. Gonzalo Travieso/IFSC-FFI-USP

Prof. Dr. Mauro Biajiz/UFSCar

Prof. Dr. Marcelo Finger/IME-USP

AGRADECIMENTOS

Ao Prof. Dr. Caetano Traina Júnior, orientador e amigo, pela competência, dedicação e compreensão dispensada durante o desenvolvimento desse trabalho. Trabalhar com o Prof. Caetano durante esses anos todos só tem feito aumentar minha admiração à sua capacidade profissional e ao seu caráter.

Ao Prof. Dr. Eurípedes Alves da Silva (Ibilce-Unesp), que desde à graduação sempre apoiou e incentivou na decisão de enveredar pela carreira acadêmica.

À Prof. Dra. Agma Juci M. Traina, pela simpatia que sempre nos tratou.

À secretária Wladerez, do IFSC, pela eficiência e simpatia com que sempre nos atendeu.

Aos amigos do Grupo de Base de Dados e Imagem - ICMC-USP: Mauro Biajiz, João Eduardo Ferreira, Roberto Figueira dos Santos Filho, Rildo Ribeiro dos Santos, pelo companheirismo e troca de conhecimento nas nossas discussões. Em especial a Roberto Figueira que contribuiu mais diretamente nesse trabalho.

Aos Departamentos de Computação Unesp-Bauru e Departamento de Ciência da Computação e Estatística do Ibilce-Unesp, que facilitaram a realização desse trabalho.

Aos meus pais, José e Theodora, que sempre incentivaram e deram o apoio necessário para concluir mais esta etapa da minha vida.

À minha esposa Soraia, pela abnegação e carinho oferecidos quando foi preciso para que esse trabalho se realizasse.

Aos meus filhos Lucas e Yúri que hão de perdoar-me pela ausência.

Ao Prof. Luís Roberto Benatti, pela valiosa ajuda na correção do português.

Ao CNPq e CAPES, pelo apoio financeiro dado para o desenvolvimento desse trabalho.

Dedico esse trabalho a
minha esposa e companheira Soraia,
aos meus filhos, Lucas e Yúri
e aos meus pais, José e Theodora
que sempre estiveram ao meu lado.

ÍNDICE

Lista de Figuras vi

Lista de Tabelas vii

Lista de Siglas viii

Resumo x

Abstract xi

Capítulo 1 INTRODUÇÃO

1.1 Considerações Iniciais 1
1.2 Motivação e Escopo 1
1.3 Objetivos e metodologias 3
1.4 Histórico tecnológico 4
1.5 Organização da tese 7

Capítulo 2 FUNDAMENTAÇÃO CONCEITUAL

2.1 Considerações iniciais 8
2.2 Modelos de dados 9
2.2.1 Modelo relacional 10
2.2.1.1 Domínio 10
2.2.1.2 Chaves 11
2.2.1.3 Integridade de entidade 12
2.2.1.4 Integridade referencial 12
2.2.2 Modelos semânticos de dados 12
2.2.3 Modelos orientados a objetos 13
2.2.3.1 Tipos 14
2.2.3.2 Hierarquia de tipos 15
2.2.3.3 Herança 15
2.2.3.4 Extensões de tipos 16
2.2.3.5 Chaves 16

2.2.3.6 Objetos 16
2.2.3.7 Objetos-coleção 17
2.2.3.8 Objetos atômicos 18
2.2.3.9 Literais 18
2.2.3.10 Objetos estruturados 19
2.2.3.11 Atributos 19
2.2.3.12 Relacionamentos 20
2.3 Identificadores de objetos 22
2.4 *Clustering* 24
2,5 *Swizzling* 26
2.6 Gerenciamento de *buffer* 26
2.7 Gerenciamento de transação 27
2.7.1 Transações longas 29
2.7.2 Transações aninhadas 30
2.7.3 Transações cooperativas 30
2.8 Controle de concorrência 30
2.8.1 Protocolo baseado em bloqueio 31
2.8.1.1 Bloqueio de duas fases 32
2.8.1.2 Bloqueio baseado em grafos 33
2.8.2 Bloqueio baseado em *timestamp* 33
2.8.3 Bloqueio com validação 34
2.8.4 Granularidade múltipla 34
2.8.5 Esquemas multiversão 37
2.9 Recuperação 38
2.9.1 Recuperação baseada em *log* 38
2.9.2 Páginas *shadow* 39
2.9.3 WAL 39
2.10 Indexação de objetos 40
2.11Considerações finais 44

Capítulo 3 GERENCIADORES ORIENTADOS A OBJETOS

3.1 Considerações iniciais 45
3.2 ORION/ITASCA 46
3.2.1 Arquitetura 46
3.2.2 Gerenciador de *buffers* 47
3.2.3 Identificador de objetos 48
3.2.4 Objetos compostos 49
3.2.5 Armazenamento e páginas de objetos 49
3.2.6 Armazenamento de classes de objetos 50
3.2.7 *Clustering* 50
3.2.8 Versões e notificação de alterações 51
3.2.9 Evolução de esquemas 52

3.2.10 Indexação 52
3.2.11 Gerenciamento de transações 53
3.2.12 Controle de concorrência 53
3.2.13 Recuperação 54
3.3 O Sistema O2 55
3.3.1 Arquitetura 55
3.3.2 O_2Engine 56
3.3.3 Backup e recuperação 62
3.3.4 Mecanismo de replicação 62
3.3.5 Coleta de lixo 62
3.3.6 Indexação 63
3.3.7 Gerenciador de *clustering* 63
3.4 IRIS 64
3.4.1 Interfaces de consultas e ambientes de programação 65
3.4.2 Gerenciamento de objetos IRIS 65
3.4.3 Gerenciamento de memória IRIS 66
3.4.3.1 Gerenciamento de transação 66
3.4.3.2 Controle de versões e controle de concorrência 66
3.5 GemStone 67
3.5.1 Arquitetura 67
3.5.2 Recuperação e *backup* 67
3.5.3 Gerenciamento de transações e controle de concorrência 68
3.5.4 Objetos e indentificador de objetos 68
3.5.5 Outras características do GemStone 69
3.6 Object Store 69
3.6.1 Arquitetura 69
3.6.2 Endereçamento 70
3.6.3 Controle de concorrência 70
3.6.4 Recuperação 72
3.6.5 Atributos do tipo coleção 72
3.7 Outros 72
3.8 Considerações finais 72

Capítulo 4 O NÚCLEO GERENCIADOR DE OBJETOS

4.1 Considerações iniciais 74
4.2 O Modelo de dados SIRIUS 75
4.3 Arquitetura do sistema SIRIUS e o suporte do núcleo para aos aspectos semânticos 78
4.3.1 Subsistema de gerenciamento de atributos 79
4.3.1.1 Suporte a armazenagem de tipos de objetos 80
4.3.1.2 Suporte para atributos 81
4.3.1.3 Suporte para a composição de objetos 83

4.3.2 Subsistema de gerenciamento de registros lógicos 83
4.3.3 Subsistema de gerenciamento de registros reais 88
4.3.3.1 Gerenciamento de registros físicos 88
4.3.3.2 Gerenciamento de transação 89
4.3.3.3 Recuperação de endereços para a armazenagem física 92
4.3.3.4 Gerenciamento de *buffers* 92
4.3.3.5 Bloqueio e controle de concorrência 93
4.3.3.6 Recuperação 94
4.4 Esquema de gerenciamento de OIds no gerenciador SIRIUS 96
4.5 Considerações finais 97

Capítulo 5 ASPECTOS DA IMPLEMENTAÇÃO

5.1 Considerações finais 99
5.2 Criação da base de dados 100
5.3 O Núcleo gerenciador de objetos 102
5.3.1 Subsistema de gerenciamento de atributos 103
5.3.1.1 Primitivas do módulo SG-Atrib-Composição 104
5.3.1.2 Primitivas do módulo SG-Atrib-Lista 104
5.3.1.3 Primitivas do módulo SG-Atrib-Tupla 105
5.3.2 O subsistema de gerenciamento de registros lógicos 106
5.3.2.1 Proposta para uso de apenas um tipo de bloco lógico 107
5.3.2.2 Proposta com o uso de dois tipos de blocos lógicos 109
5.3.2.3 Testes 112
5.3.3 Subsistema de gerenciamento de registros reais 113
5.3.3.1 As primitivas do módulo SG-Real 115
5.3.3.2 O módulo transação 118
5.3.3.3 O módulo base 119
5.4 Conclusões sobre a implementação 120
5.4.1 Armazenagem eficiente e flexível para atributos 120
5.4.2 Gerenciamento de Oids 122
5.4.3 Esquema de *clustering* 123
5.4.4 Armazenagem de atributos de tamanho variáveis nas páginas 124
5.4.5 Uso de páginas *shadow* no suporte a recuperação e gerenciamento de transação 125
5.5 Considerações finais 127

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

Capítulo 6 CONCLUSÕES E FUTURAS PESQUISAS

6.1 Considerações Iniciais 128
6.2 Histórico do Trabalho 128
6.3 Contribuições Inovadoras 129
6.3.1 Suporte nativo às abstrações de generalização, classificação e composição 129
6.3.2 Armazenagem de atributos para objetos 130
6.3.3 Gerenciamento de Oids 130
6.3.4 Esquema de *clustering* 131
6.3.5 Armazenamento de atributos de tramanho variáveis nas páginas ... 131
6.3.6 Uso de páginas *shadow* no suporte a recuperação e gerenciamento de transação 131
6.3.7 O NuGO como um gerenciador de dados para qualquer modelo de dados 132
6.4 Futuras pesquisas 132
6.4.1 Ampliação da capacidade de referenciação de registros físicos na base de dados através do diretório de páginas *shadow* - TCReFO 132
6.4.2 Esquema de recuperação 133
6.4.3 Indexação e *data mining* em base de dados multimídia 133
6.4.4 Elaboração e aplicação de um *benchmark* para o NuGO 134
6.4.5 Estratégias para otimização de *clustering* físico 134
6.4.6 Outras propostas 135
6.5 Conclusões gerais 135

Referências Bibliográficas 137

LISTA DE FIGURAS

Figura 2.1: Exemplo com as relações: Empregado, Trabalho e Projeto 11
Figura 2.2: Matriz de compatibilidade de bloqueios 32
Figura 2.3: Hierarquia de granularidades para ilustrar múltiplos níveis de bloqueio 35
Figura 2.4: Matriz de compatibilidade para múltiplas granularidades de bloqueio 36
Figura 3.1a: Componentes funcionais dos sistemas ORION 46
Figura 3.1b: Arquitetura do sistema ORION-1SX 47
Figura 3.2: Estruturas de suporte à localização de objetos nos *buffers* 48
Figura 3.3: Formato de armazenamento de objetos no disco 49
Figura 3.4: Formatação para uma página nos sistemas ORION 50
Figura 3.5: Estados possíveis para as versões de objetos no ORION 51
Figura 3.6: Arquitetura do sistema gerenciador de base de dados O_2 56
Figura 3.7: Arquitetura do O_2Engine 57
Figura 3.8: Arquitetura cliente/servidor do O_2Store 58
Figura 3.9: Contendo a movimentação (deslocamento) dum registro nas páginas 59
Figura 3.10: Estrutura para a alocação de objetos transientes no O_2 60
Figura 3.11: Um exemplo de *clustering* 64
Figura 3.12: Arquitetura cliente/servidor do ObjectStore 71
Figura 4.1: Estrutura em camadas do Gerenciador SIRIUS, com realce para o seu núcleo 78
Figura 4.2: Estrutura de listas e tuplas para atributos de objetos 82
Figura 4.3: Blocos lógicos com registros de tamanho variável 86
Figura 4.4: Camada do SG-Log2 com blocos lógicos, registros virtuais e registros de tamanho fixo 87
Figura 4.5: Estrutura e funcionalidades suportadas pelo SG-Real 89
Figura 5.1: Uma base de dados SIRIUS com os registros de controle no início 102
Figura 5.2: Blocos lógicos com respectivos tipos lógicos agrupados por colônias e objetos 107
Figura 5.3: Estrutura de controle externo às colônias 108
Figura 5.4: Estrutura de controle interno às colônias 110
Figura 5.5: Atualização do valor de controle de Oid em registros lógicos do tipo objeto . 111
Figura 5.6: Gráfico que descreve a quantidade relativa de espaço não utilizado na base de dados 113
Figura 5.7: Gráfico que descreve a quantidade relativa de espaço usado para as estruturas de controle nos blocos 113
Figura 5.8: Uma seqüencia de campos na forma utilizada pelos atributos com característica de listas 120

LISTA DE TABELAS

Tabela 5.1: Apresentação das informações componentes do ReCoB 101
Tabela 5.2: Exemplos de tipos básicos em diferentes compiladores 103
Tabela 5.3: Valores de entrada para os testes aplicados no SG-Log1 e SG-Log2 112
Tabela 5.4: Proporção entre o tamanho de uma página e o tamanho máximo da base de dados em disco, baseado em endereços de 4 bytes ... 114
Tabela 5.5: Relação entre o tamanhode uma página, tamanho da estrutura TCReFO em memória e tamanho máximo em disco referenciado por essa estrutura 115
Tabela 5.6: Relação entre tamanho de uma página, o tamanho do TCReFO e a capacidade do TCReFO em referenciar informações em disco baseado no seu tamanho na memória 126

LISTA DE SIGLAS

ACID - Atomicidade, Consistência, Isolamento e Durabilidade
API - Aplication Progamation Interface
BLOId - BLOb Identifier
CAD - Computer-Aided Design
CAM - Computer-Aided Manufacturing
CASE - Computer Aided Software Enginering
CIM - Computer-Integrated Manufaturing
CODASYL DBTG - Conference on Data Systems Languages Database Task Group
CSOID - Contador Salva Oids
DBA - Database Administrator
DBMS - Database Management System
DER - Diagrama Entidade Relacionamento
EFTIRR - Endereços de Fragmentos da Tabela de Indexação de Registros Reais
FTIRR - Fragmento da Tabela de Indexação de Registros Reais
GEO - Gerenciador de Objetos
LDD - Linguagem de Definição de Dados
LFU - Least Frequently Used
LMD - Linguagem de Manipulação de Dados
LOId - Logical Object Identification
LRU - Least Recently Used
ME-R - Modelo Entidade-Relacionamento
ME-RE - Modelo Entidade-Relacionamento Estendido
MRO - Modelo de Representação de Objetos
MRU -Most Recently Used
NuGO - Núcleo Gerenciador de Objetos
ODL - Object Definition Language
ODMG - Object Database Management Group
OId - Object Identifier
OLAP - On-Line Analytic Processing
OLTP - On-Line Transaction Processing
OQL - Object Query Language
PID - Persistent Identifier
QBE - Query By Exemple
RaaB - Registro de armazenamento do estado atual e anterior da Base de dados
ReCoB - Registro de Configuração da Base de dados
RegFis - Registro Físico
RegReal - Registro Real
RID - Record Identifier

ROD - Resident Object Descriptor
ROT - Resident Objects Table
SG-Atrib - Subsistema de Gerenciamento de Atributos
SGBD - Sistema de Gerenciamento de Banco de Dados
SGBDOO - Sistema de Gerenciamento de Banco de Dados Orientado a Objetos
SG-Log - Subsistema de Gerenciamento de Registros Lógicos
SGML - Standart Generalized Markup Language
SG-Real - Subsistema de Gerenciamento de Registros Reais
SQL - Structured Query Language
Tambuf - Tamanho de buffer
TCReFO - Tabela de Conversão de Registros Reais para Físicos e Overflow
UID - Unic IDentifier logic
XML - eXtensible Markup Language

RESUMO

A tecnologia de construção de Sistemas de Gerenciamento de Base de Dados tradicionais, em particular os que suportam o modelo relacional, tem atendido às exigências das aplicações na área de negócios ao longo dos anos. No entanto, as aplicações mais complexas, tais como projetos de Engenharia e manufatura (CAD/CAM e CIM[1]), experimentos científicos, suporte à Medicina, telecomunicações, sistemas de informação geográfica e sistemas multimídia, têm exigido recursos mais sofisticados para os quais esses sistemas gerenciadores de base de dados não foram concebidos e, portanto, têm encontrado dificuldades em propiciar suporte à tais aplicações.

Com o objetivo de proporcionar novas tecnologias para o apoio ao gerenciamento de dados não convencionais, esse trabalho apresenta a implementação de um Núcleo Gerenciador de Objetos, onde é oferecido um conjunto de recursos para um suporte flexível e eficiente às atividades de gerenciamento de dados. Esse gerenciador é apresentado no contexto de um modelo de dados orientado a objetos, porém a maioria dos conceitos e soluções apresentadas podem ser aproveitadas em outros tipos de gerenciadores de dados, independentemente do modelo de dados suportado. Em particular, o núcleo pode ser utilizado tanto para a implementação de um gerenciador de dados orientados a objetos quanto para um gerenciador relacional. O núcleo prevê também sua utilização como gerenciadores de dados que suportem a manipulação de dados fracamente estruturados (ou semi-estruturados) e como gerenciador de documentos multimídia em aplicações centrada na Web. A implementação do Núcleo Gerenciador de Objetos foi executada de maneira modular numa arquitetura em camadas, que delimitam a implementação das funcionalidades oferecidas. Apesar da forte integração entre as diversas camadas, necessária para a operação eficiente dos gerenciadores de dados, essa arquitetura provê uma definição precisa dos diversos módulos, permitindo que mais de uma alternativa de implementação possa ser avaliada para cada módulo, tornando o sistema resultante altamente configurável.

Dentre os principais tópicos para os quais esse trabalho apresenta contribuições inovadoras encontram-se: gerenciamento de identificadores de objetos (Oids); gerenciamento de transação e controle de concorrência baseado na semântica da aplicação; otimização de acessos ao disco para execução de operações de modificação de registros dentro de transações; uso de estruturas de tuplas e listas concorrentemente, para agilizar o acesso e a definição dos atributos de objetos; e a manutenção de esquemas e dados integrados numa mesma estrutura.

---

[1]CAD/CAM e CIM: *Computer-Aided Design/Computer-Aided Manufacturing e Computer-Integrated Manufaturing* - Sistemas de Projeto/Manufatura Apoiados por Computador e Manufatura Integrada por Computador.

ABSTRACT

The technologies developed to build the current generation of Database Management Systems - DBMS, like those based on the relational data model have been enough to support the needs of traditional business application. However, more demanding applications, like computer aided design and manufacturing (CAD/CAM e CIM), scientific data retrieval and analysis, computer aided medical systems, telecommunications, geographical information systems and multimedia systems have not been adequately supported.

The objective of this work is to develop new technologies to build DBMSs that support those non-conventional applications. To this, we implemented the kernel of an object manager, incorporating a representative set of tools to provide a flexible and efficient support to key operations of DBMSs. This kernel is described using an object oriented model manager as the target model. However almost all new techniques developed are worth to be used in data managers supporting other data models. In particular, we show that this kernel can be used to build Object Oriented DBMS and Relational DBMSs. The kernel supports also the construction of DBMSs that maintain loose-structured data (or semi-structured data), providing a good starting point to build web-based applications that handle multimedia documents.

The kernel was implemented in a modular, multi-level architecture. Each module provides a well-defined service, and has a well defined interface, so it is possible to have more than one implementation of each module, enabling the comparison or tunning of the kernel for specific situations. Nonetheless, this structure enforces a tight integration of the modules to enable an efficient execution of the resulting DBMS.

The main contributions of this work include new techniques to improve the following aspects of database managers: Object Identifiers management - OIds; transaction and concurrency control based on the application and data semantic; disk accesses optimisation to maintain shadow pages during transaction execution; use of tuples and list in the attribute retrieval and define structures; and storing integrated of schema and data in a common structure.

Capítulo 1

INTRODUÇÃO

1.1 Considerações gerais

A história do Gerenciamento dos Dados está vinculada às exigências apresentadas pelas aplicações mais relevantes nos respectivos períodos de sua evolução. No princípio, o gerenciamento da informação dava-se de forma manual, com o uso de tablóides, nos quais se registravam informações sobre a descrição de propriedades e taxas - considerados como marco zero da história do processamento da informação os tablóides encontrados na Suméria datados de 4000 a.C. [Gra_96]. Atualmente, com intuito de suportar as novas aplicações, as informações estão disponíveis através dos computadores nas mais variadas formas: documentos, dados científicos, sons, imagens, vídeos, planilhas, gráficos etc. Muito esforço tem sido concentrado para armazenar, recuperar, visualizar e analisar essas informações. Tais tarefas têm sido sistematicamente atribuídas aos gerenciadores de dados que, com o passar do tempo, têm evoluído para suportar o tratamento das mais sofisticadas formas de informação.

1.2 Motivação e escopo

A sofisticação das aplicações tem determinado as tendências nas pesquisas em sistemas gerenciadores de banco de dados [Sil_96b] [Gra_96]. Nos anos 90, o setor de telecomunicações e internet proporcionou avanços consideráveis provocadores do fenômeno crescente da disseminação da informação, também denominado "a explosão na informação digitalizada"[Kor_97], contribuindo de modo significativo para o

aumento da complexidade exigida pelas aplicações que se utilizam dessa infraestrutura. Exemplos de aplicações que têm direcionado a capacidade dos SGBDs são descritos a seguir para que se possa entender a agenda das pesquisas de banco de dados nesses últimos anos[Sil_96a][Sil_96b]:

- O sistema EOS/DIS – *Earth Observation System/ Data Information System*, construído pela NASA, é composto por uma base de dados alimentada por um sensor remoto de dados, cuja previsão é crescer 5 terabytes por dia (um milhão de megabytes). Calcula-se que por volta de 2007, essa massa de dados deva atingir 15 petabytes. Portanto, o suporte ao EOS/DIS deve contemplar os avanços no armazenamento de dados, gerenciamento, busca e visualização. Os dados armazenados contêm características espaciais e temporais, além de a maioria desses dados terem características científicas;

- Sistemas de suporte ao comércio eletrônico, no qual estão envolvidos grande número de participantes na rede (fornecedores e compradores) consumindo um suporte específico de gerenciadores que exigem as seguintes características: integração de informações heterogêneas, necessidade de confiabilidade na distribuição e transferência de dados;

- Sistemas de informação *health-care,* nos quais os clínicos lidam com diferentes tipos de informação durante o curso de seu trabalho. Nesses sistemas os desafios encontram-se na integração de informações heterogêneas e legadas, controle de acesso preservadores da confidencialidade dos registros médicos, e interfaces apropriadas para os profissionais da saúde;

- Publicações digitais – hoje é possível armazenar livros, artigos, trabalhos científicos, vídeos, imagens e outros elementos que podem ser transportados eletronicamente e entregues rapidamente a consumidores ligados em redes no mundo todo. Esses sistemas impõem alguns desafios que devem ser encarados pela comunidade científica de banco de dados, tais como o gerenciamento de grandes volumes de informação num tempo aceitável ou até mesmo com restrições de tempo-real; proteção de propriedade intelectual, incluído o pagamento de direitos autorais e a restrição à revenda de informações;

- Projetos cooperativos – grandes projetos, como, por exemplo, aeronaves estão sendo construídas e mantidas por consórcios de empresas independentes. As informações desses projetos têm um ciclo de vida que pode durar décadas, incluídas as atividades como modificações e manutenção. Projetos cooperativos exigem novas formas de controle de concorrência e mecanismos de compartilhamento de informações na base de dados. Interação de coleções de processos, tais como projeto e simulação, exigem o gerenciamento de *workflow*, no qual as transações longas interagem de modo seguro. Também deverá ser suprido o suporte para versões de projetos tanto para um único componente quanto o suporte para configurações que combinem versões de muitos componentes.

A geração dos SGBDs convencionais tem-se mostrado inadequada no sentido de oferecer as abstrações necessárias às aplicações avançadas e emergentes, e que permitam agir como repositório para aplicações integradas[Kho_94]. Um conjunto de carências surge ao suportar essas aplicações mais sofisticadas, por exemplo, a dificuldade em representar objetos complexos; ausência de mecanismos como o gerenciamento de configurações; disponibilidade de apenas tipos atômicos(inteiros, caracteres etc.); ausência de conceitos semânticos como generalização e agregação; modelos de transação inadequado; suporte a dados pouco estruturados; falta de suporte a dados cuja definição acompanha os próprios dados (meta-dados); e outros [Kim_92].

Com o objetivo de suprir essas carências, a geração de SGBDs, que se segue aos convencionais, tem sido apresentada como solução ao suporte a tais aplicações. Esses SGBDs, apoiados em modelos baseados no paradigma de orientação a objetos[Atk_89][Kim_92], surgiram como as primeiras tentativas de proporcionar um modelo de dados extensível. As abstrações de dados são utilizadas para conduzir os usuários na construção dos próprios tipos específicos da aplicação, os quais devem ser então gerenciados pelo SGBD. O uso de objetos também tem-se mostrado eficiente na definição de bases de dados heterogêneas, garantir a interoperabilidade entre objetos de aplicações integradas e permitir a modularidade dos próprios SGBDs. Os módulos de SGBDs podem ser descritos segundo termos provenientes da área de orientação a objetos, tornando-os expansíveis e fáceis de exportar para outros sistemas [Sil_96b].

Nos últimos anos, várias companhias fabricantes de SGBDs orientados a objeto têm emergido e, mais recentemente, um comitê composto de vendedores representativos e pesquisadores tem-se reunido com o objetivo de produzir um padrão para estes SGBDs. Desses, o que se encontra em estágio mais avançado é o padrão ODS: ODMG 2.0 - *Object Database Standard: Object Database Management Group* [Cat_97].

1.3 Objetivos e metodologias

O Grupo de Banco de Dados e Imagens que atua junto ao ICMC - USP - São Carlos vem trabalhando, como uma de suas principais atividades, no desenvolvimento de tecnologias para a construção de sistemas gerenciadores de banco de dados orientados a objeto. Um dos primeiros trabalhos desse grupo foi o Modelo de Representação de Objetos - MRO [Tra_86] [Tra_88] [Tra_92] que serviu de base para a construção do GErenciador de Objetos - GEO [Tra_91] [Tak_93], um sistema de gerenciamento de banco de dados para suportar aplicações não convencionais, como: ambientes de desenvolvimento de projetos de software; aplicativos de suporte a projetos e fabricação auxiliados por computador (CAD/CAM); e suporte às aplicações científicas e estatísticas. Posteriormente, foi proposto um modelo de dados orientado a objeto denominado SIRIUS [Bia_96], o qual suporta as abstrações de classificação, associação (composição e agregação) e generalização.

O objetivo desse trabalho é propor uma arquitetura e a respectiva implementação para um Núcleo Gerenciador de Objetos - NuGO. O Núcleo descrito no presente trabalho foi projetado inicialmente para suportar o modelo de dados SIRIUS e permitir-

lhe a utilização não apenas em gerenciadores orientados a objetos, mas também como "motor" em sistemas de gerenciamento de banco de dados convencionais, por exemplo, os sistemas relacionais. As funcionalidades e tecnologias pertinentes a um SGBD e suportadas pelo NuGO são o gerenciamento de transação, gerenciamento de *buffers*, controle de concorrência, bloqueio, recuperação, *clustering*, gerenciamento de OIds, suporte à composição de objetos através de tuplas, atributos de tamanho variáveis e fixos. No NuGO, essas funcionalidades são implementadas de modo a oferecer o suporte exigido aos SGBDs de forma flexível e eficiente. Esse trabalho também teve o objetivo de funcionar como bancada para a experimentação de novas tecnologias no âmbito do desenvolvimento de gerenciadores de banco de dados que suportam a nova geração de aplicações.

Adicionalmente, o trabalho apresenta alternativas de implementação para uma das camadas do NuGO (no capítulo 4 e 5), analisando o desempenho de cada uma das possibilidades (no capitulo 5).

1.4 Histórico tecnológico

A automatização do gerenciamento de dados suporta um conjunto de funcionalidades oferecidas para facilitar e tornar segura a manipulação de grandes quantidades de dados. Os primeiros sistemas de gerenciamento de dados surgiram no final dos anos 60, quando a IBM lançou a versão do IMS, um gerenciador de banco de dados baseado no modelo hierárquico. Apesar das restrições iniciais desse modelo, o mercado absorveu o produto, o qual trouxe consigo a inovação de que as estruturas das informações manipuladas pelas diferentes aplicações devem ser gerenciadas independentemente da estrutura da própria aplicação. Deste modo, os dados não necessitam mais ser duplicados em diferentes arquivos, as bases de dados podem ser compartilhadas pelas várias aplicações, as informações podem ser obtidas através de linguagens de consulta, esquemas de segurança são oferecidos de modo a restringir o acesso às informações, além de outros recursos [Sil_96b]. Nesse mesmo período, final dos anos 60, foi escrita a primeira especificação padrão de banco de dados denominada relatório CODASYL DBTG - *Conference on Data Systems Languages Database Task Group*, cuja data da sua publicação oficial é o ano de 1971 [Sil_99]. O Modelo DBTG CODASYL, considerado sucessor do modelo hierárquico e denominado modelo de redes, está baseado numa coleção de registros arranjados numa estrutura de grafo dirigido. Embora apresente capacidade de representação semântica maior que o modelo hierárquico, esse modelo ainda restringe a construção das aplicações aos algoritmos complexos necessários para implementá-los e, assim como no hierárquico, a adição de novas estruturas lógicas ainda causa a possibilidade de ser necessário reescrever as aplicações existentes [Sil_96a].

Em 1970, E.F. Codd publicou um artigo [Cod_70] no qual sugeriu que os dados devem ser gerenciados em um nível conceitual superior, representados através de estruturas de dados simples baseadas em relações matemáticas e que as consultas possam ser efetuadas por meio de linguagem declarativa (não procedural) de alto nível. A proposta inicial feita por Codd embasou a construção dos Sistemas Gerenciadores de Banco de Dados Relacionais e direcionou as pesquisas durante a década de 70, na qual os temas dominantes foram

[Sto_89] [Sil_96a]: padronização de uma linguagem de consulta relacional de alto nível; desenvolvimento de algoritmos para a otimização de consultas; formulação da teoria de normalização; definição de esquemas de *clustering* para tuplas de relações; construção de gerenciadores de *buffers*; técnicas de indexação para acessos aos registros; e a implementação de protótipos que acabaram constituindo-se em embriões de muitos núcleos de sistemas gerenciadores de banco de dados relacionais surgidos no mercado nos anos 80. Os sistemas gerenciadores de banco de dados relacionais completaram o ciclo dos SGBDs ditos convencionais, cujas funcionalidades estavam centradas em dados como textos e numéricos, suficientes para atender as aplicações comerciais desse período.

A geração de tecnologias em SGBDs seguinte , anunciada pela comunidade científica da área de banco de dados [Bro_90][Sil_96a] [Sil_96b], surge com o propósito de enfrentar uma série de desafios proporcionados pela nova geração de aplicações que incluem projetos auxiliados por computador, sistemas de engenharia, engenharia de software e manufatura (CAD/CAM e CASE[1]); sistemas baseados em conhecimento; sistemas multimídia; sistemas baseados em linguagem de programação, integração de sistemas heterogêneos, suporte a dados semi-estruturados etc. As novas exigências não excluem o suporte aos sistemas comerciais também, cuja evolução vêm exigindo maior capacidade de representação semântica e gerenciamento de dados, além de proporcionarem também novos desafios, tais como o suporte a transações *on-line* (OLTP), suporte à decisão (OLAP) e recentemente o comércio eletrônico. Portanto, esses desafios incluem a necessidade de utilizar-se de estruturas de representação das informações mais sofisticadas do que as relações (tabelas) utilizadas pelos SGBDs relacionais.

Desde meados dos anos 80 e ao longo dos anos 90, a fim de atender às exigências das novas aplicações, alguns resultados concretos foram obtidos pelos pesquisadores e fabricantes nas áreas de gerenciamento de dados, dentre os quais podem-se destacar os seguintes:

- o aparecimento no mercado de vários SGBDs baseados na tecnologia de orientação a objetos, de tal modo que, no início da década de 90, apenas alguns poucos protótipos foram apresentados pela comunidade científica como trabalhos desenvolvidos por pesquisadores na área;

- o surgimento de gerenciadores que adotam tecnologias resultantes de experiências obtidas dos SGBDs relacionais, mas com o suporte da tecnologia da orientação a objeto, os quais são denominados de SGBDs Relacional-Objeto;

- mais recentemente, um grupo de pesquisadores e fabricantes tem-se reunido para propor a padronização para os sistemas gerenciadores de banco de dados orientado a objetos, o que resultou no padrão ODM - ODMG[Cat_97];

---

[1]CAD - *Computer-Aided Design*
CAM - *Computer-Aided Manufacturing*
CASE - *Computer Aided Software Engeneering*

- o aparecimento de novas estratégias de suporte ao gerenciamento de transações (transações longas), replicação dos dados nos vários nós de uma rede, suporte a "versões e configurações" que permitem construir objetos a partir de diferentes versões;

- suporte a novos tipos de dados, que podem ser encontrados, por exemplo, em sistemas multimídia, suporte à Medicina e sistemas de informação geográfica - GIS.

Atualmente, muitos estudos têm sido desenvolvidos para atender a tantas necessidades. No entanto, as pesquisas têm sido efetuadas ao redor basicamente de três estruturas para a construção dos gerenciadores: gerenciadores puramente relacionais; gerenciadores relacional-objeto; e gerenciadores orientados a objetos. Os gerenciadores relacionais privilegiam o desempenho do sistema, buscando as alternativas que conduzem ao tempo de resposta o mais curto possível, mesmo que seja à custa da flexibilidade do sistema. Já os gerenciadores orientados a objeto privilegiam o tratamento natural dos dados, com o máximo de flexibilidade, suporte a reusabilidade e facilidade de desenvolvimento de aplicativos, e integração conceitual com as características dos sistemas automatizados. Já os gerenciadores relacional-objeto podem ser considerados uma eventualidade histórica, pois sua existência não se origina nas necessidades que eles se propõem a atender, mas no fato que foram desenvolvidos como uma alternativa que aproveita a imensa quantidade de conhecimento desenvolvidos para suportar os gerenciadores relacionais, buscando adaptá-las para um ambiente orientado a objetos.

Nesse cenário, não existem ainda estudos que busquem desenvolver técnicas de implementação eficientes para suportar o desenvolvimento completo de um gerenciador orientado a objetos: a eficiência é sempre buscada no contexto dos gerenciadores relacionais, enquanto o desenvolvimento dos gerenciadores orientados a objetos tem por objetivo o suporte flexível e expansível das novas necessidades, sendo que eficiência é um conceito secundário. Neste trabalho procuramos criar um ambiente de ferramentas e conceitos para a construção de gerenciadores de dados orientados a objetos que propiciam suporte eficiente, flexível e expansível às necessidades atuais dos bancos de dados. A flexibilidade e expansibilidade é alcançada através das construções que implementam o suporte oferecido pelo NuGO aos modelos de dados, enquanto que a eficiência é obtida através dos algorítmos utilizados na implementação das camadas que compõem esse Núcleo Gerenciador e que permitem a uma base de dados criada sob seu domínio cresçer em proporções lineares ao número de registros requisitados pela aplicação. Dentre as principais necessidades que buscamos contemplar dentro desse panorama estão:

- suporte a transações de curta e de longa duração, bem como suporte interno à seleção de objetos baseado na semântica da aplicação, buscando minimizar a contenção de dados utilizando características sobre o conteúdo semântico dos dados manipulados;

- suporte a dados semi-estruturados, através da utilização de esquemas de dados integrados aos próprios dados, em contraste com a situação tradicional, na qual o esquema é mantido de maneira distinta dos dados em si;

- suporte nativo à abstração de generalização, tal como foi definida para a generalização de dados. O enfoque adotado permite que a generalização possa ser empregada sem a perda do desempenho típica que ocorre quando se utilizam operações de junção no modelo relacional para emulá-la;

- suporte a atributos multivalorados e atributos não atômicos. Esse suporte deve ser capaz de recuperar de maneira eficiente tanto os atributos que compõem a estrutura de tupla fundamental dos objetos quanto as estruturas de dados multivaloradas definidas pelos conceitos de orientação a objetos;

- suporte a atributos extras não previstos no esquema das aplicações. Esse suporte, integrado aos recursos de suporte a esquemas integrados aos próprios dados, auxilia a manipulação de dados semi-estruturados e a integração com múltiplas fontes de informação, tornando o sistema resultante adequado às aplicações envolvendo integração via internet e para o suporte a protocolos como SGML e XML[2].

1.5 Organização da tese

O presente trabalho está organizado em 6 capítulos, cujos conteúdos aparecem resumidos a seguir:

Capítulo 2: descreve os principais conceitos e tecnologias que envolvem a construção de sistemas gerenciadores de banco de dados orientados a objeto;

Capítulo 3: apresenta de maneira sucinta as principais características de alguns dos gerenciadores orientados a objetos encontrados da literatura e mostram os recursos de implementação utilizados nas respectivas construções;

Capítulo 4: descreve o Núcleo Gerenciador de Objetos, o contexto sobre o qual ele está inserido e as tecnologias utilizadas para a implementação de suas funcionalidades;

Capítulo 5: apresenta os módulos de implementação das camadas componentes do Núcleo Gerenciador de Objetos, as primitivas componentes desses módulos e alguns testes caracterizando o desempenho das opções de implementações propostas;

Capítulo 6: apresenta as contribuições, as propostas de trabalhos futuros e as conclusões a respeito da implementação do Núcleo apresentadas nesse trabalho.

---

[2]SGML - *Standard Generalized Markup Language*
XML - *eXtensible Markup Language*

Capítulo 2

FUNDAMENTAÇÃO CONCEITUAL

2.1 Considerações iniciais

O desempenho dos sistemas gerenciadores de base de dados - SGBDs - está diretamente ligado às técnicas utilizadas para a implementação de suas funcionalidades. Grande número de fatores devem ser considerados para que o suporte dado pelo SGBD atenda adequadamente às necessidades das aplicações que se propõe gerenciar. Nesse capítulo são apresentados os principais conceitos e tecnologias utilizados para a implementação de fatores que envolvem os principais módulos dos SGBDs: identificadores de objetos, *clustering*, *swizzling*, gerenciamento de *buffer*, gerenciamento de transações, controle de concorrência, bloqueio, recuperação e indexação. Parte razoável dos conceitos aqui expostos surgiu no desenvolvimento de técnicas de implementação de SGBDs Relacionais - SGBDRs - empregados também na implementação de SGBDs orientados a objetos - SGBDOOs. Outra parte desses conceitos e tecnologias é conseqüência das exigências da necessidade de oferecer suporte a uma nova geração de aplicações [Sil_96a] [Sil_96a] [Bro_90], as quais, portanto, estão presentes somente nos SGBDs mais avançados. Como os modelos de dados também afetam direta e profundamente funcionalidades a serem suportadas, os conceitos fundamentais sobre modelos de dados são também apresentados. No presente capítulo serão abordados apenas conceitos e tecnologias que de algum modo envolvem o escopo do trabalho.

2.2 Modelos de dados

Um modelo de dados é uma coleção de ferramentas conceituais que possibilitam descrever os dados, os relacionamentos entre eles, sua semântica e as regras de validade, completude e consistência de tais dados[Sil_99]. No início, a representação das informações através de modelos de dados atendia apenas à descrição de suas estruturas e associações abstraídas do mundo real. Posteriormente, nos modelos de dados mais avançados, passou a preocupar-se também com a representação do comportamento das informações a ser descrita através dessa ferramenta conceitual. Desse modo, a estrutura e o comportamento dos dados são os aspectos abordados atualmente pelos modelos de dados, e a tais modelos denominam-se modelos de dados orientados a objetos. Alguns modelos de dados têm sido apresentados com o propósito de atender aplicações específicas, como, por exemplo, modelos de dados para sistemas de informação geográfico: GISER[She_97], Geo-OMT[Bor_97]. Em outros casos, propõem-se requisitos desejáveis para a modelagem de aplicações, como, por exemplo, para aplicações multimídia [Gro_97].

Ao longo dos anos 50, 60 e 70, deu-se a inserção de grandes empresas (IBM, General Electric e a Honeywell) interessadas na fabricação de sistemas capazes de gerenciar grandes volumes de dados de modo eficiente, e como uma das conseqüências desse interesse das empresas, surge o primeiro padrão de modelo de dados [DBT_71]. O modelo de redes foi especificado pelo DBTG - *Data Base Task Group*, um subgrupo do comitê de linguagem CODASYL - *Conference On DAta SYstems and Languages*. O modelo de redes consiste na visão de que os dados podem ser representados pelos tipos de registro e relacionamento especificados através da construção de conjuntos com cardinalidade implícita de um-para-muitos. Assim, um tipo de registro pode participar em diferentes tipos de conjunto, podendo formar uma rede de relacionamentos. Um outro modelo de dados surgido nessa época é o hierárquico, no qual a representação de informações também é efetuada através dos tipos de registro e seus relacionamentos. Nesse modelo, os relacionamentos permitidos são um-para-muitos do tipo pai-filho, o qual possibilita a descrição dos dados de um empreendimento através da coleção de estruturas hierárquicas. Não existe documento que especifique o modelo hierárquico. Os princípios utilizados para a definição desse modelo são derivados do *Information Management System* - IMS, um produto da IBM[Elm_00].

Nos modelos hieráquico e de rede, a obtenção da informação é efetuada através da navegação pelos registros e seus relacionamentos. Através do posicionamento de um registro pai ou proprietário, pode-se obter registros filhos ou membros com o uso de comandos como: *get, get next, find, find next* etc. Assim, uma consulta deve ser descrita através de linguagem procedural hospedeira que contém os comandos para a navegação das informações representadas numa base de dados. Com o intuito de atenuar os problemas apresentados por tais modelos, foi proposto o modelo relacional. Os SGBDs baseados no modelo relacional permitem organizar grandes quantidades de dados e recuperá-los através de consultas mais declarativas do que navegacionais. O modelo relacional, dentre os modelos de dados surgidos até os dias de hoje, é considerado o de

maior êxito e grande potencial de aceitação no âmbito comercial e industrial. Isto se deve à facilidade em manipular seus conceitos e os recursos oferecidos pelos gerenciadores que o suportam.

2.2.1 Modelo relacional

O modelo relacional foi introduzido por E. F. Codd no início do anos 70 [Cod_70][Cod_71][Cod_72]. Esse modelo está centrado numa estrutura simples denominada *relação* ou *tabela*, sobre a qual aplica-se uma fundamentação teórica baseada nos conceitos de álgebra relacional e cálculo relacional de primeira ordem.

Toda relação possui um nome e representa uma entidade, conceito ou associação de conceitos do aspecto do mundo real que está sendo modelado. Uma relação é composta por *atributos* que possibilitam descrever as características do mundo real. Um atributo possui um conjunto de valores possíveis de ser representados, denominado de *domínio* do atributo. Por exemplo, a relação *Empregado* possui os atributos *nome_empregado, endereço* e *idade*, como está exemplificado na figura 2.1. O domínio do atributo *nome_empregado* é uma cadeia de caracteres de tamanho 30, *endereço* é uma cadeia de caracteres de tamanho 40 e *idade* é um inteiro de 0 a um determinado limite que representa o número de anos de vida do empregado.

Denomina-se *tupla* um conjunto de valores, um para cada atributo da relação, cada valor escolhido dentre os admissíveis pelo domínio do respectivo atributo ou, mais intuitivamente, a uma linha da tabela. Na figura 2.1, mostra-se para a relação *Empregado* uma das suas tuplas com as informações do empregado de nome Yúri Valente.

O modelo relacional é conceitualmente composto por algumas restrições descritas a seguir.

2.2.1.1 Domínio

A restrição de domínio no modelo relacional define que, para toda relação, os domínios de seus atributos devem ser atômicos, ou seja, os valores dos atributos são indivisíveis. Por exemplo, a idade da Pessoa de nome "João" é um único valor (30 anos) e não uma coleção de inteiros.

Empregado

nome_empregado	endereço	idade
Yúri Valente	R: XV de Novembro, 30	18
Lucas da Silva	R: Pedro Amaral, 100	21
Soraia Gomes	R: XV de Novembro, 30	35
João Souza	R: dos Anjos, 40	30
Ígor Rodrigues	R: Araribóia, 150	18

Projeto

código_projeto	descrição	data_início
101	Educação	22/05/94
220	Informatização	03/02/86
340	Novos Produtos	19/04/90
452	Reorganização	21/06/96
51	Produto X	10/12/97
49	Produto Y	03/05/99

Trabalho

nome_empregado	código_projeto
Soraia Gomes	101
Yúri Valente	101
João Souza	340
Ígor Rodrigues	51
Lucas da Silva	220

Figura 2.1: Exemplo com as relações: Empregado, Trabalho e Projeto

2.2.1.2 Chaves

Uma relação é formada por um conjunto de tuplas distintas, isto é, não existe numa relação duas tuplas com a mesma combinação de valores para todos os seus respectivos atributos. Essa restrição também se aplica a um subconjunto de atributos *s*, ou seja, em toda relação existe um subconjunto de atributos *s*, tal que não existem duas tuplas t_i e t_j com a mesma combinação de valores para *s*. A esse subconjunto de atributos *s* denomina-se superchave. Toda relação tem no mínimo uma superchave. Uma superchave pode ter atributos em excesso, isto é, pode-se retirar alguns atributos de *s* de modo que o subconjunto de atributos *s'* resultantes continua a satisfazer a restrição de superchave. A um subconjunto do qual não se possa eliminar um atributo e ainda manter-se a restrição de superchave denomina-se chave candidata. Numa relação podem existir mais de uma chave candidata. Deve-se escolher sempre uma das chaves candidatas para ser a chave primária da relação. Em geral escolhe-se como chave primária a chave candidata com menor número de atributos ou a que ocorre com maior freqüência nas consultas que envolvem a tabela. Portanto, o conjunto de atributos que se constitui numa chave primária é uma restrição para qualquer relação definida numa modelagem relacional. Essa restrição deve permanecer independentemente das atualizações sofridas pela relação em uma base de dados relacional.

2.2.1.3 Integridade de entidade

A restrição de integridade em entidades especifica que nenhum valor para os atributos componentes de uma chave primária numa relação pode ser nulo. Isso se deve à propriedade de uso dos valores da chave primária para identificar tuplas numa relação. Seria impossível identificar uma tupla caso ela contivesse valores nulos em algum de seus atributos componentes da chave primária da relação.

2.2.1.4 Integridade referencial

A integridade referencial é especificada entre duas relações com o objetivo de manter a consistência entre suas tuplas. Essa restrição é definida pela obrigatoriedade de certa tupla de uma relação somente poder referir-se a uma tupla de outra relação, se esta segunda existir. Por exemplo, na relação *trabalho* da figura 2.1, todo empregado que aparece nesta relação está definido na relação *Empregado*.

Os SGBDs baseados no modelo relacional oferecem uma Linguagem de Definição de Dados - LDD e uma Linguagem de Manipulação de Dados - LMD integradas numa única linguagem padrão denominada de SQL - *Structured Query Language* [Dat_93]. O SQL contém construtores baseados na álgebra relacional, tais como: união, intersecção, diferença e produto cartesiano. Além disso, o modelo relacional propõe a extensão a essas operações de álgebra relacional e oferecidas também pelo padrão SQL, denominadas operações relacionais, tais como: seleção e projeção. Outras linguagens comerciais também foram propostas para o modelo relacional: o QBE - *Query By Exemple*[Zlo_75][Zlo_82] que tem por base o cálculo relacional de domínio; e o QUEL, baseado no cálculo relacional de tuplas [Sto_76].

2.2.2 Modelos semânticos de dados

Os modelos semânticos de dados começaram a aparecer como modelos de banco de dados alternativos aos já consagrados relacionais, de rede e hierárquico. O precursor desses modelos foi o Modelo Entidade-Relacionamento - ME-R, proposto por P.P.Chen em 1976[Che_76], surgido com o objetivo de "modelar o mundo real tanto quanto possível"[Kho_94]. Para o ME-R, qualquer coisa do mundo real pode ser descrita como um objeto e suas associações. Um objeto do mundo real pode ser uma pessoa, um evento, uma instituição etc. A um objeto do mundo real que possui as mesmas características denomina-se conjunto ou tipos de entidades. Às caracaterísticas ou propriedades desejáveis para a descrição de um conjunto de entidades denominam-se atributos. Por exemplo, para o conjunto de entidades *pessoa* os atributos desejáveis

podem ser *nome, RG, CIC, endereço* e para o conjunto de entidades *projeto* tem-se os atributos *código do projeto, descrição* e *data de início*. Às associações entre conjuntos de entidades, denomina-se de conjuntos ou tipos de relacionamentos, por exemplo, *pessoa trabalha_em projeto*, a associação entre *pessoa* e *projeto* denominada de *trabalha_em* é um conjunto de relacionamento. Além dos conjuntos de entidades e de relacionamentos, o ME-R permite definir algumas regras semânticas denominadas restrições de cardinalidade. Através dessas regras pode-se expressar com quantas entidades uma outra entidade está associada através dum conjunto de relacionamentos.

Após a proposta original do ME-R por P.P.Chen em 1976, um conjunto de extensões foi sendo apresentado para esse modelo[Smi_77][San_79][Elm_85]. O modelo Entidade-Relacionamento Estendido - ME-RE, é assim denominado[Elm_00] por incorporar um conjunto de abstrações adicionais, como, por exemplo, superclasses/subclasses, herança, agregação, generalização/especialização e categorização.

O modelo entidade-relacionamento tem sido amplamente utilizado como ferramenta para projetos conceituais de banco de dados, posteriormente, transformados para os respectivos projetos lógicos. Um projeto lógico é composto pelo mapeamento do projeto conceitual para o modelo de dados suportado pelo SGBD, seja ele relacional, rede ou hierárquico[Elm_00]. Para a especificação de um projeto conceitual utiliza-se de notação gráfica denominada de Diagrama Entidade-Relacionamento - DER, cujos principais componentes são:

- **retângulos** - utilizados para representar conjunto de entidades;
- **losangos** - utilizados para representar conjunto de relacionamentos;
- **elipses** - representam atributos;
- **linhas** - ligam conjuntos de entidades aos conjuntos de relacionamentos e os conjuntos de entidades a seus atributos;

Assim como o ME-R, outros modelos semânticos de dados também devem ser ressaltados: o modelo semântico de dados[Ham_81], que apresenta alguns conceitos avançados de modelagem, como a abstração de classe e subclasse e o modelo funcional DAPLEX[Ber_94].

2.2.3 Modelos orientados a objetos

Um modelo de dados orientado a objeto é formado por um conjunto de conceitos do paradigma de orientação a objetos utilizados para modelagem de dados. Num modelo orientado a objetos são definidos os construtores semânticos que permitem especificar

as características de um objeto, como um objeto relaciona-se com outros objetos, como um objeto pode ser denominado e identificado.

Assim como o modelo relacional é fundamental para a definição da funcionalidade de um gerenciador de base de dados relacional, estabeleceu-se um padrão denominado de modelo de objetos ODMG 2.0[Cat_97] que define os conceitos e funcionalidades para bases de dados orientadas a objetos. Os construtores semânticos suportados por esse padrão são os seguintes:

- Os elementos básicos numa modelagem são os objetos e os literais. Todo objeto possui identificador único e os literais não têm identificadores;
- Os objetos e os literais similares são agrupados por tipos. Os elementos pertencentes a um mesmo tipo compartilham igual conjunto de estados possíveis (conjunto de propriedades) e comportamento (conjunto de operações). Um objeto é uma instância de seu tipo;
- O estado de um objeto é definido por valores atribuídos às suas propriedades. Uma propriedade pode ser um atributo ou uma referência para um ou mais objetos, nesse último caso, caracterizando um relacionamento entre objetos. O estado de um objeto pode ser alterado no decorrer do tempo;
- As operações especificadas para um objeto definem-lhe o comportamento. As operações podem possuir uma lista de parâmetros de entrada e saída, cada um com o respectivo tipo. Uma operação pode retornar um valor compatível com o tipo especificado;
- Um *esquema* numa base de dados é definido por uma Linguagem de Definição de Objetos - ODL. Um objeto, instância desse esquema armazenada numa base de dados, deve estar disponível para o compartilhamento entre vários usuários e aplicações.

2.2.3.1 Tipos

O tipo de objeto, também denominados classe, apresenta igual noção do conjunto de entidades do modelo entidade-relacionamento. A diferença é que os tipos de objeto são acrescidos de comportamento. Por exemplo, uma instância de nome *Maria* pode ser um objeto do tipo *Pessoa* e a operação capaz de criar uma instância de *Pessoa* é parte de seu comportamento.

Um tipo é definido através de dois componentes: especificação e implementação. Na especificação definem-se as características de interface de um tipo, ou seja, tudo aquilo que é visível a um usuário do tipo: as operações, as propriedades e

as exceções. As operações ou métodos especificam as ações (comportamento) definidos para instâncias do tipo. As propriedades, especificam as variáveis que devem compor os valores de estado para as instâncias de determinado tipo. As exceções podem ser disparadas pelas operações e comunicam os resultados de exceção. A implementação define como as operações suportadas pelo tipo são executadas.

A definição de um tipo permite especificar o grau de encapsulamento do estado e o comportamento para seus objetos. Assim, não existirá nenhum outro modo de acessar um objeto a não ser através da interface pública especificada pelo tipo.

2.2.3.2 Hierarquia de tipos

Nos sistemas orientados a objetos um tipo pode ser especializado em subtipos, e esses subtipos por sua vez poderão ser especializados em outros subtipos, permitindo a formação de uma hierarquia de tipos. No mundo real, freqüentemente ocorrem situações que solicitam a representação através de estruturas hierárquicas de subtipos. Por exemplo, um tipo *caminhão* é um *veículo*, assim como *ônibus* também é um *veículo*. No entanto, *caminhão* e *ônibus* se distinguem, pois possuem características particulares. Para instâncias do tipo *caminhão* é importante armazenar informações tais como a tonelagem. Essas informações são específicas de *veículos* do tipo *caminhão* mas não de instâncias do tipo *ônibus*. O mesmo acontece para o tipo *ônibus*, o qual necessita armazenar informações como o número máximo de passageiros transportados em cada instância desse tipo de *veículo*, tal informação é dispensável para *caminhão*. No entanto, *caminhão* e *ônibus* são *veículos*, portanto possuem características genéricas de *veículos*, como por exemplo, número de rodas e licença. Assim, o tipo mais genérico *veículo* é denominado de supertipo e os tipos *caminhão* e *ônibus* são subtipos do supertipo *veículo*, criando uma hierarquia de generalização/especialização. Essa hierarquia está associada através de relacionamentos tipo-subtipo entre os subtipos e seu supertipo. Este relacionamento também é denominado de especialização-generalização ou relacionamento *é-um*, no qual o supertipo é o tipo mais genérico e o subtipo o mais específico.

2.2.3.3 Herança

Através da hierarquia de tipos, para os modelos orientados a objeto, é possível utilizar-se de um recurso expressivo de reusabilidade, a herança do comportamento e do estado. Quando no uso de relacionamentos tipo-subtipo, um subtipo tem sua especificação baseada no supertipo. Assim, o subtipo herda as propriedades, comportamentos e mensagens de seu supertipo, além de poder ter suas propriedades, comportamentos e mensagens específicas e adicionais.

2.2.3.4 Extensões de tipos

As extensões de um tipo são os conjuntos de todas as instâncias de determinado tipo armazenada numa base de dados. Para uma base de dados relacional, toda instância criada constitui extensão de determinada tabela. Ao contrário, no modelo de objetos ODMG, o projetista da base de dados pode decidir se manterá ou não a extensão automática para cada tipo. Tal flexibilidade deve-se à sobrecarga provocada pela manutenção (inserção/remoção) das extensões de um tipo. A extensão de um tipo é freqüentemente chamada de classe de objetos, ou seja, tipo representa um conjunto de características e classe representa o conjunto de todas as instâncias do tipo.

2.2.3.5 Chaves

Assim como no modelo relacional, as instâncias de objetos podem ser, em alguns casos, identificadas através de valores, para um conjunto das suas propriedades. A esse conjunto de propriedades, no modelo relacional, denomina-se chave candidata. O escopo da unicidade para uma chave é a extensão do tipo. As chaves formadas de uma única propriedade denominam-se chaves simples. Uma chave formada por mais do que uma propriedade denomina-se chave composta.

2.2.3.6 Objetos

Assim como no modelo entidade-relacionamento, os objetos, em sistemas orientados a objeto representam entidades do mundo real, como, por exemplo, uma pessoa, um fato, etc. A um objeto estão associados seu estado e comportamento, definidos pelas propriedades e métodos especificados pelo tipo a que pertencem. Um objeto é identificado através do seu Identificador de Objetos - OId, um valor único e inalterável na base de dados, cuja existência independe dos valores componentes do seu estado. Um objeto pode estar relacionado com outros objetos através do uso dos OIds dos objetos os quais se relaciona.

Através dos métodos definidos para um tipo de objeto pode-se criar instâncias de objetos, eliminar objetos, definir o tipo de bloqueio (leitura, escrita, atualização) para um objeto, comparar identidade de objetos ou executarem-se operações pertinentes a cada tipo.

Os objetos podem ter um ou mais nomes definidos pelos usuários ou programadores. Esses nomes, denominados de *object names,* são independentes dos valores de seus atributos, de sua chave e seu OId, portanto não são definidos pela interface do tipo do objeto ou pelo sistema. Um gerenciador de objetos deve fornecer as

funções para a definição desses nomes e associá-los ao OId do objeto desejado, cujo escopo de unicidade do OId é toda a base de dados em que foi definido.

Um objeto pode estar num estado transiente ou persistente. Durante a execução de um processo, o objeto é alocado em memória gerenciada pela linguagem de programação. Nesse caso seu estado é denominado transiente. Quando o processo é concluído e as informações que componentes do objeto são armazenadas na base de dados, o objeto passa para o estado persistente. Assim, um objeto persistente continua a existir mesmo se o processo ou procedimento que o criou tenha sido concluído. Esse conceito de estado de um objeto em sistemas orientados a objetos é diferente do modelo relacional. Nesse último modelo, toda tupla de uma relação é persistente. No caso do modelo de objetos, pode-se ter para um tipo de objeto, instâncias persistentes e transientes. Uma base de dados é o conjunto de todos os objetos persistentes e transientes existentes num dado instante. Note-se que poderá haver mais de uma cópia de um mesmo objeto, por exemplo uma cópia na memória de um processo e outra em disco. A identidade do objeto é preservada pelo OId comum a todas as cópias.

2.2.3.7 Objetos-coleção

O conceito de coleção no Modelo de Objetos aplica-se a um conjunto de objetos distintos de um mesmo tipo. Uma instância de uma coleção pode ser um tipo atômico, uma outra coleção ou um tipo literal.

As coleções suportadas pelo modelo de objetos ODMG, são:

- *conjunto*: objeto conjunto é uma coleção desordenada de objetos distintos, ou seja, sem duplicatas. As operações possíveis para esse objeto são: união entre dois conjuntos, intersecção, diferença, verificação de pertinência de um subconjunto em um conjunto etc;
- *bag*: objeto *bag* é uma coleção desordenada de objetos, que permitem repetições de objetos, ou seja, com duplicatas. O objeto *bag* permite operações como: verificação de pertinência de um elemento num *bag*, criação de um *bag* resultante da união de outros dois, intersecção e diferença;
- *lista*: objeto lista é uma coleção ordenada de objetos. A ordem num objeto lista é posicional e permite operações como: remover o primeiro elemento da lista, remover o último, substituir um elemento por outro, obter um elemento numa determinada posição, concatenar duas listas, inserir uma lista noutra etc;

- *array*: objeto *array* é uma coleção ordenada, cujo tamanho é definido dinamicamente, e seus elementos podem ser localizados pela posição. O objeto *array* permite operações como: substituir um elemento de uma determinada posição, remover um elemento de uma determinada posição, obter um elemento numa posição, redefinir o tamanho do *array*;
- *dicionário*: objeto *dicionário* é uma seqüência desordenada de pares chave e valor não repetidos. Esse objeto permite operações como: inserir um par chave e valor, eliminar um par chave e valor, obter um par através da especificação da chave, etc. *Array* é um caso particular de dicionário, no qual a chave é um número inteiro de um domínio fechado, em que toda a seqüência de valores está permanentemente definida.

Cada um desses objetos-coleção possui um conjunto de operações que suportam as respectivas características, como foi exemplificado. Algumas das operações são básicas e herdadas por todos, por exemplo, as operações de criação, inserção, eliminação de uma instância de objeto numa coleção. Pode-se, através das operações comuns, verificar o número de instâncias numa coleção, se está vazia, se foi ordenada, se permite repetições, se contém determinado elemento etc.

O suporte à definição de objeto-coleção e identificador de objetos permite a implementação de objetos complexos. Um objeto complexo[Ber_94] possui em seus atributos valores que podem ser um objeto ou uma coleção de objetos. Deste modo, um objeto complexo pode ser definido por outros objetos.

2.2.3.8 Objetos atômicos

Um objeto atômico é um tipo definido pelo usuário. No modelo ODMG2.0 não está definido qualquer tipo de objeto atômico embutido.

2.2.3.9 Literais

Os objetos literais não têm OId e são em geral representados diretamente pelo valor a que correspondem. O padrão ODMG define os seguintes:

- literais atômicos: *long, short, unsigned long, unsigned short, float, double, boolean, octet, char, string e enum*;

- literais-coleção: *conjunto*, *bag*, *lista*, *array* e *dicionário*. Esses literais são os mesmos definidos para os objetos coleção, com a diferença de que não têm identificadores. O modelo ODMG2.0 engloba o modelo relacional ao definir o tipo *tabela* para expressar consultas do SQL. O tipo *tabela* é semanticamente equivalente a uma coleção de estruturas (*structs*);
- literais estruturados: um literal estruturado tem um número fixo de elementos, cada um com nome de variável e pode ainda conter um valor literal ou um objeto. Um elemento de uma estrutura é referenciado por um nome de variável, por exemplo, endereço.cep = "15050-000"; endereço.cidade = "SJRPreto". As estruturas suportadas pelo modelo ODMG são: data, intervalo, tempo, *timestamp*. Além disso, os usuários poderão definir as próprias estruturas[1];
- literais nulos: todo literal (atômico, coleção, etc) possui valor nulo, por exemplo, *nullable_float* ou *nullable_set<>*. A semântica para valores nulo é a mesma definida pelo SQL-92 [Dat_93].

2.2.3.10 Objetos estruturados

Alguns objetos são comumentemente utilizados pelas aplicações, portanto intrinsecamente definidos pelo modelo de objetos. São estes os objetos estruturados: data, intervalo, tempo e *timestamp*. Para cada objeto estruturado existe um conjunto de operações pré-definidas para a sua manipulação.

2.2.3.11 Atributos

Os atributos são utilizados para descrever as características de um tipo de objeto e lhe definem o estado abstrato. Por exemplo, um tipo de objeto *Pessoa* pode ser definido do seguinte modo pelo padrão ODMG2.0:

---

[1]Através da palavra chave *struct* pode-se confeccionar os tipos definidos pelos usuários.

```
interface Pessoa{
        attribute short idade;
        attribute string nome;
        attribute enum sexo {masc, fem};
        attribute endereço end_residência;
        attribute set<num_tel> telefones;
        attribute Departamento dept;
}
```

Um objeto do tipo *Pessoa* é composto de um conjunto de valores de atributos especificados para esse tipo. O atributo *dept* para uma instância de *Pessoa* contém um identificador de objeto de tipo *Departamento* que representa o departamento ao qual a pessoa está alocada.

Os atributos, nesta versão do modelo de objetos ODMG 2.0, não são de "primeira classe", isto é, os atributos não são em si mesmos um objeto, portanto não têm OId. Assim, não é possível definir atributos de atributos, relacionamentos entre atributos ou operações específicas de subtipo para atributos.

2.2.3.12 Relacionamentos

A versão 2.0 do modelo de objetos ODMG permite somente a existência de relacionamentos binários de cardinalidade um-para-um, um-para-muitos e muitos-para-muitos. Por exemplo, um relacionamento binário entre *Professor* e *Curso* pode ser especificado do seguinte modo:

```
interface Professor{
        .....
        relationship set<Curso> ministra
                inverse Curso:: é_ministrado_por;
        .....
}
```

e

```
interface Curso{
        .....
        relationship Professor é_ministrado_por
                inverse Professor:: ministra;
        .....
}
```

Neste exemplo, determinado professor pode ministrar um conjunto de cursos e um curso é ministrado por um único professor, configurando-se, assim, um relacionamento de cardinalidade um-para-muitos entre *Professor* e *Curso*. Além disso, o SGBDOO deve garantir a integridade referencial dos relacionamentos. Para o exemplo, isto significa que, se determinado *curso* for extinto, deverá ser eliminado do conjunto dos *cursos* do respectivo *professor* responsável por ministrá-los. Do mesmo modo, deve-se garantir para determinado *curso* não se aloque um professor inexistente para ministrá-lo.

Um valor de atributo pode ser um objeto, por exemplo:

```
interface Turma{
        .....
        attribute Estudante  primeiro_da_turma;
        .....
}
```

Nesse caso, o atributo *primeiro_da_turma* é um objeto do tipo *Estudante*. Diz-se que esse tipo de associação não é um relacionamento verdadeiro, pois, diferentemente do exemplo dado antes para *Professor* e *Curso*, não ocorrerá o controle de integridade, em ambos os sentidos, entre os tipos de objeto envolvidos nessa associação. Portanto, apenas é efetuada a restrição de integridade que verifica se o valor para o atributo *primeiro_da_turma* é um objeto existente do tipo *Estudante*.

Em[Ber_95] é apresentada proposta de extensão ao modelo de objetos ODMG [Cat_94] para o suporte a objetos compostos. Um objeto composto é um objeto construído pela agregação de outros objetos componentes. Os objetos componentes estão associados ao objeto composto através de relacionamentos denominados agregação ou *é-parte-de*. O aninhamento formado entre um objeto e seus componentes, através de relacionamentos de agregação, forma uma hierarquia de agregação[Ber_94]. Esse tipo de associação entre objetos faz com que se definam algumas restrições, como integridade referencial, de exclusividade e dependência existencial. No caso da

integridade referencial, deve-se assegurar que um objeto só poderá ser definido como componente de um objeto composto, se tal objeto composto existir. No caso da exclusividade, um objeto só poderá ser componente de um único objeto composto. A restrição de dependência existencial define que, se um objeto composto for eliminado, seus objetos componentes também deverão ser eliminados. O modelo para objetos compostos proposto em[Ber_95] suporta a restrição de integridade referencial equivalente à suportada pelo SQL em SGBDs relacionais, mas não suporta as restrições de exclusividade e dependência existencial.

2.3 Identificador de objetos

Através do controle de um gerenciador, um objeto pode ser armazenado e compartilhado, sem perda de informação, por múltiplos aplicativos em execução concorrente. Além disso, conceitos de orientação a objetos necessitam de estruturas de suporte diferentes das exigidas para modelos convencionais. Um exemplo são os objetos complexos que podem utilizar-se de referências a outros objetos, confeccionando-se relacionamentos de forma aninhada entre objetos. A disponibilidade de objetos complexos induz ao aumento da complexidade do processamento de consultas. Estruturas de índices devem ser elaboradas de modo a permitir o suporte à localização eficiente de tais objetos. Por outro lado, a geração e o controle dos identificadores de objetos deve atender requisitos mais rígidos do que utilizados por programas ou linguagens de programação, nos quais não se considera a persistência de objetos. Por exemplo, o caso do emprego de linguagens de programação orientada a objetos não persistentes como C++ ou Eifell.

Tradicionalmente, as linguagens de programação referem-se a objetos através de seu endereço em memória. Seu endereço (ou ponteiro como é denominado em C++) é usado como identificador físico de um objeto. Não se considera serem os mesmos objetos que tenham o mesmo valor, se estiverem ocupando diferentes endereços de memória. Por outro lado, os sistemas de gerenciamento de base de dados - relacionais, rede etc. - tradicionalmente consideram que dois objetos (tuplas, registros, dependendo do modelo de dados envolvido) são os mesmos se tiverem o mesmo valor em seus atributos chave. Assim, é comum que as linguagens de programação adotem a identificação física de objetos, enquanto que os SGBDs adotem a identificação lógica, ou seja, não especificam a localização física do objeto [Kim_92] [Ber_94] [Cat_94].

Os gerenciadores de objetos estendem a armazenagem persistente à linguagem de programação, além de prover comportamento (descrito numa linguagem de programação) a objetos armazenados numa base de dados. Assim, é necessário unificar os diferentes conceitos de identificação, uma vez que:

- sob o ponto de vista da linguagem de programação, poderão existir diversas cópias de um mesmo objeto - na base de dados em si e nos diversos programas aplicativos em execução - impedindo que seu identificador seja sua referência em memória;
- sob o ponto de vista de um SGBD, um objeto pode estar em diferentes estágios de atualização, e, portanto, com diferentes valores em seus atributos, e ainda ser o mesmo objeto. É possível, mesmo que um objeto tenha os atributos chave alterados, continuar a ser o mesmo objeto - impedindo, assim, que a identificação seja feita pela comparação dos valores dos atributos chave.

Para contornar tais requisitos conflitantes, os gerenciadores de objetos realizam a identificação de objetos ao atribuir um identificador de objetos - OId - para diferenciar um objeto de outro num domínio de armazenamento. Na maioria das vezes o domínio de armazenamento é a base de dados. Um OId é um código gerado automaticamente pelo sistema, único para cada um dos objetos da base de dados - no caso de bases distribuídas único para todas as partes da base de dados da rede - inalterável durante o tempo de vida do objeto, inclusive se for eliminado, não deverá ser reutilizado, e independentemente dos valores de seus atributos[Cat_94].

Os modelos de dados tradicionais utilizam-se do conceito de chave primária na definição dos identificadores para a recuperação das informações armazenadas em base de dados. No modelo relacional, um conjunto de atributos que formam uma chave primária de uma relação é suficiente para que se recupere ou identifique uma tupla desejada. Através dos valores para os respectivos atributos da chave primária é possível obter a tupla que os contém. Se os valores que identificam uma tupla forem alterados, a tupla torna-se outra, além de os antigos identificadores estarem perdidos. Nos modelos de dados orientados a objeto, o uso do OId é inacessível aos usuários, portanto livres de modificações, controlados pelo sistema, e tem-se a garantia da unicidade do objeto na base de dados. Apesar do OId, o conceito de chave primária deve ser preservado pelos modelos de dados orientados a objetos, pois em algumas situações é necessário fazer-se uso de atributos identificadores definidos pelo usuário. Nesse caso, o sistema usa os valores da chave fornecidos pelo usuário para localizar o objeto e seu respectivo OId. O modelo de objetos ODMG2.0, apresentado na seção 2.2.2, utiliza-se do conceito de chave primária.

Várias abordagens têm sido utilizadas para a manipulação de identificadores de objetos. Um OId pode ser físico, isto é, ele pode conter o atual endereço do objeto; ou pode ser lógico, isto é, pode ser mapeado através de um índice para obter a localização do objeto. Existem mais do que uma abordagem para ambos identificadores físicos e lógicos, resultando em pelo menos quatro espécies de OIds [Cat_94]:

- Endereços: o modo mais simples para a confecção de OId é o que tradicionalmente se usa para linguagens de programação: o endereço físico do objeto;
- Endereços estruturados: muito utilizados pelos modelos relacionais, relacionais-estendidos e orientados a objetos, são os OIds com componentes físicos e lógicos. Tais identificadores contém os endereços de segmento, página e o número do *slot* lógico. Essa forma de endereçamento permite fazer-se a alteração do local físico de armazenamento do objeto, sem que para isso tenha-se que modificar seu OId;
- *Surrogates*: são OIds puramente lógicos e podem ser gerados por algum algoritmo que garanta a unicidade desses identificadores (por exemplo, tempo, data, contador etc). Esses identificadores devem ser mapeados para o endereço físico do objeto tipicamente através de índices;
- *Surrogates* tipados: uma variante do identificador *surrogate* que contém ambos um identificador de tipo e um identificador do objeto. O identificador de objeto é gerado por diferentes contadores, um para cada tipo. Isso permite obter o tipo sem ter de acessar o objeto, uma vez tendo-se o Oid, tem-se o código do tipo do objeto. Em outros sistemas, deve-se efetuar a busca para obter-se o tipo do objeto.

O modo como os OIds são implementados está relacionado com o desempenho do SGBD na sua utilização. Durante o processo de recuperação de um objeto, deve-se considerar o número de acessos exigidos a disco, o número de indireções necessárias até que o endereço físico do objeto desejado seja obtido dado o seu OId etc. Além disso, os problemas mencionados devem ser considerados na implementação do OId como a capacidade em manter-se único em toda a base de dados durante o tempo de vida do objeto e/ou da base de dados. Este aspecto está relacionado com a capacidade do OId em manter-se inalterado apesar das possíveis realocações de espaço físico exigidas para um objeto durante o tempo de vida. Outro fator de desempenho é o tamanho do OId, de acordo com o qual o uso de estruturas muito grandes para a sua representação poderão dificultar a utilização.

2.4 *Clustering*

As técnicas de *clustering* tem como objetivo alocar em espaços físicos próximos objetos logicamente relacionados, ou seja, que são freqüentemente acessados em conjunto. O agrupamento desses objetos visa otimizar as operações de I/O numa base de dados, evitando-se efetuar um número relativamente exagerado de acessos a disco comparado ao tamanho do objeto a ser recuperado. Característica capaz de indicar se a

estratégia de *clustering* utilizada é eficiente é a baixa quantidade de espaços disponíveis nas páginas alocadas pelo SGBDOO[Kim_95].

Os SGBDs relacionais utilizam-se de duas técnicas de *clustering*[Ber_94]: uma permite o armazenamento das tuplas de uma relação num mesmo segmento de página no disco, baseado num ou mais valores de atributos da relação; a outra técnica, permite o armazenamento das tuplas de várias relações num mesmo segmento, isso se as tuplas tiverem um ou mais atributos em comum, e se esses atributos têm os mesmos valores.

Diferentemente dos SGBDs relacionais, as estratégias de *clustering* aplicadas nos SGBDOOs levam em consideração objetos complexos, a herança e a presença de métodos. Por exemplo, os objetos complexos podem ser representados numa estrutura hierárquica ou num dígrafo acíclico - DAG. As estratégias para *clustering* em SGBDOOs podem ser classificadas do seguinte modo [Cat_94]:

- Objetos compostos - um caso particular de objetos complexos, no qual a estratégia de *clustering* baseia-se nos relacionamentos de agregação. Esses *clusters* podem ser definidos em tempo de execução, seja pela indicação de um particular objeto na formação de um objeto composto, seja pela definição de esquemas, nos quais são definidos os relacionamentos de agregação;
- Referências - a proximidade física dos objetos é definida pela existência de referências nos objetos que são componentes das associações. Assim, os objetos e as informações a ele atreladas poderão ser facilmente recuperadas pela proximidade física. Esse caso é uma generalização do *clustering* definido por objetos compostos;
- Tipos - o agrupamento é definido pelo tipo dos objetos. Por exemplo, todos os objetos do tipo Pessoa devem ser armazenados num mesmo segmento. Os subtipos, que por ventura forem contemplados pelo tipo, devem ser armazenados no mesmo segmento. Esta estratégia é equivalente à utilizada pela maioria dos modelos relacionais onde, ao invés dos tipos, a proximidade das tuplas é definida pelas relações;
- Índices - assim como em SGBDs relacionais, uma coleção de atributos podem definir um agrupamento. Os objetos poderão estar fisicamente próximos de acordo com os valores de atributos pré-definidos. Essa estratégia terá sentido quando a freqüência dos acessos exigir tal organização;
- *Custom* - esta estratégia consiste em armazenar um objeto, quando de sua criação, em espaços onde existiam outros objetos. Portanto, trata-se de um processo não definido por relacionamento lógico entre objetos, mas pela tentativa de ocupação de espaços liberados por objetos eliminados;

2.5 *Swizzling*

O espaço físico para o armazenamento persistente dos objetos numa base de dados, geralmente é maior que o espaço disponível em memória principal para suas alocações transientes. Isto faz com que o número de bytes utilizados para o armazenamento dos endereços transientes seja menor que o exigido para os endereços persistentes. Essa diferença exige que alguns passos intermediários sejam executados de modo que os endereços dos objetos na base de dados sejam convertidos adequadamente em endereços na memória principal ou vice-versa. A esse processo de conversão (ou reversão) de endereços denomina-se *swizzling*[Kim_92] [Ber_94] [Cat_94] [Sil_99]. Uma vez que foi efetuado o *swizzling*, o endereço resultante da conversão permite que a manipulação do objeto no modo transiente seja mais rápida, pois evita que, nas próximas requisições do objeto, a busca seja efetuada no modo persistente. Esse processo não é tão simples, pois pode envolver objetos com endereços transientes que também sofreram *swizzling*. Portanto, quando tais objetos forem novamente escritos na base de dados, deverão ser efetuadas todas as conversões de endereços transientes para persistente. Essa operação permite também a existência de diversas cópias do mesmo objeto em diferentes espaços de endereçamento de diferentes processos, tanto no mesmo quanto em vários processadores.

Existem dois tipos de *swizzling*[Sil_99]: o que foi descrito, denominado de *swizzling* por software e o *swizzling* por hardware[Jin_98]. Esse segundo, cujo nome justifica-se por utilizar-se do hardware de gerenciamento de memória do computador tem como objetivo minimizar o trabalho dos programadores na manipulação dos endereços, com a tarefa de lembrar-se sempre da existência dos dois tipos de endereços persistentes e transientes. O *swizzling* por hardware utiliza-se dum único endereço, cuja fusão ocorre através da extensão do tamanho dos endereços transientes para o mesmo tamanho dos endereços persistentes, com um bit adicional que diferencia entre os dois tipos de endereço. O custo em utilizar-se de endereços de maior tamanho é mais alto e nem todos os sistemas permitem tal possibilidade. No entanto, esta forma de *swizzling* permite a transparência entre os endereços sem a necessidade de adaptações, além de permitir a representação de endereços persistentes e transientes nas mesmas quantidades em termos de bytes.

2.6 Gerenciamento de *buffer*

Num SGBD, a tarefa de se efetuar a transferência dos blocos do disco para a memória principal e vice-versa, com os itens de dados manipulados pelas aplicações, é do gerenciador de *buffer*. Como o espaço em memória principal é pequeno em relação ao volume de informações manipulados pelas aplicações, uma porção limitada de memória principal é reservada para conter os blocos lidos do disco. A porção que contém uma cópia de bloco denomina-se de *buffer*.

O objetivo do gerenciador de *buffer* é atender às aplicações proporcionando o mínimo possível de acessos ao disco. Para que isso ocorra, o gerenciador de *buffer* deverá ter como compromisso manter na memória principal o máximo de tempo possível, os blocos requisitados pelas aplicações à medida que gerencia uma quantidade limitada de *buffers.*

Como não é possível prever quais blocos serão requisitados pela aplicação e como o espaço é insuficiente para receber todos os blocos utilizados, os gerenciadores de *buffer* utilizam-se de políticas para a substituição de *buffer*, ou seja, os blocos já utilizados deverão ceder espaço para os que estão sendo requisitados. Essa política, inicialmente utilizada em sistemas operacionais, é baseada numa análise do comportamento passado, para prever o comportamento futuro. Uma dessas políticas de substituição de *buffer* é o uso menos recente ou LRU - *Least Recentely Used*, na qual o *buffer* candidato a receber o bloco novo é o utilizado há mais tempo. Uma outra política é o mais recentemente usado ou MRU - *Most Recentely Used*, na qual o *buffer* candidato a receber o novo bloco é o usado a menos tempo.

Em alguns SGBDOOs, a disponibilidade dos objetos para as aplicações pode ser feita através do uso de dois gerenciadores de *buffers*, um no servidor compondo-se de um gerenciador de *buffers* de páginas obtidas diretamente do disco e outro, um gerenciador de *buffers* de objetos, em cada processo cliente, composto de objetos obtidos das páginas no servidor. A esse equema de *buffering* de objetos denomina-se *dual-buffer*[Kim_92]. O uso do *dual-buffer* deve-se à necessidade de gerenciar o objeto em duas diferentes situações: quando em disco, deverá ser recuperado eficientemente em seu formato de armazenamento; quando estiver em memória, deverá ser manipulado no formato exigido pela linguagem de programação.

Uma forma mais simples de disponibilizar os objetos, bastante utilizada pelos SGBDOOs[Cat_94], é através da utilização de um único gerenciador de *buffer.* Nesses casos, através do gerenciador de *buffer* contendo páginas armazenadas na memória principal, obtêm-se os objetos do disco que serão, posteriormente, alocados em espaços de memória principal utilizados pela aplicação. Um outra forma, é a implementação do gerenciador de *buffer* utilizando-se da memória virtual. Com isso, otimiza-se a disponibilidade dos objetos que são transferidos diretamente da memória virtual para a memória utilizada pela aplicação.

2.7 Gerenciamento de transação

Uma transação consiste numa sequência de operações de leitura e escrita efetuadas por um processo numa base de dados [Kim_92]. Geralmente uma transação é o resultado da execução de um programa de aplicação escrito numa linguagem de alto

nível (Cobol, C++, etc) ou linguagem de manipulação de dados (SQL, OQL etc.) suportada pelo SGBD.

O gerenciamento de transações é uma das funcionalidades oferecidas pelos SGBDs, pois é fundamental para a integridade, compartilhamento e recuperação de uma base de dados [Cat_97]. Através de políticas e estruturas de dados de suporte ao gerenciamento de transação, são implementadas as condições necessárias para que uma base de dados possa sempre manter um estado consistente, compartilhar informações persistentes e recuperá-las, caso possa ocorrer qualquer falha durante seu uso. Para que os recursos oferecidos pelo sistema de gerenciamento de transação sejam acionados, é necessário proceder-se ao uso das operações de delimitação que compõem uma transação. Toda transação deverá ser iniciada e finalizada, sendo que término poderá ser um operação de cancelamento ou efetivação das atualizações executadas pela transação na base de dados. As operações básicas de uma transação são as seguintes[Elm_00]:

- *begin*: uma transação é inicialmente criada através de uma operação explícita *begin*;
- *read e write:* define as operações de leitura e gravação para os itens utilizados numa transação;
- *commit*: define o encerramento de uma transação, ou seja, ao executar um comando *commit*, todas as alterações efetuadas pela transação deverão estar armazenadas na base de dados;
- *abort*: define o cancelamento de uma transação, isto é, qualquer atualização que tenha sido efetuada pela transação deverá ser cancelada.

O modelo ODMG 2.0 define algumas operações adicionais às apresentadas acima que são as seguintes:

- *checkpoint*: equivalente a uma operação de *commit*, onde na sua aplicação, todas as atualizações efetuadas pela transação são efetivadas na base de dados de modo a torná-las permanente. Sua diferença com o *commit* é que após a aplicação de uma operação *checkpoint* a transação continua aberta e seus bloqueios não são liberados;
- *join* e *leave*: operações que, respectivamente, associam e dissociam uma tarefa corrente no objeto transação. Se uma implementação permite que vários objetos transação possam estar ativos, então essas operações possibilitarão que uma tarefa possa alternar entre tais transações existentes;

Segundo o modelo de objetos ODMG 2.0[Cat_97], uma transação é uma unidade lógica nos SGBDOOs que deve garantir as propriedades ACID, um acrônimo formado pela iniciais das seguintes propriedades:

- Atomicidade - uma transação é uma unidade atômica, isto é, ou se realiza completamente ou não realiza nada. Isto significa que todas as operações de uma transação, ou se acham totalmente refletidas de modo persistente numa base de dados, ou nada ocorre. Não se verificam efeitos parciais;
- Consistência - a execução correta duma transação deve conduzir a base de dados de um estado consistente para outro;
- Isolamento - a execução de uma transação, até que se encerre (*commit*), não deverá ser visível às outras transações, portanto, uma transação não deve depender ou então ser afetada pelo conteúdo de outras transações em execução paralela;
- Durabilidade - ao final duma transação, o resultado de suas atualizações deverá persistir na base de dados, mesmo na ocorrência de falhas.

Os SGBDs convencionais e alguns SGBDOOs suportam as propriedades ACID, embora algumas aplicações avançadas exijam relaxamento em algumas dessas propriedades. Um exemplo são SGBDOOs que suportam aplicações multimídia[Gro_97] [Kim_95], sistemas de trabalho cooperativo, dentre outros. Esses sistemas exigem modelos de transação que contemplam transações mais longas, situação que não ocorre com os SGBDs convencionais, cujas transações são de curta duração. Exemplos desses modelos de transação para essas aplicações mais avançadas são apresentados a seguir.

2.7.1 Transações longas

Algumas aplicações avançadas exigem transações mais demoradas (horas ou dias). Por exemplo, um sistema de reserva em agências de turismo é composto pela reserva do vôo, reserva de automóvel e reserva de hotel. Nesse caso, se, durante a operação de alguma dessas reservas, ocorrer falha, pela propriedade de atomicidade da transação, todo o processo de reserva efetuado anteriormente deverá ser cancelado. Isso não é desejado, uma vez que o trabalho feito anteriormente poderá ser custoso e, portanto, é indesejável que possa se perder. Além disso, se a transação for demorada e complexa, maior a probabilidade de ocorrer conflitos entre outras transações. Assim, em aplicações como essa, é importante que parte do trabalho efetuado numa transação longa esteja representada na base de dados de modo persistente e disponível para as outras transações. Para atender a estas aplicações alguns SGBDs propõem o suporte através de modelos de transação que apresentam algum relaxamento nas propriedades ACID. Por exemplo, que as transações longas possam ser subdivididas em transações curtas ou, em outros modelos, que as transações também possam se reunir numa transação cooperada. Exemplos de modelos de transações longas são apresentados a seguir.

2.7.2 Transações aninhadas

Uma das propostas para o suporte às transações longas é o modelo de transações aninhadas introduzido por Moss em 1981[Kho_94]. Como o nome sugere, uma transação longa pode ser composta por subtransações curtas, também denominadas transações filhas. Nesse modelo, uma transação raiz T pode ter um conjunto de transações filhas (t_1, t_2, ..., t_n), cada uma dessas subtransações t_j também poderá ter um subconjunto de transações filhas, e assim por diante. A vantagem do modelo aninhado de transações é que, se uma subtransação falha, todo o trabalho efetuado por T não precisará ser desfeito, somente a subtransação que falhou é que poderá ser desfeita. Mesmo assim, alguns modelos permitem que a transação pai da que falhou tente novamente executar a transação que falhou, ou que tente uma transação de compensação para minimizar os efeitos da falha. Agora, se a transação T falha ou sofre uma operação de *abort*, então todas as subtransações serão desfeitas, mesmo que tenham sido efetivadas. A outra situação possível, ou seja, para que a transação raiz T seja encerrada, todas as subtransações deverão ter sido encerradas. Esse é o caso em que o modelo de transações proporciona o relaxamento da propriedade de atomicidade, pois a existência de subtransações induz à quebra dessa propriedade.

2.7.3 Transações coooperativas

Os modelos de transação cooperativas permitem que subtransações ligadas a uma transação raiz possam trabalhar de modo colaborativo na realização de tarefas comuns. Alguns modelos de transação cooperada têm sido propostos, entre esses há o modelo apresentado por Nodine em 1992[Kho_94] e o modelo orientado a grupos[Kim_95]. A principal característica desse modelo de transação é possibilitar a visibilidade entre as transações cooperadas, isto é, através do relaxamento da propriedade de isolamento, as transações no modo cooperado têm acesso aos resultados de outras transações pertencentes ao mesmo modo. O grau de cooperação é definido pelo usuário que estabelece as regras de coordenação e sincronismo entre as subtransações integrantes desse estado. Esses sistemas que permitem cooperação entre transações apresentam como principais desafios a definição de técnicas para o sincronismo ou coordenação de operações cooperadas e conflitos entre as transações[Zha_99].

2.8 Controle de concorrência

A maioria dos SGBDs permite que várias transações possam ser executadas de modo concorrente. Nessas condições, controlar as atualizações dos dados passa a ser uma tarefa que exige esforços adicionais, cujo objetivo é manter sempre a base de dados

num estado consistente. Um esquema de controle de concorrência deverá garantir a capacidade de serialização das transações em execução concorrente, ou seja, a execução intercalada das transações deverá produzir o mesmo efeito na base de dados que a sua execução serial. Esta tarefa é efetuada por uma variedade de mecanismos denominados esquemas de controle de concorrência que garantem a serialização das transações ou, em outros casos, o relaxamento dessa serialização, proporcionando os vários níveis de isolamento.

As principais estratégias, utilizadas pelos SGBDs, para o controle de concorrência sãoos protocolos de bloqueio, ordenação por *timestamp*, técnicas de validação e esquemas de multiversão[Sil_99].

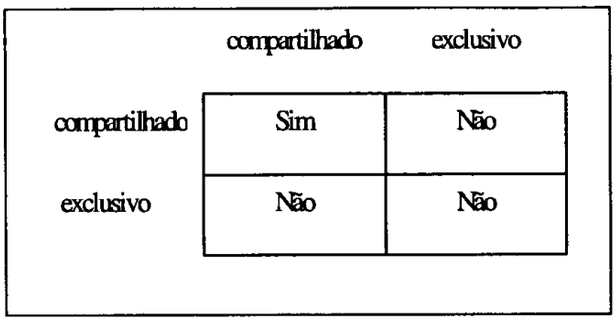
2.8.1 Protocolo baseado em bloqueio

A garantia da ordem de execução das transações poderá ser efetuada através de dispositivo de bloqueio de acesso aos itens de dados. Uma transação, antes de acessar um item de dado, deverá verificar se ele já não está sendo utilizado por outra transação. Dependendo do tipo de bloqueio do item de dado, outras transações não poderão modificá-lo, mas apenas utilizá-lo para leitura. Desse modo, o tipo de bloqueio permitirá o gerenciamento adequado entre as transações, ao possibilitar o uso alternado entre os itens de dados e evitar que alterações não desejáveis sejam armazenadas na base de dados. A liberação dos bloqueios efetuados numa transação ocorrerá numa operação de *commit*, além de fazer com que todos os objetos persistentes criados ou modificados pela transação sejam armazenados na base de dados. Todos os bloqueios solicitados deverão ser liberados, portanto, voltarão a estar acessíveis às outras transações. Numa operação de *checkpoint*, os bloqueios ficarão retidos e a transação continurá aberta, tal como é definido pelo modelo de transações ODMG 2.0.

O modo como são utilizados os protocolos de bloqueio define dois tipos de estratégias de controle de concorrência: as denominadas pessimistas e as otimistas [Kho_94]. As estratégias pessimistas trabalham com o pressuposto de que conflitos entre as transações irão ocorrer, então o bloqueio dos itens de dados é efetuado antes mesmo de acessá-los ou atualizá-los. As estratégias otimistas trabalham com a expectativa de que dificilmente ocorrerão conflitos entre as transações, portanto os bloqueios só serão analisados quando os itens de dados modificados forem armazenados na base de dados.

Existem vários modos de bloqueio, dos quais os mais simples são os de leitura (compartilhado) e de gravação (exclusivo). Um item de dado com o bloqueio no modo de leitura poderá ser compartilhado por outras transações que também possam requisitá-lo no modo leitura. Um item de dado, cujo bloqueio especificado é o de gravação, poderá ser utilizado somente pela transação que originou o bloqueio. Nenhuma outra

transação poderá utilizá-lo nem mesmo para a leitura. A estratégia de controle de concorrência que utiliza desses dois tipos de bloqueios o faz com o auxílio de uma tabela, através do identificador do item verifica-se o seu tipo de bloqueio. Um bloqueio exclusivo fará a transação requisitante esperá-lo até que a transação ocupadora do item de dado libere-o ou o coloque no modo compartilhado. O modo compartilhado ou exclusivo dos bloqueios pode ser visualizado através de uma matriz de compatibilidade, como é mostrado na figura 2.2.



	compartilhado	exclusivo
compartilhado	Sim	Não
exclusivo	Não	Não

Figura 2.2: Matriz de compatibilidade de bloqueios

2.8.1.1 Bloqueio de duas fases

Uma estratégia de solicitação e liberação de bloqueios é a denominada de *protocolo de bloqueio de duas fases*[Sil_99]. Esse protocolo divide as etapas de requisição de bloqueios em duas fases, que são as seguintes:

- Fase de crescimento - nessa fase são efetuadas somente as solicitações de bloqueio por uma transação e não é permitida nenhuma liberação de bloqueio;
- Fase de encolhimento - nessa fase só se permite à transação liberar bloqueios. Nenhuma solicitação de bloqueio dessa transação será atendida.

Algumas variantes desse protocolo têm sido apresentadas, por exemplo: o protocolo de bloqueio severo de duas fases que permite a liberação de bloqueios obtidos por uma transação no modo exclusivo somente quando ela se efetivar; a outra variante é o protocolo de bloqueio rigoroso de duas fases que possibilita a liberação de todos os bloqueios efetuados por uma transação, mas somente quando a transação for efetivada. Os protocolos de bloqueio de duas fases garantem que as transações concorrentes sejam executadas no modo serializável. Se ocorrer a utilização de transações que não atendam a esse protocolo, alguma informação adicional deverá ser fornecida para que se estabeleça a escala de serialização de conflitos.

2.8.1.2 Bloqueio baseado em grafos

As informações necessárias para que ocorra a serialização de conflitos são impostas nesse protocolo de bloqueio através de uma ordem aplicada aos itens de dados[Sil_99]. Essa ordem pode ser física, lógica ou imposta somente para atender à serialização. Assim, para o conjunto de todos os itens de dados $D=\{d_1, d_2, \ldots, d_n\}$, se d_j antecede d_k (notação: $d_j \rightarrow d_k$), então qualquer transação que efetue acessos a esses dois itens deverá fazê-lo obedecendo essa ordem, ou seja, primeiro d_j, depois d_k. Tal ordem estabelece um grafo acíclico para o conjunto de itens de dados D e pode ser melhor aplicada se estabelecer restrições definidoras de um grafo acíclico do tipo árvore com raiz. Esse é apenas um dos casos mais simples de definição de bloqueio por grafos, no qual as restrições para uma transação T_i podem ser definidas do seguinte modo:

- T_i poderá efetuar o primeiro bloqueio em quaisquer itens de dados;
- Após o primeiro bloqueio, T_i só poderá bloquear um item de dado qualquer d_k, se e somente se o item de dado pai de d_k já tiver sido bloqueado por T_i;
- Os itens de dados podem ser desbloqueados a qualquer momento;
- Se T_i efetuar a seqüência: bloqueado e desbloqueado, para um item de dado, tal item não poderá mais ser novamente bloqueado pela transação.

Essas restrições garantem ao protocolo de bloqueio de árvores que os itens de dados sejam desbloqueados mais cedo do que nos protocolos de duas fases. Além disso, dispensa a necessidade de esquemas de controle de deadlock. No entanto, tal protocolo apresenta a desvantagem de poder manter bloqueado itens de dados não utilizados pela transação.

2.8.2 Protocolo baseado em *timestamp*

Nos protocolos de bloqueio baseados em *timestamp*[Sil_99][Elm_00], a ordem imposta às transações para alcançar a serialização é estabelecida por algum dispositivo de controle de tempo. Por exemplo, pode-se utilizar para isso o relógio do sistema ou um contador lógico, a ser incrementado durante o uso. Assim, uma transação T_1 que receba um *timestamp* $Ts(T_1)$ terá sempre, em relação a uma outra transação T_2 que entrou posteriormente no sistema, a seguinte ordem: $Ts(T_1) < Ts(T_2)$. A garantia de funcionamento da escala serial em que a transação T_1 antecede T_2 deve ser acrescida do *timestamp* para cada item de dado especificado em cada tipo de acesso. Desse modo, para um item D de dado qualquer, verificam-se os seguintes *timestamp* por tipo de acesso:

- W-*timestamp*(D) representa o maior *timestamp* de escrita para qualquer transação que envolva o item de dado D;
- R-*timestamp*(D) representa o maior *timestamp* de leitura para qualquer transação que envolva o item de dado D.

A comparação do *timestamp* da transação e o *timestamp* do item de dado, de acordo com o tipo de acesso requisitado pela transação, estabelecerá se a transação será desfeita ou se irá efetivar o bloqueio desejado com a serialidade de conflitos garantida.

2.8.3 Protocolo baseado em validação

Essa estratégia de bloqueio tem como objetivo minimizar o *overhead* provocado pelo controle de concorrência em ocasiões em que a freqüência de acessos de leitura prevalece sobre a escrita[Sil_99]. Nesses casos, com uma freqüência de operações de escrita muito baixa, verifica-se a situação próxima ao fato que, se forem executadas sem o controle de concorrência, poderiam deixar a base de dados num modo consistente. Portanto, a estratégia consistirá em retardar o teste de bloqueio para os itens de dados utilizados pela transação, de modo que, após efetuar várias ações de leitura e até mesmo de atualizações, é que será feita a validação de tais operações. Esse protocolo de bloqueio exige que toda transação passe pelas seguintes etapas: leitura, validação e escrita. Na etapa de leitura, inicia-se a execução da transação e suas operações de leitura e escrita são feitas em variáveis locais temporárias. Na etapa de validação, são efetuados os testes que verificam se as variáveis temporárias com as atualizações poderão ser escritas na base de dados e sem conflitos de serialização. Na etapa final de escrita, após o êxito da etapa de validação, efetivam-se as atualizações na base de dados. Caso o resultado da validação não tenha sido positivo, a transação será desfeita. Essa estratégia utiliza-se da técnica de *timestamp* para a definição da ordem de serialização entre as transações.

2.8.4 Granularidade múltipla

Algumas estratégias de bloqueio permitem que vários itens de dados sejam bloqueados de uma única vez, diferentemente das estratégias apresentadas até aqui, em que a solicitação de bloqueio é feita individualmente[Kim_92][Sil_99]. Por exemplo, existem situações nas quais interessa à transação atualizar todos os objetos de uma classe. Logo, o adequado seria que todo o grupo de objetos fosse bloqueado para a escrita, pois do contrário, haveria um maior dispêndio de tempo para a efetivação dos bloqueios individuais.

O conceito de granularidade[Elm_00] está vinculado à quantidade de itens de dados a ser considerado. Um *grânulo grosseiro* representa um item de dado de proporcões maiores e um *grânulo fino* um item de dados de proporções menores. Através de uma árvore pode-se representar os vários grânulos (fino ou grosseiro) e a hierarquia composta dos múltiplos níveis de granulação. Os grânulos finos estão aninhados como filhos de grânulos grosseiros pai.

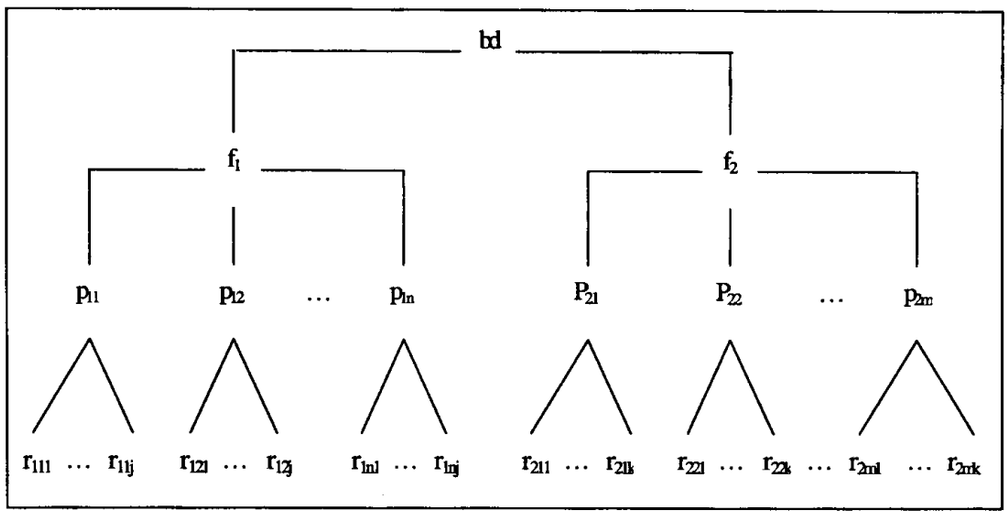


Figura 2.3: Hierarquia de granularidades para ilustrar múltiplos níveis de bloqueio

O exemplo apresentado na figura 2.3 mostra uma hierarquia de granularidades, na qual há uma base de dados composta de dois arquivos, cada arquivo contendo várias páginas, cada página com vários registros.

As estratégias de bloqueio de granularidade múltipla poderão utilizar-se, no modo mais simples, de bloqueios exclusivos ou compartilhados. Nesse caso, se um bloqueio exclusivo for solicitado para um nó da árvore, esse bloqueio se estenderá a todos os descendentes. O bloqueio de multigranularidade permite que transações diferentes estabeleçam bloqueios em diferentes níveis de granularidade. Por exemplo, ao utilizar a hierarquia da figura 2.3, supõe-se que uma transação T1 deseja efetuar modificações no arquivo f1. Nesse caso, o nó f1 é bloqueado no modo exclusivo pela transação T1 e conseqüentemente, todas as páginas e registros pertencentes aos nós descententes de f1 também serão bloqueados no mesmo modo. Se uma outra transação T2 desejar ler o registro r1nj, o gerenciador de bloqueios deverá navegar nos níveis acima desse nó, verificando-se se não ocorrerão conflitos de bloqueio. Assim, a concessão ou não desse pedido de bloqueio será resultante da análise dos bloqueios dos nós antecessores na estrutura de árvore. Essa estratégia poderá impor que outras transações tenham de ficar aguardando, caso necessitem efetuar o bloqueio em algum dos nós descendentes ao bloqueado. Essa situação poderá agravar-se quando o bloqueio for feito num nó raiz da árvore (grânulo grosseiro), pois todos os seus nós descendentes ficariam indisponíveis para outras transações, fato que implicará na redução da

concorrência. Além disso, a transação que quer utilizar-se de um nó descendente bloqueado deverá navegar através dos nós que o antecedem até chegar no nó explicitamente bloqueado, pois esta informação não é repassada aos descendentes de modo explícito.

Um modo de contornar esse impasse é pelo uso de uma classe de bloqueios, denominado bloqueio intencional, onde um nó que antecede o nó alvo poderá ser bloqueado no modo de intenção, ou seja, um bloqueio menos restritivo, e o nó desejado, numa granularidade mais fina, poderá ser bloqueado no modo explícito ou mais restrito. O propósito é propagar o efeito da restrição de bloqueio do grânulo mais fino para o grânulo mais grosseiro através do uso de um bloqueio de intenção adequado. Por exemplo, numa base de dados orientada a objetos, uma transação pode desejar bloquear no modo de intenção de leitura uma classe, para depois solicitar o bloqueio de leitura para uma instância dessa classe.

O bloqueio de intenção permite os seguintes modos de bloqueio:

S - compartilhado,

IS - intencional compartilhado,

X - exclusivo,

IX - intencional exclusivo,

SIX - compartilhado intencional exclusivo.

	IS	IX	S	SIX	X
IS	Sim	Sim	Sim	Sim	Não
IX	Sim	Sim	Não	Não	Não
S	Sim	Não	Não	Não	Não
SIX	Sim	Não	Não	Não	Não
X	Não	Não	Não	Não	Não

Figura 2.4: Matriz de compatibilidade para múltiplas granularidades de bloqueio

A figura 2.4 mostra a matriz compatibilidade entre os bloqueios. No caso, há um modo intencional associado ao bloqueio compartilhado e ao bloqueio exclusivo. Se um bloqueio intencional compartilhado é executado em um nó de uma árvore, então o

bloqueio explícito será propagado apenas para os nós descendentes e somente no modo compartilhado.

2.8.5 Esquemas multiversão

Os esquemas multiversão[Sil_99] utilizam-se da estratégia de criação de versões[2] para garantir o gerenciamento de conflitos de serialização. Por exemplo, num sistema que se utiliza de multiversões, para toda operação de escrita num item de dado qualquer D, é criada uma nova versão com a atualização do item D. Numa operação de leitura para o item de dado D, o sistema verifica quais das versões de D deverão ser disponibilizadas para essa operação. Assim, cabe ao gerenciador de concorrência tornar disponíveis as versões adequadas, ao evitar os conflitos de serialização e, de modo otimizado, satisfazer o desempenho do sistema.

Existem estratégias diferentes para esquemas multiversão, uma delas é a que se utiliza de ordenação por *timestamp*. Esse esquema multiversão associa um *timestamp* único e estático para cada transação e para cada versão de item de dado armazenam-se as seguintes informações:

- conteúdo - o valor da versão do item de dado;
- *W-timestamp* - é o *timestamp* da transação que criou a versão do item de dado;
- *R-timestamp* - é maior *timestamp* entre as transações que leram com êxito esta versão de item de dado;

Com a utilização pelas transações dos *timestamp* dos itens de dados, esses vão sendo atualizados de modo a permitir a existência de controle de conflitos de serialização.

Uma outra estratégia[Sil_99] combina as vantagens da multiversão e as vantagens do bloqueio de duas fases, onde as transações de atualizações executam um bloqueio rigoroso em duas fases, ou seja, os bloqueios são mantidos até a efetivação da transação. Portanto, a serialização ocorrerá baseada na ordem de efetivação das transações. Neste caso, o *timestamp* é único pra cada item de dado. A estratégia de multiversões é utilizada quando uma transação de atualização deseja escrever num item de dado. Nesse caso, efetiva-se o bloqueio para escrita nesse item de dado e executam-se as modificações numa versão, a qual receberá as alterações executadas pela transação. No caso de leitura por uma transação de atualização, um bloqueio

---

[2]Versão: cada uma das interpretações de um mesmo ponto[Aur_99].

compartilhado é imposto ao item e efetua-se a operação de leitura na última versão desse item.

2.9 Recuperação

Um sistema de gerenciamento de transações deverá permitir que todas transações possam utilizar-se das informações armazenadas numa base de dados de modo a deixá-las sempre num estado consistente. A atomicidade de um transação deverá ser preservada e, mesmo na ocorrência, de falhas a impossibilidade de término da transação deverá conduzir a uma base de dados consistente. Existe um conjunto de falhas que podem impedir que as transações se completem. Essas falhas poderão ser classificadas em três tipos [Kho_94][Sil_99]:

- **transação** - uma transação poderá ser impedida de se efetivar através de operações de *abort*; de conflitos de concorrência, por exemplo, *deadlock*; ou por violação a restrições de integridade;
- **queda do sistema** - problemas de mal funcionamento de hardware ou software poderão provocar a perda de informações da memória principal, situação que provocará a queda do sistema;
- **problemas no disco** - uma falha no sistema de armazenamento secundário (disco, etc.) poderá provocar a impossibilidade de transferência dos dados em blocos para a memória principal.

Os sistemas de recuperação executam uma das tarefas mais difíceis num SGBD, pois devem permitir que, na ocorrência de falhas, o sistema possa retomar o seu funcionamento em condições confiáveis, consistentes e factíveis. Para que os sistemas de recuperação possam desempenhar suas tarefas, oferecem-se algumas técnicas que possibilitam o suporte às suas atividades de modo adequado. Essas técnicas são descritas a seguir.

2.9.1 Recuperação baseada em *log*

Essa estratégia utiliza-se de arquivos que armazenam o histórico das alterações efetuadas pelas transações. Esses arquivos de histórico podem conter informações que representam o estado anterior e posterior de uma atualização, ao compor cada um dos registros desses arquivos. Com isto, a recuperação poderá recompor a base de dados no momento adequado, ao concretizar as atualizações ou desfazer-se através da leitura adequada dos registros de *log*. Uma técnica para a confecção do arquivo de *log* é a modificação imediata, na qual os registros de *log* são armazenados na base de dados

enquanto a transação ainda está ativa. Uma outra técnica é a modificação adiada. Nesse caso, os registros de *log* são confeccionados a cada atualização, mas são repassados para a base de dados quando a transação for efetivada.

Para tentar reduzir a sobrecarga provocada pela leitura de grandes arquivos de *log*, essas técnicas poderão utilizar-se de pontos de controle ou *checkpoint* que exigem a execução de uma seqüência de ações que colocam, de modo persistente, na base de dados, as atualizações feitas até o momento. Desse modo, se for necessária a recuperação, esta será efetuada a partir do último *checkpoint*. O modelo de transação ODMG2.0 oferece o *checkpoint* como uma das operações da classe transação, equivalente à operação de *commit* seguida de uma operação *begin*, menos os bloqueios solicitados pela transação que não serão liberados. Após o *checkpoint* a transação mantém-se aberta.

2.9.2 Páginas *shadow*

Essa técnica constitui uma alternativa às técnicas de construção de arquivos de *log*. Nesse caso, o fundamento básico é manter para cada atualização feita pela transação uma cópia da página que contém o item de dado atualizado. Assim, todos os itens de dados alterados terão a página original que o contém e uma cópia da mesma página, apenas contemplando a alteração efetuada. Essas duas páginas são mantidas em disco, sendo que apenas uma delas prevalecerá após o encerramento ou cancelamento da transação. Se a transação for encerrada com êxito, as páginas com as modificações é que deverão ser consideradas e as originais deverão ser liberadas, de modo que sejam recuperadas por algum esquema de coleta de lixo. Se a transação for cancelada, as páginas com as modificações é que deverão ficar disponíveis para a coleta de lixo, sendo que as páginas originais continuarão armazenadas em disco e consideradas pelo sistema como as que contém os itens de dados.

2.9.3 WAL - *Write Ahead Logging*

A técnica WAL[Moh_92] difere da técnica *shadow* porque efetua todas as atualizações no mesmo espaço em que está armazenado o item de dado lido. A estratégia WAL evita a duplicação de espaço para o armazenamento das informações atualizadas e, para que isto seja possível, deve-se assegurar que as informações antigas do ítem de dado atualizado sejam armazenadas num arquivo de *log*. Assim, a estratégia não permite que as atualizações sejam armazenadas permanentemente em disco, até que todas as informações necessárias a uma eventual operação de *undo* possam estar corretamente registradas e armazenadas num arquivo de *log*.

2.10 Indexação de objetos

As técnicas de indexação poderão ser divididas em estruturais e comportamentais[Ber_95]. As estruturais estão baseadas em atributos dos objetos e são as mais utilizadas, pois a maioria das linguagens de consulta orientadas a objetos utilizam-se dos valores de atributos para compor seus predicados. Essas técnicas de índices em geral utilizam-se da estrutura de dados B^+-*tree* para representar as diversas formas de predicados aninhados. Algumas dessas técnicas serão apresentadas a seguir. Antes, porém, será necessário expor algumas definições[Ber_94]:

- **Caminho** - é uma subarvore da hierarquia de agregação. Tem ínicio com uma classe C e termina com os atributos aninhados de C;
- **Instanciação de um caminho** - é a seqüência de objetos obtidos pela instanciação de classes pertencentes a uma caminho;
- **Índice aninhado**: é um índice que estabelece uma conexão direta entre o objeto do início e o objeto do final de uma instanciação de caminho. A chave de um índice é o objeto no final de uma instanciação dum caminho.

Para ilustrar as técnicas de índices apresentadas a seguir, consideram-se as seguintes classes escritas segundo o modelo de objetos ODMG2.0:

```
interface Projeto{
        attribute Companhia companhia_contratada;
        ....
}

interface Companhia{
        attribute Divisão divisões;
        ....
}

interface Divisão{
        attribute string cidade;
        ......
}
```

- **Multi-índices** - essa técnica baseia-se na definição de índices para os pares de caminhos contidos num determinado caminho. Por exemplo, considere o caminho formado pelos seguintes atributos:

  P = Projeto.companhia_contratada.divisões.cidade,

  Portanto, para este caminho P, três índices serão alocados

  I1 = Projeto.companhia_contratada;

  I2 = Companhia.divisões e

  I3 = Divisão.cidade

  Para I1, verificam-se as seguintes instâncias de caminho:

  (Companhia[k], {Projeto[i]})

  (Companhia[i], {Projeto[j], Projeto[l]})

  (Companhia[m], {Projeto[k]})

  Para I2, verificam-se as seguintes instâncias de caminho:

  (Divisão[h], {Companhia[i]})

  (Divisão[i], {Companhia[j]})

  (Divisão[k], {Companhia[k]})

  (Divisão[j], {Companhia[m]})

  Para I3, verificam-se as seguintes instâncias de caminho:

  (São Paulo, {Divisão[h]})

  (Campinas, {Divisão[i]}, {Divisão[j]})

  (Rio de Janeiro, {Divisão[k]})

- **Índices de junção** - esta técnica foi introduzida para suportar operações de junção para o modelo relacional e tem sido utilizada no suporte à implementação de objetos complexos. Dado um caminho, são obtidos os índices dos possíveis pares objeto.atributo desse caminho, no qual, para cada par, armazenam-se duas cópias de *clustering*. Para o caminho P, verificam-se os seguintes índices:

- Primeiro índice de junção sobre o par Projeto.companhia_contratada

  Uma cópia com *clustering* sobre o OId das instâncias de Companhia

(Companhia[k], {Projeto[i]})
(Companhia[i], {Projeto[j], Projeto[l]})
(Companhia[m], {Projeto[k]})

Outra cópia com *clustering* sobre o OId das instâncias de Projeto
(Projeto[i], {Companhia[k]})
(Projeto[j], {Companhia[i]})
(Projeto[l], {Companhia[i]})
(Projeto[k], {Companhia[m]})

- Segundo índice de junção sobre o par Companhia.divisões
  Uma cópia com *clustering* sobre o Oid das instâncias de Divisão
  (Divisão[h], {Companhia[i]})
  (Divisão[i], {Companhia[j]})
  (Divisão[k], {Companhia[k]})
  (Divisão[j], {Companhia[m]})

  Outra cópia com *clustering* sobre os OId das instâncias de Companhia
  (Companhia[i], {Divisão[h]})
  (Companhia[j], {Divisão[i]})
  (Companhia[k], {Divisão[k]})
  (Companhia[m], Divisão[j]})

- Terceiro índice de junção sobre o par Divisão.cidade
  Uma cópia com *clustering* sobreo o OId das instâncias de Divisão
  (Divisão[i], {Campinas})
  (Divisão[h], {São Paulo})
  (Divisão[j], {Campinas})
  (Divisão[k], {Rio de Janeiro})

  Outra cópia com *clustering* sobre os OId (valores) das instâncias de *string*
  (São Paulo, {Divisão[h]})
  (Campinas, {Divisão[i]}, {Divisão[j]})
  (Rio de Janeiro, {Divisão[k]})

- **Índices aninhados** - Com o objetivo de reduzir o número de índices a serem acessados, essa técnica permite que se especifique a associação entre um objeto de uma classe do final de um caminho e a correspondente instância da classe do início do caminho. Por exemplo, para o caminho

  P = Projeto.companhia_contratada.divisões.cidade

  os índices aninhados têm os seguintes valores:

  (São Paulo, {Projeto[j]})

  (Campinas, {Projeto[j], Projeto[k], Projeto[l]})

  (Rio de Janeiro, {Projeto[i]})

- **Índices de caminho** - é baseado num único índice, assim como em índices aninhados. A diferença com o índice aninhado é que o índice de caminho não armazena somente o valor chave do final do caminho e o objeto inicial, mas também as instâncias intermediárias dos caminhos, sejam eles completos, sejam eles parciais. Por exemplo, considerando-se o caminho P, verificam-se os seguintes valores do seu índice de caminho:

  (São Paulo, {Projeto[j].Companhia[i].Divisão[h]})

  (Campinas, {Projeto[j].Companhia[i].Divisão[i]},

  {Projeto[k].Companhia[m].Divisão[j]},

  {Projeto[l].Companhia[i].Divisão[i]})

  (Rio de Janeiro, {Projeto[i].Companhia[k].Divisão[k]})

A execução de uma consulta é efetuada através de buscas dos valores armazenados para os índices. Por exemplo, considere o seguinte predicado especificado para uma consulta: "Obter todos os projetos executados pela Companhia[i]", e supondo-se que a técnica de índices utilizada é índices de junção. Deve-se efetuar a busca através da chave Companhia[i] e obter todos os seus projetos, que são Projeto[j] e Projeto[l]. A escolha entre uma técnica e outra basea-se no tipo de consulta mais freqüentes efetuadas sobre o sistema que a utilizará. Assim, a escolha deverá recair sobre o menor custo entre ter um número maior ou menor de índices armazenados e executar o mínimo de buscas para atender as consultas efetuadas com maior freqüência pelas aplicações.

Algumas das técnicas de indexação de hierarquias de herança estão baseadas nas técnicas convencionais, nas quais os índices são construídos com o uso de atributos desejados em cada uma das classes componentes dos subgrafos da hierarquia. Uma outra solução é denominada de índice herdado[Ber_94], na qual mantém-se um índice para os atributos comuns a todas as classes pertencentes à hierarquia de herança, além

de uma entrada de índice com todos os identificadores de instâncias pertencentes às classes da hierarquia.

As técnicas de indexação comportamentais[Ber_94] estão baseadas na pré-computação dos resultados dos métodos requisitados pelas consultas. Os resultados dessa pré-computação são armazenados nos índices ou em outras estruturas de acesso, para que, posteriormente, possam ser utilizados. Essas técnicas deverão contornar o problema de se manter nos índices os resultados corretos e atuais da pré-computação dos métodos, mesmo após as possíveis atualizações nos atributos utilizados por tais métodos. Para que isso ocorra, os índices, nesses casos, conterão informações que minimizam a tarefa de mantê-los consistentes com as atualizações.

2.11 Considerações finais

Os conceitos e tecnologias apresentados no presente capítulo são resultantes dos esforços de pesquisadores e fabricantes que atuam no desenvolvimento de SGBDs, cujo objetivo é atender aos desafios e à demanda exigidas pelas aplicações do mundo real. O que se observa é uma crescente expansão das aplicações e o surgimento de novos tipos de dados a ser suportados pelos SGBDs. Tal situação vem pressionando os limites das tecnologias disponíveis nessa área. A manipulação de grandes volumes de informações; o armazenamento e a recuperação de dados multimídia; as novas tecnologias de distribuição de informações; as novas tecnologias de gerenciamento de transações; o suporte aos sistemas de apoio à decisão e *workflow,* têm exigido a elaboração de algoritmos, estruturas de dados e técnicas de armazenamento e recuperação complexas capazes de atender eficientemente a essas necessidades.

Capítulo 3

GERENCIADORES ORIENTADOS A OBJETOS

3.1 Considerações iniciais

Os Sistemas Gerenciadores de Base de Dados Orientados a Objetos -SGBDOOs vêm sendo desenvolvidos e apresentados à comunidade desde a segunda metade dos anos 80. Eles surgiram com o propósito de suportar aplicações mais complexas que as então usuais, como os sistemas CAD/CAM/CIM/CASE[1], baseados em conhecimento, multimídia etc.[Sil_96a][Sil_96b], uma vez que se achava consenso de que as tecnologias utilizadas pelos SGBDs tradicionais não conseguiam suportar de modo adequado essas aplicações.

Os SGBDOOs são baseados em algum modelo de dados orientado a objetos e permitem: a definição de classes com atributos e métodos; a definição de herança de atributos e métodos para as subclasses de uma superclasse; as instâncias duma classe possuírem identificadores únicos gerados pelo sistema, os quais permitem recuperá-las na base de dados; além de outras características apresentadas pelo paradigma de orientação a objeto.

---

[1]CIM - *Computer-Integred Manufacturing*. O desenvolvimento de sistemas CIM geralmente envolvem muitos departamentos e subsistemas especializados. Um exemplo ocorre quando um grupo de companhias que, rotineiramente não trabalham como uma unidade, se unem para atender determinados pedidos de clientes[Sil_96a];

No presente capítulo são apresentados alguns dos sistemas gerenciadores de base de dados orientados a objetos encontrados na literatura e no mercado. Esse levantamento bibliográfico é resumido, e contempla apenas alguns, dentre os vários SGBDOOs disponíveis. O objetivo é descrever as principais características operacionais oferecidas pelos SGBDOOs, suas arquiteturas, tecnologias utilizadas para o suporte a essas características e, em alguns casos, o ponto de vista a respeito das tecnologias adotadas pelos pesquisadores ou fabricantes.

3.2 ORION/ITASCA

O projeto ORION iniciou-se no final de 1985 junto ao *Advanced Computer Technology* (ACT - MCC, em Austin, Texas) e consistia no desenvolvimento de uma série de protótipos de gerenciadores orientados a objetos capazes de suportar a próxima geração de aplicações. Desde essa época, foram projetados e implementados os seguintes gerenciadores de objetos[Kim_90]: ORION-1, ORION-1SX e o ORION-2. Esses sistemas foram implementados em *common* LISP, numa máquina *symbolics* 3600 LISP e portáveis para estações de trabalho SUN/Unix. O ORION-1 é um sitema monousuário e multitarefas; o sitema ORION-1SX foi projetado para executar numa arquitetura cliente/servidor numa rede local de computadores; e o ORION-2 é um sistema de gerenciamento de objetos distribuídos, no qual todos os computadores numa rede participam do gerenciamento duma base de dados persistente e compartilhada. A arquitetura do ORION-2 não será abordada nesta descrição.

Uma versão do ORION está sendo comercializada desde 1995 pela IBEX Corporation[IBE_00], com o nome de ITASCA *Distributed Object Database Management System*. Esse sistema está disponível para plataformas UNIX, HP-UX, Sun Solaris, SunOS, Digital UNIX, SGI, AIX e Windows NT. O ambiente de desenvolvimento ITASCA necessita de no mínimo 16Mb de memória e 60Mb de espaço para *swap* onde executa o servidor.

3.2.1 Arquitetura

Nas figuras 3.1a e 3.1b são apresentadas as arquiteturas do ORION-1 e ORION-1SX, nos quais são mostrados os componentes principais desses sistemas. Em ambos, o manipulador de mensagens gerencia todas as mensagens recebidas pelos sistemas ORION. O subsistema de objetos efetua a otimização de consultas, o gerenciamento de esquemas,

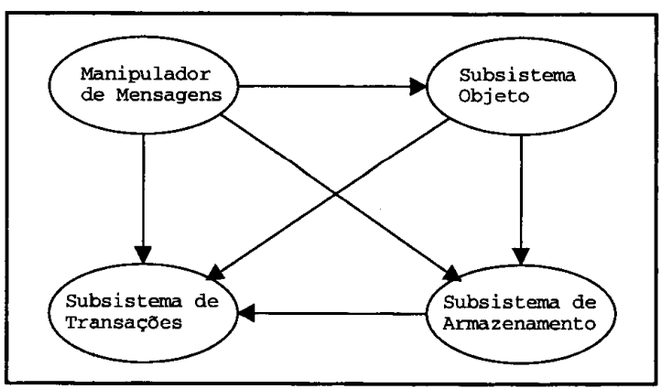


Figura 3.1a: Componentes funcionais dos sistemas ORION

o gerenciamento de dados longos (inclusive pesquisas em textos), suporte a versões de objetos, objetos compostos e objetos multimídia. O subsistema de gerenciamento de transações permite o acesso concorrente entre as transações através do gerenciamento de bloqueios e permite a recuperação com o uso de técnicas de *logging*. O subsistema de armazenamento persistente executa o armazenamento dos objetos e gerencia a transferência desses objetos entre o disco e os *buffers* de memória.

As arquiteturas ORION-1 e ORION-1SX são diferentes quanto ao fato de alguns subsistemas do ORION-1SX serem separados em processos cliente e servidor, enquanto que no ORION-1 todos os seus subsistemas são executados numa única máquina. No caso do ORION-1SX, essa separação ocorre nos seguintes subsistemas: espaços de endereçamento de memória, esquema de gerenciamento *dual-buffer*, *caching* de objetos e estruturas de dados do sistema.

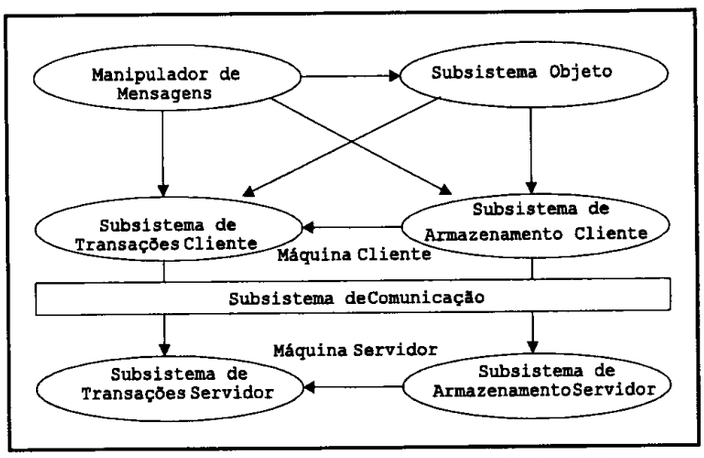


Figura 3.1b: Arquitetura do sistema ORION-1SX

3.2.2 Gerenciamento de *buffers*

O ORION utiliza-se dum refinamento do esquema denominado de *dual-buffer* para o gerenciamento de *buffers*, que consiste da partição da área de memória disponível para os *buffers* em duas porções. Uma compondo o gerenciador de *buffers* para as páginas e a outra porção, o gerenciador de *buffers* para os objetos. O objetivo do uso desse esquema é proporcionar um suporte mais eficiente às aplicações consideradas mais sofisticadas (não convencionais), quanto ao gerenciamento de *buffers*, que os oferecidos pelos SGBD's tradicionais. Tais aplicações centralizam suas atividades nos objetos e não nas páginas. Portanto, para o suporte às aplicações torna-se disponível o gerenciador de *buffers* que trabalha somente com os objetos. A disponibilidade de um objeto num *buffer* de objeto é possível pelo prévio oferecimento da página que o contém no gerenciador de *buffer* de páginas. O objeto desejado é localizado, recuperado e enviado ao gerenciador de *buffer* de objetos que o torna disponível para a aplicação.

O gerenciador de *buffers* para objetos mantém uma estrutura composta por uma Tabela de Objetos Residentes (ROT) que permite localizar os objetos no conjunto de *buffers* de objetos. Um objeto entrará para esse conjunto de *buffers* após operação de criação ou recuperação. Entre o objeto de fato e o ROT existe outra estrutura denominada Descritor do Objeto Residente (ROD) que contém um conjunto de informações (Identificador Lógico Único - UID do objeto, ponteiro para o ROD da

classe, Identificador da Localização Física - PID, um bloqueio de *cache* etc.) de cada objeto, um Número de Validação de Objeto - OVN que permite verificar se o objeto, para o qual foi requisitado um bloqueio, possui suas inforamações consistentes com o que está armazenado no servidor. A figura3.2 ilustra as estruturas ROT e ROD.

O gerenciamento de transação dispõe de mecanismos para assegurar a consistência dos objetos na memória, uma vez que estão alocados nos dois gerenciadores de *buffers* e podem conter diferentes cópias do mesmo objeto.

3.2.3 Identificador de objetos

O identificador de objetos no ORION é composto pelo par *<identificador de classe, identificador de instância>*, de tal modo que o identificador de classe é a referência à classe a que pertence o objeto e o identificador de instância é um identificador único na base de dados toda para a instância. A esse par denomina-se

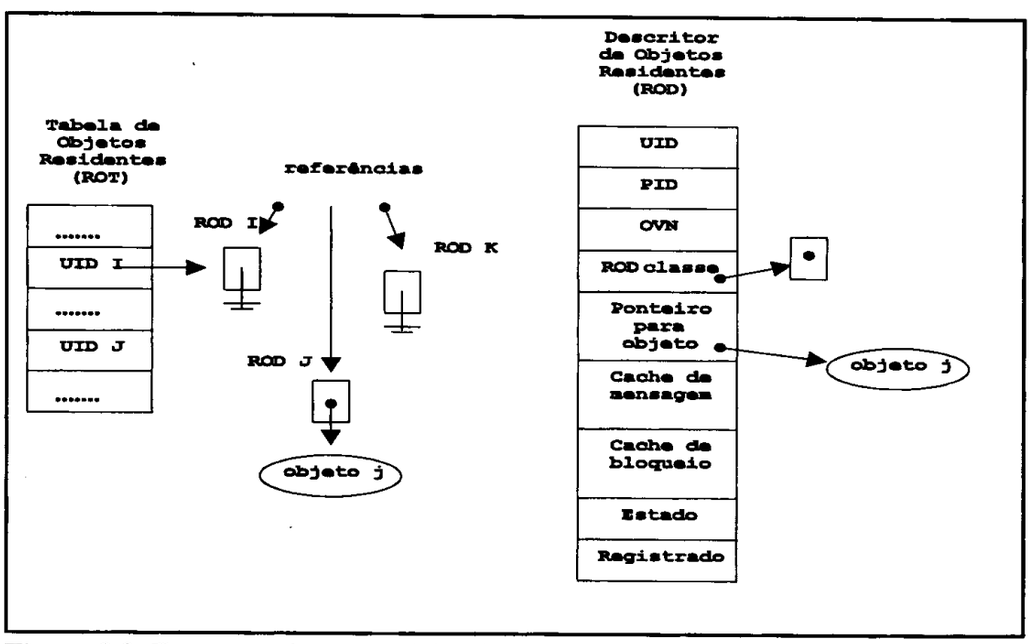


Figura 3.2: Estruturas de suporte à localização de objetos nos *buffers*

Identificador Lógico Único - UID. A adoção do UID nesse formato proporciona algumas vantagens, por exemplo, quando uma mensagem é enviada a um objeto, não há a necessidade de recuperá-lo para verificar se tal mensagem é válida. Essa verificação é efetuada diretamente com o acesso à classe desse objeto, sem necessidade de tê-lo disponível. Uma desvantagem é que a migração do objeto duma classe para outra passa a ter um custo alto, pois, quando isso ocorre, todas as referências UID de um objeto nessa circunstância deveriam ser atualizadas, ou seja, deve-se modificar o seu novo identificador de classe.

3.2.4 Objetos compostos

O ORION suporta um modelo de hierarquia com dependências exclusivas para os objetos componentes. Um objeto componente não pode ser compartilhado por outro objeto, ou seja, não pode ser componente de mais de um objeto composto. Além disso, sua existência depende da existência do objeto que o contém. Esse último caso implica em que, quando um objeto constituinte de um objeto composto é eliminado, todos os objetos dependentes dele também devem ser eliminados.

3.2.5 Armazenamento e páginas de objetos.

Um objeto no ORION contém informações a respeito de seu tamanho total, o número de atributos que o compõem, vetor de identificadores dos atributos nele representados (não há valores *default*), vetor de referências aos valores de seus atributos na área de valores do próprio objeto e os valores dos atributos. Esses itens componentes das informações de um objeto estão representados na figura 3.3.

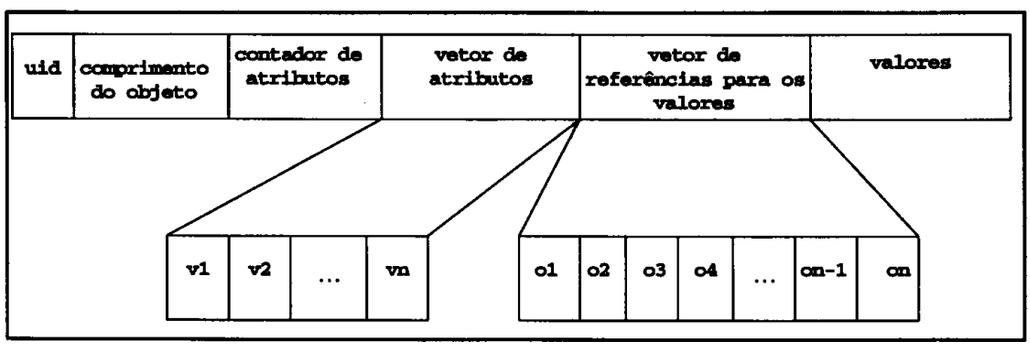


Figura 3.3: Formato de armazenamento de objetos no disco

O valor de atributo de um objeto pode ser definido pelo usuário ou pelo sistema. No primeiro caso, pode ser um, ou um conjunto de valores primitivos (inteiros, cadeia de caracteres etc.) ou referências UID's. No segundo tipo de possíveis valores de atributos tem-se o número da versão, o *timestamp* atualizado para os objetos *versionables*[2] e objetos cuja atualização deve ser notificada.

Um disco utilizado pelo ORION é composto por um conjunto de partições divididas em segmentos que contêm páginas ou blocos. Uma página de objetos, representada na figura3.4, é composta por um *cabeçalho*, vetor de ponteiros indiretos e o espaço para o armazenamentos dos objetos. Nessa ilustração, têm-se no vetor de endereços indiretos, os locais na página onde estão os objetos Z, A, F e B. O *cabeçalho* contém informações como o número de objetos na página, total de espaços livres, total de espaços contíguos livres, referências para os espaços livres etc.

---

[2]*Versionable* ou *non-versionable*: pode ou não ter versões.

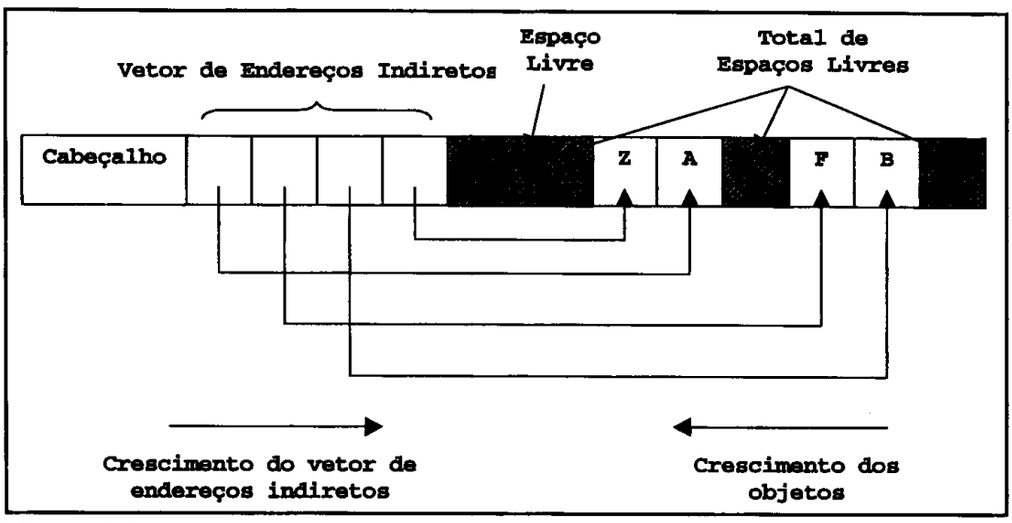


Figura 3.4: Formatação para uma página nos sistemas ORION

Ao ser armazenado um objeto numa página são promovidas alterações no cabeçalho e no vetor de endereços indiretos. Esse último receberá em seu respectivo índice um endereço de entrada na página. Em cada uma dessas entradas estarão armazenados os endereços do local, na página, onde o objeto se inicia. Esse artifício permite que um objeto possa ter sua localização alterada, modificando-se apenas o endereço da respectiva entrada. Isso faz com que seja otimizado o trabalho de propagação das alterações conseqüentes da realocação dum objeto.

3.2.6 Armazenamento das classes de objetos

As estruturas que armazenam as informações a respeito das classes definidas num esquema contêm dois tipos de informações: uma é a especificação dos atributos e métodos da classe (definidos pelos usuários ou pelo sistema); outra é a especificação dos atributos e métodos aplicados a todas as instâncias da classe. Um exemplo para esse último caso são os atributos que executam propriedades de agregados em todas as instâncias da classe (média aritmética, total etc.).

3.2.7 *Clustering*

O ORION efetua o agrupamento físico de informações relacionadas entre si por meio da manutenção num mesmo segmento físico dos objetos pertencentes a uma mesma classe. Também são mantidos num mesmo segmento físico as instâncias duma mesma coleção especificadas pelo usuário para uma classe.

3.2.8 Versões e notificação de alterações

O modelo de versões do ORION considera a existência de três tipos de versões: *transiente*, *working* e *release*. Uma versão *transient* ou versão temporária pode ser atualizada ou eliminada somente por quem a criou. Uma versão *working* ou versão estável não pode ser atualizada e somente será eliminada por quem a criou. Uma versão *release* não pode ser mais atualizada ou eliminada por qualquer usuário.

Uma versão *transiente* pode ser criada pela primeira vez derivada duma versão *release* (através duma operação *check-out*), *transiente* ou *working*. Várias versões *transientes* podem ser obtidas a partir de versão existente ao longo do tempo, criando-se uma hierarquia de derivação de versões. Uma versão *transiente* pode ser promovida a versão *working*. Uma versão *working* pode ser promovida (ou sofrer uma operação de *check-in*) a versão *release*. Essas etapas que as versões podem apresentar ao longo do tempo numa base de dados são apresentadas na figura3.5. O algoritmo que controla a atualização de versões *working* deve restringir essa operação de modo a impedir que as suas versões *transientes* sejam afetadas por essas modificações.

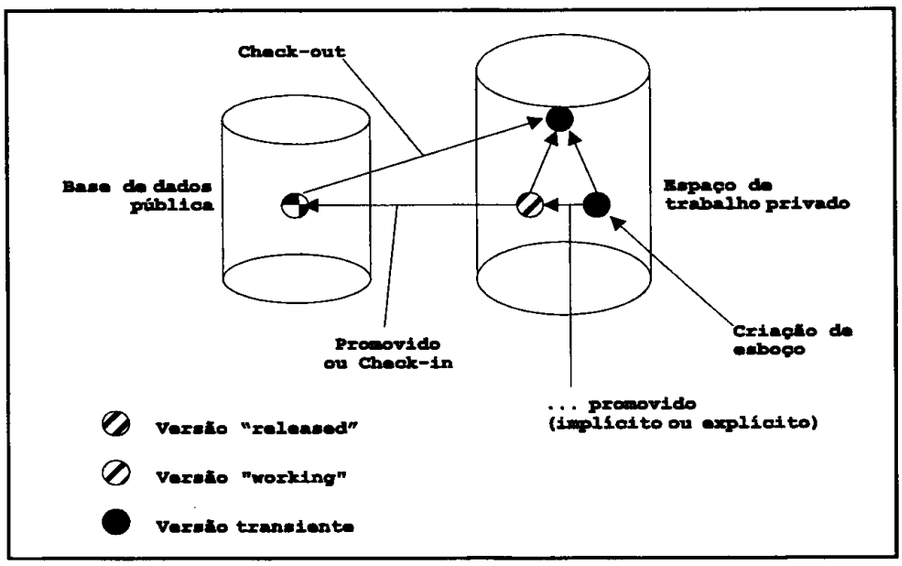


Figura 3.5: Estados possíveis para as versões de objetos no ORION

Os objetos são distinguidos em *versionable* e *non-versionable*. Um objeto *versionable*, também denominado objeto genérico, é uma instância duma classe *versionable*. No objeto genérico está representada a hierarquia de derivação de versões, na qual cada instância específica (cada uma contém seu OId) representa uma versão do objeto *versionable* (que também tem seu OId). Além do OId, um objeto genérico possui um conjunto de atributos definidos pelo sistema. São eles: contador do número total de instâncias dessa versão, número a ser atribuído à próxima instância de versão, número da versão utilizada como versão *default* entre as instâncias do objeto genérico, chave indicando se a versão *default* foi definida pelo usuário e o descritor da árvore de derivação do objeto genérico. O descritor da árvore de derivação contém uma árvore de descritores de versão. Cada descritor de instância de versão contém as seguintes

informações: números de versão usados para distinguir entre instâncias de versões num mesmo objeto *versionable*; o tipo de versão que pode ser *transiente* ou *working* e o OId do seu objeto genérico.

Os sistemas ORION suportam dois modos para a notificação de alterações[Ber_94], denominados de baseado em mensagens e em *flags*. No caso da abordagem baseada em mensagens, as alterações são comunicadas através de mensagens enviadas para o usuário (uma pessoa ou uma aplicação). O momento de envio dessa mensagem é definido pelo usuário e pode ser imediato ou retardado. No caso da abordagem baseada em *flag*, todo objeto notificado da ocorrência de uma alteração recebe dois *timestamps* distintos, um denominado *timestamp* de notificação de alteração, utilizado para representar o momento da criação ou da última atualização do objeto; o outro, denominado *timestamp* de aprovação da alteração, representa o momento em que o proprietário do objeto aprovou todas as alterações associadas a esse determinado objeto. Através de programas utilitários e do suporte oferecido pela notificação de alterações, o usuário (ou a aplicação) pode determinar referências inconsistentes num objeto complexo.

3.2.9 Evolução de esquemas

O ORION utiliza-se do mecanismo denominado atualização retardada no suporte à evolução dinâmica de esquemas. Essa abordagem é a mesma utilizada em SGBDs relacionais, como, por exemplo, o SQL/*Data Systems* da IBM. A estratégia utilizada por essa abordagem permite esconder informações obsoletas de uma instância da base de dados, logicamente atingida pela alteração no esquema, até que seja necessário proceder-se à recuperação da página que a contém. Isso evita sobrecarregar o sistema na tarefa de leitura de registros para efetuar as atualizações, distribuindo a carga de processamento ao longo das operações subseqüentes, ou seja, quando as páginas tiverem de ser necessariamente recuperadas para a leitura de objetos. Por exemplo, se uma classe *Documento* tem seu atributo *data_de_criação* eliminado, uma instância do objeto *Documento* terá esse atributo omitido (e eliminado) na recuperação. Como a operação de atualização de todas as instâncias envolvidas nessa modificação do esquema é operacionalmente custosa, a abordagem adotada opta por pulverizar esse custo entre as operações de recuperação subseqüentes. Desvantagem dessa abordagem é que as informações obsoletas ficam armazenadas na base de dados até que ocorra a primeira vez em que as páginas, onde estão armazenadas essas informações, tenham de ser recuperadas.

3.2.10 Indexação

O ORION usa o esquema de indexação de hierarquia de classes implementando uma estrutura similar a uma árvore-B^+. O domínio de um atributo pode ser uma classe primitiva (por exemplo, um inteiro ou uma *string*) ou alguma classe definida pelo usuário. Além disso, o valor de uma chave em um índice pode ser um identificador de objeto, algum valor primitivo ou um conjunto de valores primitivos ou de

identificadores de objetos. O ORION suporta somente a indexação através de atributos simples, isto é, permite ao usuário criar índices utilizando apenas um atributo simples.

3.2.11 Gerenciamento de transações

O Gerenciador de Transações (GT) do ORION permite a execução de transações serializáveis, ou seja, várias transações podem executar concorrentemente sem que uma transação possa provocar algum efeito nas demais. Isto é válido para todas as transações que atuam de modo concorrente. O GT contém o módulo gerenciador de bloqueios, que mantém uma tabela de bloqueio para as transações e outra tabela contendo os estados das transações (ativas ou suspensas). A tabela de bloqueio armazena o tipo de bloqueio aplicado ao objeto sob influência da respectiva transação. Através do valor de bloqueio é possível determinar se um objeto poderá ser recuperado ou atualizado pelas transações. O conjunto de bloqueios sob o comando de determinada transação é liberado após a operação de *commit* ou *abort*.

O sistema ORION-1SX utiliza-se da estrutura cliente/servidor para distribuir as tarefas do GT. Assim, o GT possui um gerenciador de bloqueios que mantém um *cache* de bloqueio no cliente e a mesma estrutura no servidor. Quando um bloqueio é requisitado para o servidor, esse o executa e o transfere para o cliente que, em seguida, o registra na entrada adequada do ROD do objeto bloqueado. A partir daí, em toda requisição feita ao objeto, é verificado o tipo de bloqueio no cliente contra o bloqueio solicitado pela requisição. Se o poder do bloqueio da requisição for inferior, não será efetuado qualquer acesso ao servidor.

O GT contém um gerenciador de impasses (*deadlock*) que atua em segundo plano (*background*), com o objetivo de detectar conflitos de impasses entre as transações e determinar soluções para essas situações. A estratégia adotada pelo gerenciador de impasses é baseada no uso de *timeout*, ou seja, é atribuído um tempo de tolerância para a duração do impasse. Se esse tempo expira, a transação suspensa torna-se candidata a cancelar a ação provocadora do impasse.

3.2.12 Controle de concorrência

O suporte ao controle de concorrência é garantido pelas políticas de bloqueio adotadas pelos sistemas ORION. Essas políticas atuam em três tipos de hierarquias, a saber: a hierarquia de granularidade para as entidades lógicas, a hierarquia de classes e objetos compostos e a hierarquia composta pela árvore-B^+ no suporte às consultas.

O primeiro bloqueio ocorre no nível físico, através das páginas que contêm as entidades lógicas. Posteriormente, é fixado o bloqueio lógico dos nós pelo gerenciador de bloqueios (ou grânulos bloqueáveis numa hierarquia de granularidades), os quais podem estar em uma das hierarquias anteriormente descritas. Assim, tanto o bloqueio físico quanto o lógico são efetuados através de acessos às páginas, pois essas contêm as unidades lógicas.

Um bloqueio deve ser capaz de garantir que, quando determinada transação bloqueia uma classe, todas as subclasses descendentes da classe acessada também devem ser bloqueadas. Por exemplo, quando é efetuada uma consulta numa instância de classe, os descendentes na hierarquia não podem estar sendo modificados por outras transações. A política de bloqueio adotada pelo ORION evita tais situações, mas o faz ao impor alguns limites, pois o bloqueio duma hierarquia pode ser uma operação custosa. Os sistemas ORION utilizam-se da política de bloqueio de duas fases, protocolo que separa rigidamente a transação em fase de crescimento da fase de redução. Durante a fase de crescimento são obtidos todos os bloqueios e, durante a fase de redução, são liberados todos os bloqueios. Em ambas fases, tanto a obtenção quanto a liberação são efetuadas preferencialmente de forma incremental. O ORION utiliza-se de bloqueios de intenção, onde são reconhecidos cinco tipos de bloqueios divididos nos grânulos: instâncias de objetos e classes de objetos. Os bloqueios estão divididos por granularidade do seguinte modo:

Bloqueio para objetos instâncias:

S bloqueia uma instância de objeto no modo *Compartilhado* para leitura;
X bloqueia uma instância de objeto no modo *Exclusivo* para atualização;

Bloqueio para objetos classes:

IS bloqueia uma definição de classe no modo *Intencional Compartilhada* para leitura. Bloqueia objetos instância no modo S;

IX bloqueia uma definição de classe no modo de *Intencional Exclusiva* para atualização e objetos instância no modo X;

S bloqueia uma definição de classe para leitura e implicitamente bloqueia instâncias da classe no modo S;

X bloqueia uma definição de classe para atualização e implicitamente bloqueia instâncias da classe no modo X;

SIX bloqueia uma definição de classe para leitura. Implicitamente bloqueia instâncias da classe no modo S. Explicitamente bloqueia instâncias para serem atualizadas no modo X.

3.2.13 Recuperação

O ORION utiliza o esquema de arquivos de *log* para suportar a recuperação de falhas de software e operações de cancelamento de transações (*abort)*. Esse esquema foi escolhido devido à dificuldade em suportar o controle de concorrência de múltiplas transações numa estrutura de duplicação de páginas (*shadow*). Um arquivo de *log-undo*, ou seja, que permite desfazer as atualizações através de operação de *undo*, é mantido pelo sistema. Embora existam outras opções de arquivos *log*, foi esse o

adotado, segundo os autores, por sua simplicidade de implementação. Esse esquema exige que as páginas contendo a versão anterior dos objetos atualizados sejam armazenadas em disco (numa operação de *commit*), pois que, numa operação *undo,* esses antigos objetos devem ser recuperados. Esse processo acha-se em sintonia com o protocolo *Write Ahead Logging*-WAL.

O ORION efetua a recuperação de objetos multimídia também através do esquema de arquivo de *log*. Um objeto multimídia no ORION é composto por dois elementos: o próprio dado multimídia e o seu descritor. O sistema mantêm uma lista de blocos livres que são alocados para o armazenamento dos dados multimídia. O suporte à recuperação desse tipo de dado é proporcionado pela manutenção dum arquivo de *log* das alterações da lista de blocos livres e do descritor. Assim, numa operação de cancelamento para uma transação que criou um dado multimídia, os blocos que os contêm retornará à lista de blocos livres e o descritor conterá o valor nulo como sua referência.

3.3 O Sistema O_2

O sistema O_2[Deu_90][Deu_91][Zan_95] foi construído, no período de cinco anos, num projeto denominado Altair, cujo início ocorreu em 1986. O Objetivo desse projeto foi implementar um sistema gerenciador de base de dados que atendesse aos requisitos das então denominadas "novas aplicações", por exemplo, os sistemas CAD/CAM, sistemas geográficos e urbanos, sistemas de informação editorial e automação de escritório. O O_2 também está habilitado a suportar as aplicações na área de negócios e aplicações transacionais para instituições financeiras, bancos e companhias de seguro. A construção do Sistema O_2 dividiu-se em duas fases: a primeira, de 3 anos de duração, consistiu na construção dum protótipo do sistema; a segunda, de 2 anos, a fase de desenvolvimento do sistema.

O sistema O_2 está disponível para a maioria das plataformas UNIX (Sun, Hewlett-Packard, IBM, Digital Equipment Corporation, Silicon Graphics, BULL, Siemens) e plataformas Intel (Windows 95, Windows NT, Solaris, SCO e NextStep)[OOT_00].

3.3.1 Arquitetura do O_2

O sistema gerenciador de base de dados O_2 é composto pelos módulos especificados na figura3.6, a saber[OOT_97]:

- O gerenciador de objetos O_2Engine - implementa um sistema de gerenciamento de banco de dados baseado no padrão de modelo de dados ODMG. As aplicações podem ser desenvolvidas sobre o O_2Engine com C, C++, Smalltalk, Java, O_2C, uma Linguagem de 4a. Geração e a Linguagem de Consulta para Objetos - OQL - segundo o padrão ODMG;

- O ambiente de programação (RAD *Tools*), que inclui O_2Look, um construtor de interfaces gráficas para objetos, o O_2Graph - um pacote de manipulação gráfica, O_2Report - um gerador de relatórios para objetos O_2 e o O_2Tools, um ambiente de desenvolvimento gráfico;

- Um conjunto de ferramentas de comunicação:

  - O_2ODBC, possibilita que as aplicações ODBC sejam executadas sobre banco de dados O_2;
  - O_2DBAcces permite a construção de aplicações que se intercomunicam com banco de dados relacionais;
  - O_2Corba é um adaptador de objetos para Corba;
  - O_2Web possibilita a conexão de banco de dados O_2 na Web.

O O_2Engine é descrito com mais detalhes nas seção 3.3.2 a seguir.

3.3.2 O_2Engine

O O_2Engine, como está representado na figura 3.7, é um sistema de objetos e de base de dados. É um sistema de objetos porque suporta o padrão ODMG, contemplando características como objetos complexos que contêm identificadores, relacionamentos, coleções, classes, tipos, encapsulamento, métodos, herança simples e múltipla, sobrecarga de métodos e operadores e acoplamento tardio de métodos. É um sistema de base de dados porque proporciona o suporte de modo confiável, seguro e compartilhado para o acesso, armazenamento e atualização de grandes quantidades de dados.

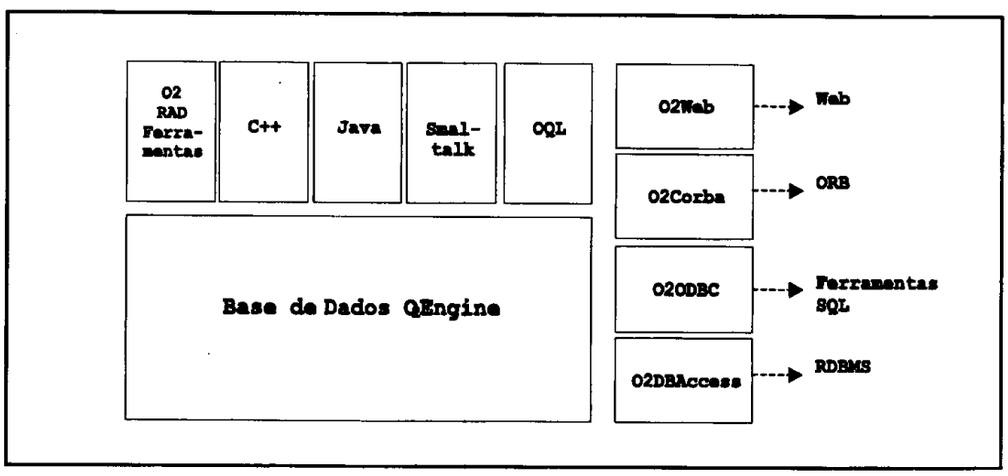


Figura 3.6: Arquitetura do sistema gerenciador de base de dados O_2

As camadas que compõem O_2Engine são:

a) O_2Store, ou o gerenciador de memória, é a camada mais inferior do O_2Engine e cuida do armazenamento e recuperação dos dados através dos dispositivos de armazenamento secundário. O O_2Store gerencia o armazenamento através dum conjunto de depósitos denominados "volumes O_2", no qual os arquivos estão alocados e, em cada um, estão distribuídas informações como: arquivo de *log* para o suporte à recuperação, esquema de duplicação de páginas, catálogo[3] e os volumes pertinentes a cada um dos vários usuários. A arquitetura cliente/servidor do O_2Store aparece na figura 3.8

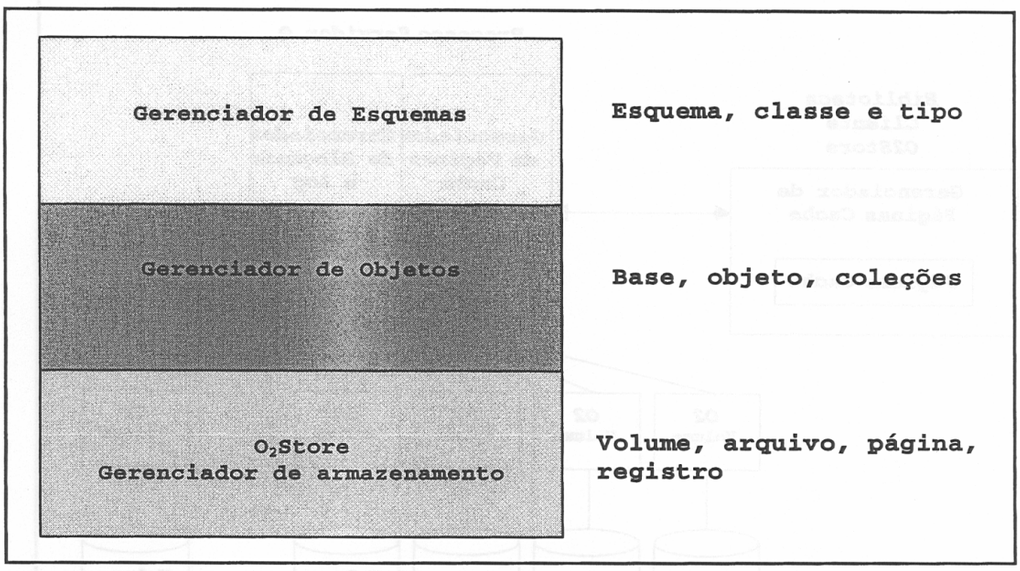


Figura 3.7: Arquitetura do O_2Engine

O esquema de endereçamento dos registros do O_2Store manipula arquivos seqüenciais onde são alocados os registros utilizados pelo sistema O_2. Os registros podem ter tamanho variável e alocados ou eliminados em qualquer posição num arquivo. A identificação dum registro é efetuada através do seu RID - *Record Identifier*, ou seja, o OId do registro e é composto por um conjunto de 8 bytes dispostos do seguinte modo:

2 bytes para a identificação do volume;
4 bytes para a identificação da página num volume;
2 bytes para a identificação duma entrada (*slot*) numa página;

Através desse sistema de endereçamento, é possível que determinado registro tenha a capacidade de crescer ou diminuir com certo grau de liberdade. Se um registro tiver seu tamanho alterado, de modo que a página onde se encontra não mais consiga

[3]Catálogo: armazena os dados que descrevem cada base de dados. A esses dados também se denomina de *meta-dados*. Inclui a descrição do esquema conceitual da base de dados, o esquema interno, quaisquer esquemas externos e o mapeamento entre os esquemas de diferentes níveis[Elm_00]

suportar esse crescimento, ele poderá ser deslocado para outra página com condições de recebê-lo. Para isso, o O_2Store utiliza-se de dispositivo que permite tal deslocamento sem que haja alteração do seu identificador. Esse processo é implementado pela atualização do conteúdo do *slot*, de modo a conter a identificação do novo local onde se encontra o registro, ou seja o *slot* do registro deverá conter o endereço do *slot* da nova página denominado *fowarding marker*[Gam_95]. A figura3.9 mostra como esse processo de referenciação é executado. O custo desse procedimento é acrescido de mais um acesso, pois ao posicionar na página antiga, um outro acesso deverá ser executado para a nova página.

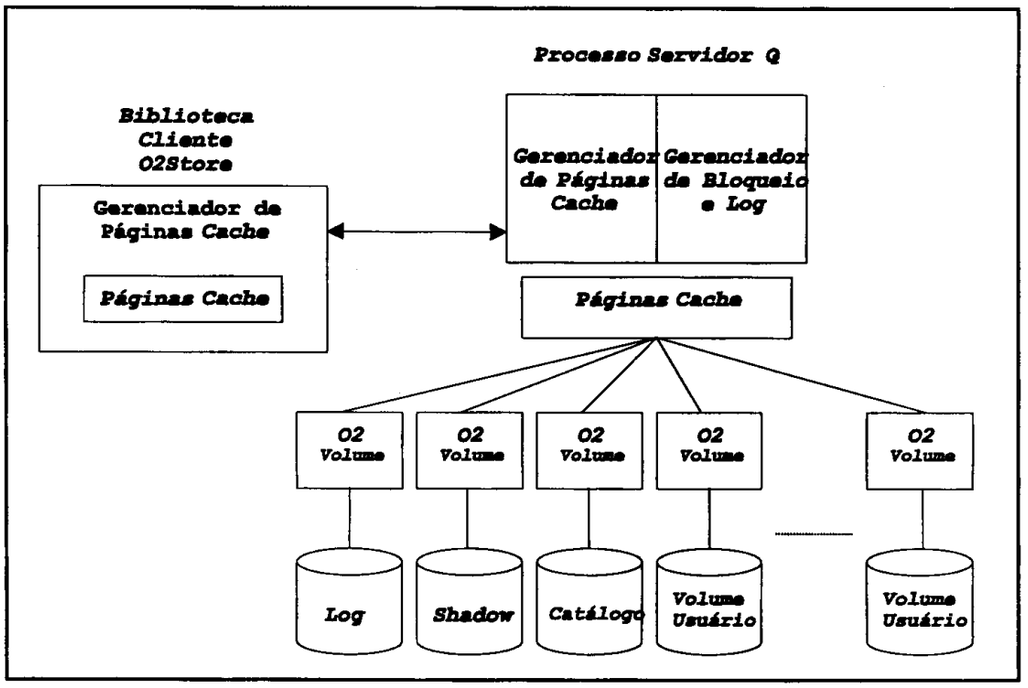


Figura 3.8: Arquitetura cliente/servidor do O_2Store

Como pode-se notar, o O_2 não utiliza a estratégia de endereçamento lógico para seus OId's mas, sim, o endereçamento físico e, através desses, localiza diretamente os objetos e valores no disco. Essa forma de endereçamento tem por objetivo evitar que se mantenha de modo persistente tabelas de correspondência grandes com uma entrada para cada objeto, de modo a associá-lo ao seu endereço físico. Como os objetos podem corresponder a dados bastante pequenos e em grande número, a quantidade de operações de procura de objetos por seu OId poderá ser muito grande.

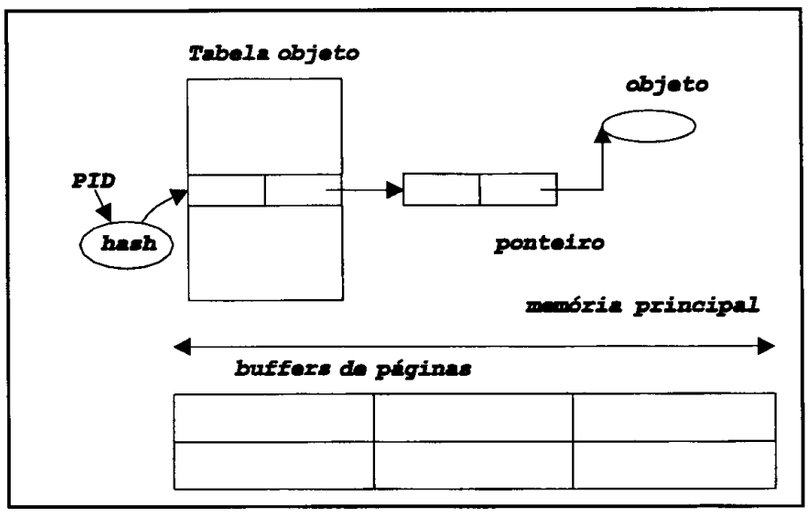


Figura 3.9: Contendo a movimentação (deslocamento) dum registro nas páginas

O O_2 utiliza-se duma tabela de espalhamento (*hash*) para armazenar os objetos usados pelas aplicações, ou seja, para o gerenciamento dos objetos transientes[Gam_95]. Essa estrutura de endereçamento de objetos transientes é apresentada na figura 3.10. Todo objeto que se encontra na memória principal é acessado através de endereço nessa tabela de espalhamento. Quando o objeto é requisitado pela primeira vez, deve-se criar uma entrada (*slot*) para esse novo objeto. Portanto, o objeto é obtido na memória secundária, seu conteúdo copiado na memória principal e um *slot* fará a associação do Identificador Persistente (PID - *Persistent Identifier*) que nada mais é que o RID, com o endereço da imagem (cópia) do objeto. Esse processo, em que se utiliza da tabela de espalhamento, não efetua a conversão do OId em endereços de memória virtual, mas sim a associação com um endereço local. A essa operação denomina-se desreferenciação ou *swizzling*. Tal estratégia adotada pelo O_2, permite que as transações possam efetuar o compartilhamento em disco e as atualizações tornem-se persistentes somente após uma operação de encerramento de transação, ou seja, *commit*.

Os acessos aos registros RID's em disco são facilitados pelo uso de estruturas de árvores-B associadas aos arquivos. O O_2 utiliza-se de dois tipos de estruturas de árvores-B, os tipos padrão e posicional. Essa última tem a característica de considerar as informações dos arquivos como lista ordenada, na qual, como num vetor, um registro pode ser inserido ou retirado diretamente na posição relativa desejada. Nos nós duma árvore-B posicional, as chaves são os endereços onde se encontram os elementos da lista ordenada.

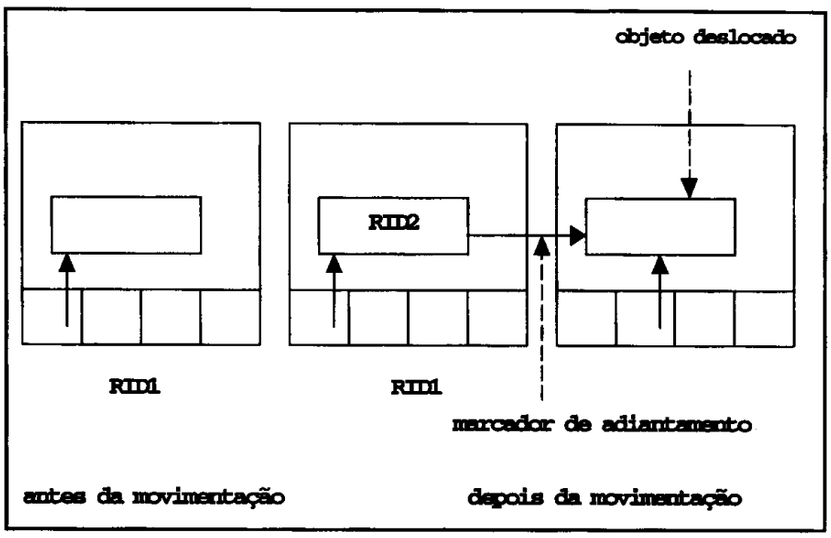


Figura 3.10: Estrutura para alocação de objetos transientes no O_2.

O O_2Store é responsável também pelo esquema de segurança do O_2. Esse esquema é implementado considerando-se uma arquitetura cliente/servidor baseada num servidor de páginas, cujo conteúdo semântico das páginas não é interpretado pelo servidor[Dew_90]. São utilizados os registros de *log*, *on-line* ou *off-line*, gerados pelo mecanismo de recuperação para se efetuar o *backup* incremental de informações. No caso do *backup on-line,* esse é efetuado de modo a assegurar que as informações armazenadas mantenham-se consistentes, embora obtidas enquanto as transações ainda estão sendo executadas e enquanto efetua suas atualizações. O mecanismo *off-line* é efetuado quando o servidor acha-se desligado. O controle de concorrência é garantido pela utilização dum esquema de bloqueio de duas fases que, nas operações de escrita duma transação, num primeiro instante efetiva o bloqueio local, ou seja, em nível de cliente. As alterações locais geram os arquivos de *log*, do tipo refazer (*log-undo*), que são enviados a uma *cache* de páginas de *log*. Posteriormente, se o bloqueio local foi garantido, esses arquivos de *log* são repassados para o servidor que cuidará de efetivá-los. Esse repasse ocorrerá na execução duma operação de *commit*. Esse processo proporciona um alívio na transferência de dados entre cliente/servidor, pois são repassados apenas os bytes alterados.

b) Gerenciador de Objetos é a camada intermediária do O_2Engine. Atua sobre o O_2Store e sob o gerenciador de esquemas. Essa camada é responsável pelas seguintes tarefas: gerenciamento dos objetos e seus identificadores, dos atributos estruturados e suas operações, implementação de estratégias de persistência, tanto o modelo de "alcançabilidade" do Smalltalk, quanto o modelo de persistência do ODMG em C++, implementa a coleta de lixo para os objetos liberados e implementa as estratégias de indexação e *clustering* baseadas em objetos complexos e heranças. É nessa camada que são gerenciadas as várias versões dos objetos, considerando-se objetos complexos e referências entre objetos *versionables* e os *non-versionables*. O tratamento dado pelo gerenciador de objetos na transformação dos objetos e valores em registros obedecem aos seguintes critérios:

- valores não atômicos ou valores construídos dependem da especificação dos limites para a definição dos registros necessários para o seu armazenamento. Um exemplo são as *tuplas* que, para armazenar seus atributos dependem de como esses foram definidos. Supondo que uma *tupla* seja composta por um atributo cujos valores são uma lista e um outro atributo inteiro. Nesse caso, deverão ser requisitados pelo menos dois registros, um para cada atributo da *tupla*. Um outro exemplo são as seqüências de caracteres (*strings*) que podem exigir um ou mais registros segundo o seu tamanho;

- coleção de valores (listas, conjuntos e *bags*), se tem um tamanho pequeno e cabem numa página, são armazenados através da disposição seqüencial dos identificadores num único registro. Se o tamanho excede o registro, então são dispostos, para cada caso, do seguinte modo:

  lista- seus elementos estão dispostos numa estrutura de árvore-B do tipo posicional;

  conjuntos - seus elementos estão armazenados numa estrutura de árvore-B onde as chaves nos nós são identificadores de registros que armazenam os elementos que compõem o conjunto ou, no caso de serem atômicos, as chaves são os próprios elementos atômicos;

  bags - são representados em arquivos seqüenciais com os identificadores dos registros que compõem seus elementos ou, no caso de serem atômicos, uma sequência composta pelos próprios elementos.

c) O Gerenciador de Esquemas do O_2Engine trabalha como uma aplicação sobre o gerenciador de objetos. Essa camada é responsável pela criação, eliminação, atualização e recuperação das características estruturais e comportamentais das informações manipuladas pela base de dados, como as classes, tipos, métodos, funções e nomes. As atividades do gerenciador de esquemas podem ser efetuadas em tempo de compilação ou de execução. O último caso é uma propriedade importante que permite a definição de esquemas dinâmicos.

O processo que possibilita aos esquemas e dados evoluir, pode ser concretizado pelo projetista após a confirmação das atualizações ou pode ser adiado, de forma a reduzir, naquele momento, o *overhead* exigido pela operação. No caso da alteração no esquema ser imediata, o gerenciador de esquemas encarrega-se da criação de novas versões das classes modificadas e da adaptação do esquema a essas atualizações. Nesse processo, como os identificadores de objetos devem ser preservados, a evolução dos esquemas é limitada para os casos em que se fere a integridade referencial.

O gerenciador de esquemas permite a definição de classes e métodos temporariamente incompletos. Por exemplo, pode-se definir como atributo duma classe outra classe ainda não definida. Permite-se também que a interface dum método utilize classes ainda não especificadas.

Esse módulo contém um conjunto de APIs que possibilitam obter informações a respeito do esquema da aplicação, como, por exemplo, qual a classe e sua estrutura para um dado objeto, os tipos dos atributos etc. Através das APIs é possível navegar na estrutura duma hierarquia de classes.

3.3.3 *Backup* e recuperação

O O_2 proporciona duas possibilidades de execução de *backup*: *on-line* e *off-line*. O *backup on-line* é efetuado de modo consistente enquanto o sistema acha-se em execução e ao mesmo tempo em que as transações possam estar efetuando suas atualizações. O *backup off-line* é efetuado com o servidor fora de execução. O processo de recuperação é disparado automaticamente sempre que o sistema servidor é reiniciado.

3.3.4 Mecanismo de replicação

O sistema de replicação de dados é baseado no arquivo de *log* para as atualizações de páginas, cujos registros são obtidos durante a execução de transações. A replicação de dados é um processo que tem como objetivo transferir as atualizações num arquivo de *log* dum sistema ativo para um sistema passivo. A transferência é efetuada através do sincronismo entre dois processos, o *send,* no sistema ativo, e o *receiver*, no sistema passivo. Os dois processos são disparados e executados independentemente para suportar a replicação de dados, em seguida o *send* envia o identificador da última transação para o *receiver*, do lado passivo, que se sincroniza com o identificador de transação. Esse processo pode ocorrer em tempo real ou de modo assíncrono. Em tempo real, toda vez que são gerados registros de *log* duma nova transação, ocorrerá o sincronismo baseado nos identificadores de transações e a efetivação das atualizações através da transmissão entre *send* e *receiver.*

3.3.5 Coleta de lixo

Após o uso intensivo da base de dados poderá ocorrer a liberação de espaços utilizados pelas aplicações, mas que se tornaram ociosos. O O_2 possui um sistema coletor desses espaços que os identifica e os elimina, através da compactação da base de dados. Esse recurso de reorganização utiliza-se também de possíveis alterações ou criação de novas bases de dados em volumes, como descrito na seção 3.3.2.

3.3.6 Indexação

O O_2 utiliza estruturas de indexação como suporte às consultas. Assim como as árvores de *clusters*, descritas na seção a seguir, as estruturas de indexação são informações de esquemas físicos atrelados à base de dados. Essas estruturas obedecem ao princípio de independência lógica e física, ou seja, a criação ou eliminação de índices deve afetar somente o desempenho da aplicação. As estruturas de indexação são construídas sobre coleções "nomeadas" de objetos ou valores, uma opção melhor do que as que atuam nas extensões de classes, fato que minimiza o *overhead* provocado pela manutenção dos índices. A estrutura de indexação suporta herança entre classes e a solução nesse caso é a mesma dada a uma coleção de objetos, ou seja, há entradas para todas as instâncias, inclusive para as das subclasses. Um índice pode ser um atributo simples dum objeto ou um atributo de atributo, de modo a formar um caminho de composição, como, por exemplo, o mostrado a seguir:

documento.capítulo.parágrafo

O comando abaixo cria um índice para a classe documento que poderá ser utilizado, por exemplo, para recuperar todas as instâncias de documentos que possuam determinado parágrafo:

add index Doc_paragrafo *on class* documento *path* capítulo.parágrafo

A implementação adotada pelo O_2 é a proposta por Maier e Stein em 1987 denominado Múlti-índice[Ber_94], na qual os componentes básicos de índices são obtidos por decomposição aplicada ao caminho da estrutura hierárquica, obtendo-se um índice para cada *link* do caminho. A necessidade da decomposição em índices básicos deve-se à facilidade em assimilar as alterações que possam ocorrer nos seus valores.

Na indexação de hierarquias, o O_2 utiliza-se do esquema de indexação de hierarquia de classes proposto por Kim[Ber_94], cuja implementação apresenta alteração nos nós folhas, nos quais é possível recuperar-se todos os objetos duma subclasse num único bloco, sem que se tenha de ler a folha toda.

Toda consulta efetuada através da OQL faz com que o otimizador de consultas identifique as estruturas de indexação adequadas, utilizando-as para a composição do resultado a ser retornado.

3.3.7 Gerenciador de *clustering*

O Gerenciador de *Clustering* permite ao Administrador de Banco de Dados (DBA - DataBase Administrator) controlar e definir a forma de alocação dos objetos. Isso é efetuado através duma expressão que possibilita a descrição dos componentes dum objeto complexo, os quais devem estar fisicamente próximos. A expressão descrita pelo DBA é um subgrafo da composição hierárquica das classes alvos da alocação esperada. Um exemplo dessa expressão é mostrada a seguir:

create cluster Pessoa *on* (nome, mora_em, data_nascimento)

No qual a composição hierárquica para a classe *Pessoa* é a apresentada através do exemplo na figura3.11.

A figura3.11 destaca o subgrafo da composição hierárquica, cuja raiz é a classe *Pessoa*. Nesse caso, a expressão orienta o sistema para que os componentes nome, *mora_em* e *data_aniversário* de cada instância da classe *Pessoa* sejam fisicamente alocados o mais próximo possível. Do contrário, ou seja, se nenhuma expressão for definida, o sistema irá gerar uma árvore de alocação *default* para toda hierarquia de composição dessa classe. A árvore de alocação pode ser modificada ao longo do tempo e ser separada de qualquer informação do esquema da base de dados.

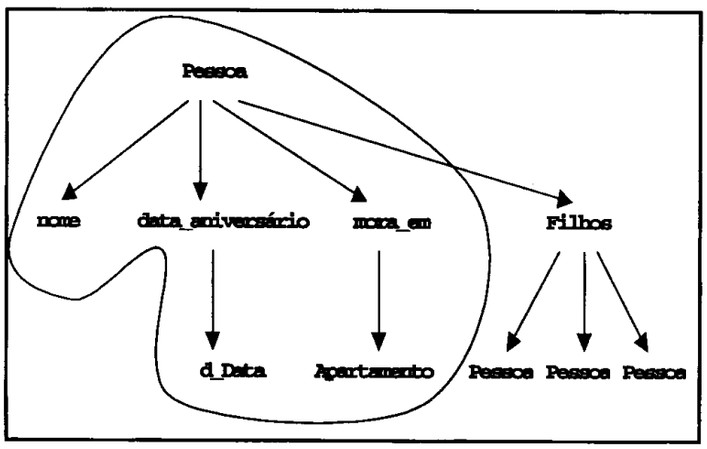


Figura 3.11: Um exemplo de *clustering*

O algoritmo de *clustering default* é executado durante uma operação de *commit* e procura alocar o mais próximo possível todos os componentes dum objeto complexo criado pela transação. A árvore de composição da hierarquia da classe define os componentes da classe que devem ficar fisicamente próximos. O objetivo desse algoritmo baseado na árvore de alocação é possibilitar boas alocações na criação dos objetos, mas ele não leva em consideração sua evolução dinâmica.

3.4 IRIS

O Iris é um Sistema de banco de dados desenvolvido pela Hewlett-Packard Laboratories, de Palo Alto, CA[Fis_87][Wil_90][Zan_95]. A implementação do Iris partiu da idéia em reunir as principais características da nova geração de SGBDs para atender as necessidades das aplicações emergentes: sistemas de informação de escritórios, sistemas baseados em conhecimento, sistemas de engenharia de testes e medidas e sistemas de desenvolvimento de projetos de software e hardware.

Além de apresentar as características dos SGBDs convencionais tais como: dados persistentes, controle de concorrência, *backup* e recuperação, o Iris foi projetado para oferecer novos recursos, como, por exemplo: um conjunto de construtores para a modelagem de dados, suporte à inferência direta na base de dados, suporte a novos tipos

de dados (imagens, voz, texto, vetores, matrizes), suporte às transações longas e suporte à múltiplas versões dos dados.

A arquitetura do Iris pode ser representada pelos seguintes módulos:

3.4.1 Interfaces de consultas e ambientes de programação

A interface com os usuários do Iris é efetuada através dos seguintes módulos:

- Interfaces de consultas interativas. As consultas interativas no Iris podem ser executadas através da linguagem **Object SQL** - uma extensão do SQL ou através do Inspetor - um utilitário do ambiente Hewlett-Packard Artificial Intelligence Workstation, que possibilita aos usuários da linguagem LISP examinar entidades numa base de dados Iris. O Inspector oferece um *browser* textual, onde é possível especificar os níveis de detalhes que se deseja apresentar as informações armazenadas na base de dados;

- Suporte à construção de aplicações em Lisp. Um ambiente de programação Lisp suportado por um conjunto de subrotinas em Lisp;

- Suporte à construção de aplicações em C. Uma interface para a construção de programas de aplicação na linguagem C, com um conjunto de recursos implementados oferecidos através de subrotinas em C.

3.4.2 Gerenciador de objetos IRIS

Esse módulo suporta o modelo de dados Iris através da definição de esquemas, manipulação de dados e processamento de consulta. Um esquema baseado no modelo de dados Iris é definido através dos seguintes elementos: construtores como objetos, tipos e operações; de representações de abstrações estruturais como classificação, generalização/especialização e agregação; de heranças, propriedades genéricas, restrições, dados complexos, operações definidas pelos usuários, controle de versões, inferência e tipos de dados extensíveis.

3.4.3 Gerenciador de memória IRIS

Esta camada do protótipo Iris tem uma concepção similar ao do gerenciador de memória relacional do SGBD Hewlett-Packard's Allbase[Wil_90]. É nessa camada que se efetua o *clustering* dos objetos de mesmo tipo, isto é, tornam-se disponíveis em uma mesma relação. Os relacionamentos entre tipos de objetos diferentes são especificados através de atributos, cuja forma de armazenamento é omitida pela linguagem de programação. Portanto, as relações são construídas para armazenar as linhas componentes dum relacionamento.

3.4.3.1 Gerenciamento de transação

O gerenciador de memória Iris foi projetado para suportar transações longas e aninhadas e permite a definição de transações com *savepoint* e *restore to savepoints*, isto é, podem-se definir pontos numa transação que serão utilizados como limitantes no processo de recuperação.

3.4.3.2 Controle de versões e controle de concorrência

Nesse módulo é implementado o suporte do controle de versões, cujo objetivo é atender às aplicações que exigem várias descrições dos aspectos dum mesmo objeto, seja por motivos experimentais, corretivos, adaptativos, seja também por refinamento ou atualizações, definindo um marco dum trabalho realizado ou a realizar-se. O controle de versões permite que várias versões de um objeto possam ser especificadas pelos usuários numa árvore de versões. Todo objeto pode-se tornar *versionable* e, após adquirir esse estado, só poderá ser acessado através da aplicação explícita dos comandos de *check-out* ou *check-in*, capazes de criar novas versões do objeto. O módulo gerenciador de memória trabalha com dois tipos lógicos de base de dados, a pública e a privada. O controle de bloqueio é efetuado através dum modelo hierárquico de bloqueio por intenção, bem como o uso dos bloqueios convencionais exclusivo e compartilhado. O mecanismo de controle de versões é a base da implementação do controle de concorrência. Por exemplo, quando a versão dum objeto está no modo *check-in*, essa versão do objeto e toda a sua sub-árvore de versões ficam bloqueadas na base de dados pública. A efetivação do bloqueio a esse objeto na base de dados evita a ocorrência de acessos por outras transações, embora seja possível efetuar acessos para as outras versões desse mesmo objeto. Uma vez que se tenha aplicado ao objeto um comando de *check-out*, as referências às versões desse objeto poderão ser feitas na base de dados privada. As revisões das versões privadas ficam efetivadas na base de dados pública a partir do instante que toda sub-árvore sob o comando *check-in* for liberada.

3.5 GemStone

A primeira versão do GemStone[But_91] executava numa máquina DEC Vax, sobre o sistema operacional VMS. Nessa plataforma era possível obter-se até seis réplicas da base de dados através da rede. A versão seguinte passou a suportar redes heterogêneas de estações de trabalho, nas quais os servidores Gem passaram a ser portáveis para as seguintes máquinas: Sun 3, Sun 4, Sony NEWS, DECstations e IBM RS6000. A versão GemStone/S 5.1.4 opera também sobre plataformas HP9000 e sistemas Intel, usando respectivamente os sistemas operacionais HP-UX e Windows NT[Gem_00].

3.5.1 Arquitetura

A arquitetura do GemStone está baseada em dois processos: o servidor Gem e o monitor Stone. O processo Gem é responsável pela execução no servidor do comportamento dos objetos e da avaliação de consultas. O processo Stone executa a alocação de objetos, através da manipulação de blocos de memória e coordena as operações de *commit* nos vários servidores Gem.

3.5.2 Recuperação e *backup*

O processo de recuperação é suportado por atividades como a desempenhada pelo sistema de coleta de lixo. Esse sistema foi inicialmente tornado disponível somente para execução em *off-line* e, nas versões posteriores, também em *on-line*. Através da utilização de algoritmos de coleta de lixo, como o de varredura-marcação (*mark-sweep*), é possível identificar preventivamente os objetos candidatos a serem excluídos. Os objetos temporários, isto é, objetos criados durante uma transação, mas que ainda não foi encerrada, são marcados e, se for o caso, eliminados, caso a transação não seja efetivada. O GemStone utiliza-se de dois sistemas de *caches*: um de páginas e outro de objetos. O *cache* de objetos contém apenas objetos criados pela transação e é nesses objetos que aplica-se o algoritmo de prevenção para a fase de coleta. Assim, no caso de ocorrência de falhas, esses objetos marcados serão alvos do algoritmo de coleta de lixo. Esse processo ocorre, no caso *on-line*, sem provocar conflitos nas transações.

O GemStone permite a execução de *backup* total, ou seja, o alvo dessa operação é toda a base de dados, e incremental, nesse caso, apenas as informações a respeito das alterações que serão armazenadas. O *backup* incremental é efetuado no modo *on-line*, ou seja, não há a necessidade do sistema estar desligado para que se execute essa tarefa.

3.5.3 Gerenciamento de transações e controle de concorrência

O GemStone implementa um gerenciamento de transações que obedece as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Embora possibilite ao administrador do banco de dados selecionar o relaxamento no modo de serialização, permitindo que ocorram leituras "sujas", isto é, uma transação poderá ler informações que estejam sendo alteradas por outra transação, sem que esta última ainda tenha executado uma operação de *commit*.

O controle de concorrência proporcionado pelo GemStone tem por objetivo minimizar o impacto do acesso concorrente entre os objetos melhorando seu desempenho. Isso é efetuado com o uso do protocolo de concorrência otimista sobre os objetos mais prováveis de ser utilizados por outras transações e utilizando-se a forma de bloqueio pessimista para os objetos temporários. Essa combinação de bloqueios tem por propósito suportar grande quantidade de usuários que compartilham o mesmo objeto. No controle de concorrência otimista as transações operam considerando-se que não ocorrerão conflitos. Durante a operação de efetivação verifica-se a garantia do isolamento das transações envolvidas. Se não ocorreu qualquer conflito, então a transação efetiva as alterações na base de dados. Caso contrário, a operação *commit* falha e o servidor Gem tem por opção cancelar a transação, iniciando-a novamente.

O suporte ao gerenciamento de transações e controle de concorrência é efetuado pelo esquema de duplicação (*shadow*) de páginas, no qual as modificações efetuadas por uma transação são implementadas numa cópia da página e não na página original. O armazenamento em disco das alterações só será efetuado quando a transação requisitar uma operação *commit*. Nesse instante a cópia da página passa a ser a original e armazenada em disco, enquanto que a antiga original só será liberada para novos usos quando todas as transações que a requisitaram para a leitura efetivem uma operação de *abort* ou *commit*. No caso de ser efetuada uma operação de *abort*, todas as páginas alteradas pela transação serão liberadas.

3.5.4 Objetos e identificador de objetos

O GemStone utiliza-se de uma tabela de objetos armazenada numa estrutura de árvore que possibilita fazer-se o mapeamento entre o identificador do objeto e o endereço físico do objeto. Essa tabela é sempre atualizada, numa operação *commit* para todos os objetos modificados, de modo que sejam acessados os novos objetos em vez dos velhos. Uma cópia da raiz da tabela objeto (*object table root*) é mantida na base de dados para o suporte à propriedade de atomicidade das transações. Assim, se houver ocorrência de falhas, identificadas pela presença de cópias diferentes, deve-se recuperar a cópia mais antiga da raiz da tabela.

3.5.5 Outras características do GemStone:

- Permite o *clustering* de objetos logicamente relacionados e freqüentemente acessados em espaços próximos, com o objetivo de facilitar as operações de recuperação dos objetos e seus valores. Os objetos grandes, ou seja, aqueles que são maiores do que as páginas, são decompostos em pequenas porções para que, numa operação de recuperação seja possível requisitar apenas as partes utilizadas e não o objeto todo;

- Para auxiliar as consultas é utilizada uma estrutura de índices de caminho apresentada no capítulo 2, a qual possibilita-lhe que seja efetuada uma avaliação prévia no servidor antes de determinar o resultado;

- Oferece suporte às políticas de versionamento linear e paralelo;

- Permite a modificação dinâmica de esquemas, nas quais é possível efetuar alterações nas definições das classes com a conversão automática das instâncias das classes afetadas por essas modificações;

- Oferece o suporte à comunicação (*gateway*) que possibilita a utilização de dados numa base de dados Sybase;

- Oferece o suporte à utilização de múltiplas linguagens: C, C++, Smalltalk;

- Possui uma coleção de ferramentas gráficas para atividades como: definição de esquemas, programação e navegação na base de dados.

3.6 ObjectStore

O ObjectStore[Lam_90][Obj_00] é um sistema de gerenciamento de base de dados orientado a objetos que oferece uma interface para a criação de aplicações em C++; permite o acesso aos dados no mesmo formato manipulados pelas aplicações; permite a descrição, o armazenamento e a consulta de objetos complexos, bem como os dados tradicionalmente manipulados pelos SGBDs relacionais; e permite o armazenamento persistente dos dados independentemente do tipo de dado. O ObjectStore[Obj_00] está disponível em paltaformas que se utilizam dos sistemas operacionais Windows, UNIX, Sun Solaris SPARC & Intel, Hewlett-Packard HP-UX, Silicon Graphics Irix, IBM AIX e Compaq Tru64 UNIX.

3.6.1 Arquitetura

O ObjectStore é baseado numa arquitetura cliente/servidor, cujo princípio é aliviar o servidor das atividades que possam ser efetuadas localmente. Algumas atividades freqüentemente executadas pelos servidores nos SGBDs relacionais, no ObjectStore são efetuadas pelo cliente, como, por exemplo, parte do processamento de

consultas e algumas tarefas do SGBD. Ao servidor cabem tarefas como a manipulação (armazenamento, recuperação, transmissão, recepção etc.) dos objetos e a conversão de valores persistentes para transientes e vice-versa. Isso sempre é feito utilizando páginas de memória cujo conteúdo não é interpretado pelo servidor. O servidor executa o controle de concorrência para as múltiplas aplicações dos clientes e efetua a tarefa de recuperação da base de dados, através de técnicas oriundas dos SGBDs convencionais.

A figura3.12 mostra uma arquitetura cliente/servidor, na qual existem vários terminais (*host*) conectados por enlaces de comunicação. Dentre esses terminais, alguns contêm aplicações cliente; um outro é um servidor de arquivos que permite o compartilhamento de recursos como discos, impressoras etc.; e um outro é o servidor de banco de dados que faz o gerenciamento dos dados e os acessos aos discos da base de dados. O ObjectStore possui um gerenciador de *cache* local em cada *host* que contém uma ou mais aplicações clientes em execução, e outro gerenciador de *cache* no servidor de banco de dados.

No ObjectStore, a comunicação entre as aplicações cliente e o servidor de base de dados ocorre da seguinte maneira: se um objeto é desreferenciado na memória virtual da aplicação, esse poderá ou não estar presente no conjunto das páginas mais recentemente usadas e controladas pelo *cache* local. Caso não se encontre disponível, essa ação provocará violação de acesso detectada por hardware, a qual fará com que o sistema operacional acione o ObjectStore, comunicando-o que houve falha de acesso à memória. Nesse momento, o ObjectStore obtém a página desejada do servidor e a aloca no *cache* do *host* cliente. As leituras subseqüentes desse objeto não provocarão falhas, que ocorrerão somente quando estiver novamente apenas no estado persistente, ou seja, ausente do *cache*. A transferência dum dado do servidor para o cliente depende da liberação adequada para o tipo de acesso, sendo que essa tarefa é gerenciada pelo sistema de controle de concorrência.

3.6.2 Endereçamento

O ObjectStore possui uma área denominada "local de endereços" para cada processo cliente. Essa área é composta duma porção de memória virtual e pode receber todas as espécies de dados utilizados pela aplicação, ou seja, dados persistentes e transientes. O local dos endereços, particular para cada cliente, reserva uma porção da memória para os dados persistentes e outra para os demais dados. Esses dados são oriundos da memória física, do *cache* do cliente (para os dados persistentes) e do local de *swap* (para os dados transientes).

3.6.3 Controle de concorrência

O processo servidor armazena e recupera páginas em resposta às requisições dos processos cliente. Na realidade, essas requisições serão ou não atendidas, de acordo com o tipo de bloqueio implementado para as páginas envolvidas. No ObjectStore, a tarefa de controlar a concorrência cabe ao servidor que a executa utilizando-se do esquema de

bloqueio de duas fases de páginas.

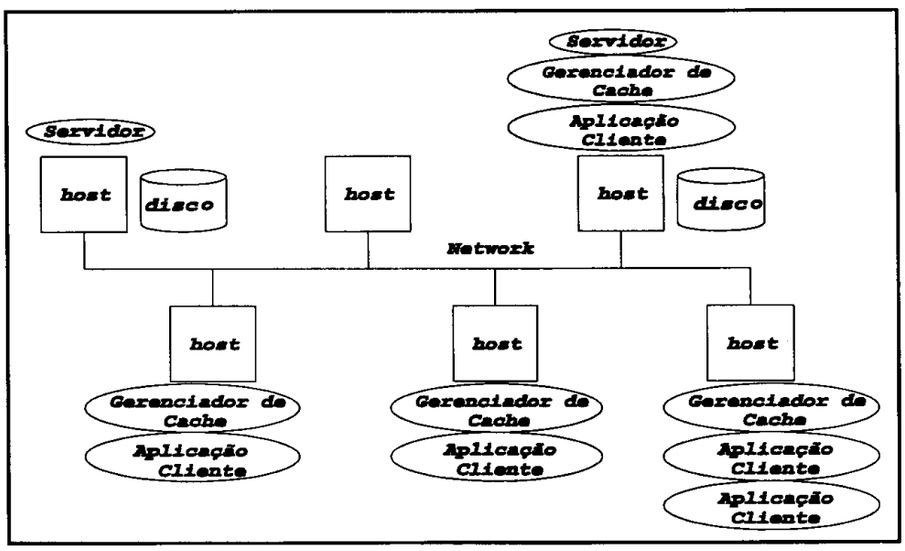


Figura 3.12: Arquitetura cliente/servidor do ObjectStore

Situação que pode ocorrer é uma página que reside num *host* cliente, sem bloqueio, sofrer uma modificação em outro *host* cliente que também a possui, fato que invalida a cópia do primeiro cliente. Esse processo é controlado por um mecanismo denominado "coerência de *cache*", cuja finalidade é manter a consistência das cópias das páginas nos respectivos clientes, garantindo a coerência quando uma transação altera uma das cópias. Para que esse mecanismo funcione, devem ser mantidos pelo servidor os caminhos para as cópias das páginas e os respectivos *caches* clientes que as utilizam, e em qual modo de bloqueio (compartilhado ou exclusivo) cada uma está. Assim que um cliente requisita uma página para o servidor, este notifica o cliente que a tem sob seu domínio, questionando a existência de conflito nos modos de bloqueios. Nesse caso, ocorrerá uma requisição no sentido contrário ao usual, ou seja, o servidor é que notificará o cliente. Essa mensagem é denominada de *callback* e é baseada no sistema de arquivos Andrew. Se a página requisitada estiver bloqueada, o cliente que detém o seu domínio responderá negativamente. Caso contrário, esse cliente torna-a disponível removendo-a do seu *cache*. Quando a resposta for negativa, o servidor envia uma mensagem para que o cliente requisitante espere até que a página solicitada seja liberada. Com essa estratégia, o servidor impedirá que uma mesma página seja atualizada ao mesmo tempo por dois ou mais clientes.

3.6.4 Recuperação

A recuperação de falhas está baseada na criação de registros de *log* que utiliza o protocolo *Write Ahead Log* - WAL. Nesse protocolo, todo servidor mantém um *log* das transações que permitem recuperar a base de dados para um estado consistente em caso da ocorrência de falhas. O *log* é composto de dois segmentos denominados *log* registro e *log* dado. Esses dois segmentos são utilizados pelo servidor durante a recuperação. Através duma operação denominada "propagação", o servidor transfere para o modo persistente os dados modificados numa operação *commit* descritos no *log* dado - isto é efetuado de modo transparente ao processo cliente.

3.6.5 Atributos do tipo coleção

O ObjectStore permite a manipulação de dados estruturados, tais como listas, conjuntos e *bag*, através da definição da classe *os_collections*. Um conjunto de métodos é oferecido para essa classe, os quais permitem a manipulação desses tipos de dados como valores de atributos dos objetos.

3.7 Outros

Outros SGBDOO podem ser encontrados na literatura ou diponíveis no mercado, alguns deles são: ONTOS da Ontos Inc., Objectivity/DB da Objectivity Inc.,VERSANT da Versant Object Technology Inc., POET da Poet Software Inc, GNOMO[For_93], Jasmine da Computer Associates [Ish_96] e GOA++, um projeto da COPPE/UFRJ [Mau_98].

3.8 Considerações finais

Nesse capítulo procurou-se mostrar o amplo esforço que vem sendo feito no sentido de suportar a nova geração de aplicações através da exposição de tecnologias utilizadas para a implementação das várias funcionalidades suportadas pelos SGBDOOs. Esses SGBDOOs permitem aos usuários criar classes com atributos e métodos, possibilitam que as subclasses herdem atributos e métodos de suas superclasses, criem instâncias de cada classe, cada qual com um único identificador, recuperem as instâncias individualmente ou em coleções e executem seus métodos.

Na apresentação dos SGBDOOs, mostraram-se algumas das arquiteturas e diferentes técnicas de suporte ao gerenciamento de *buffers*, gerenciamento de transação, controle de concorrência, recuperação, *backup*, identificadores de objetos, indexação, *clustering*, objetos complexos, atributos estruturados, *swizzling*, controle de versões etc. Algumas dessas técnicas são também encontradas em SGBDs relacionais e, portanto, não são exclusivas dos SGBDOOs. Outras técnicas são específicas ao suporte à orientação a objetos, como, por exemplo, estruturas de OIds, técnicas de *swizzling*,

indexação de hierarquia de classes etc. No entanto, a implementação de algumas características da orientação a objeto é, em alguns casos, limitada pelo grau de *overhead* que provocam no sistema. Um exemplo é apresentado pelo ORION, onde um objeto componente não pode ser compartilhado por outro objeto composto, ou no caso do ORION, O_2 e GemStone, no qual se aplica a propagação retardada das atualizações provocadas pela evolução de esquemas.

Os esforços despendidos nas implementações dos SGBDOOs contribuíram para a definição do padrão ODMG2.0. Assim como ocorreu com os SGBDs Relacionais, acredita-se que, com a padronização, seja possível obter um salto significativo na disseminação da construção de SGBDOOs e conseqüentemente permitir o desenvolvimento de novas tecnologias para o suporte às funcionalidades desejadas.

Capítulo 4

O NÚCLEO GERENCIADOR DE OBJETOS

4.1 Considerações iniciais

No presente capítulo, apresenta-se o Núcleo Gerenciador de Objetos - NuGO construído com o objetivo de suportar modelos de dados orientados a objetos, com ênfase no apoio as operações necessárias para a implementação das três principais abstrações de dados: classificação, generalização e associação. O desenvolvimento desse gerenciador usou como referência o modelo de dados SIRIUS[Bia_96] que suporta essas três abstrações. No entanto, o projeto levou em consideração que esse núcleo deve ser genérico, e foi previsto o suporte a qualquer modelo de dados, orientado ou não a objetos. Para isso, foi usado como referência também o suporte ao modelo relacional. Esse modelo solicita do núcleo apenas o suporte à abstração de associação, dispensando o suporte à classificação e à generalização. No entanto, a maior parte das técnicas de implementação de gerenciadores de dados foram desenvolvidas tendo em vista esse modelo. Assim, procurou-se desenvolver o NuGO de maneira a aproveitar o máximo possível técnicas e algoritmos desenvolvidos para o modelo relacional e tentando, dentro do possível, que o suporte às demais abstrações não levasse à degradação do desempenho do sistema global, quando o suporte às abstrações de classificação ou de generalização não se mostrassem necessárias. Com isso, pretendeu-se que o desenvolvimento do NuGO pudesse aproveitar a tecnologia já existente para a construção de gerenciadores de base de dados, ao mesmo tempo em que permitiu explorar novas fronteiras de importantes recursos para a implementação de gerenciadores que suportem as construções do paradigma de orientação a objetos.

Como resultado, o NuGO é genérico e em grande parte independe do modelo de dados para o qual se pretenda que ele suporte. A exposição da implementação efetuada é feita aqui através do relato dos principais conceitos, da arquitetura e das principais estruturas de dados que envolveram o desenvolvimento deste núcleo gerenciador de objetos. As principais funções suportadas por esta implementação são: os elementos básicos que compõem genericamente qualquer modelo de dados; criação de objetos, tuplas, atributos monovalorados, multivalorados e estruturados; alocação e gerenciamento de memória (*clustering*) baseados em composição de objetos; identificação de blocos de memória e objetos baseados na definição de tipos elementares; gerenciamento de *buffers;* gerenciamento de transação; bloqueio e controle de concorrência; e recuperação.

4.2 O Modelo de dados SIRIUS

Nesse subcapítulo, apresenta-se o modelo de dados SIRIUS utilizado como referência para o desenvolvimento do Núcleo Gerenciador de Objetos - NuGO. Essa descrição restringe-se aos aspectos que determinam a implementação do núcleo ou que são por ele afetados. Uma descrição completa do modelo é encontrada em[Bia_96] e [Ara_98].

O SIRIUS é um modelo de dados orientado a objetos e elaborado a partir de um formalismo que busca representar os elementos de um esquema de dados através da descrição de seus construtores semânticos na qualidade de particularizações de abstrações[Kim_87]. Utilizando esse formalismo, o SIRIUS foi elaborado com o objetivo de adequar-se à modelagem de sistemas de apoio à engenharia, apoio a projetos e a sistemas de apoio à decisão. Respaldando sua capacidade de modelagem de apoio a projetos (envolvendo diversas modalidades de Engenharia, tal como elétrica, mecânica e de software), bem como sistemas de apoio à Medicina, pretendeu-se incluir também o suporte às atividades de desenvolvimento, tal como a implementação de sistemas de software com desenvolvimento integrado às ferramentas de projeto, extensão de um projeto mecânico para entrar em produção, simulação etc.

O NuGO, parte fundamental do Gerenciador SIRIUS, atua como um servidor de objetos para as camadas semânticas que implementam o modelo de dados. Para efeito de praticidade, em algumas seções desse capítulo, será também referenciado como pertencente ao Gerenciador SIRIUS, aspectos funcionais implementados pelo NuGO, pois ambos são os mesmos diferindo apenas pelo suporte semântico adicional oferecido pelo Gerenciador SIRIUS. A menos da característica particular de propiciar um forte suporte para abstrações de objetos, o Núcleo é genérico e independente do modelo de dados adotado, podendo ser utilizado para suportar qualquer modelo de dados orientado a objetos que se pretenda implementar. As operações de *buffering* de dados no disco e de controle de transações, envolvidas neste trabalho aqui apresentado, são totalmente desempenhadas pelo Núcleo.

Como o Núcleo apresentado presta-se igualmente bem para suportar outros modelos, apesar de desenvolvido com vistas especificamente ao modelo SIRIUS, um entendimento detalhado do modelo não é necessário. Assim, pode-se descrever rapidamente o SIRIUS como um modelo que considera o mundo real como algo que pode ser modelado através de um conjunto enumerável de objetos. Todo objeto possui:

- um tipo ou classe;
- um código atribuído univocamente pelo sistema;
- um conjunto (possivelmente vazio) de identificadores do objeto que são definidos pelo usuário e
- um conjunto de atributos, cujos valores são de interesse para a aplicação.

Resumidamente, os principais conceitos que distinguem SIRIUS de outros modelos de dados existentes são principalmente:

- Disponibilidade de construtor semântico para representar as abstrações de classificação, generalização e composição em modelagens;
- Suporte unificado das abstrações normalmente tratadas em separado para
  generalização mais ampla (mais genérica);
- Distinção entre as propriedades e restrições das abstrações de generalização e classificação (tais como herança e valores *default*), normalmente misturadas no tratamento dado à abstração de generalização;
- Distinção entre as propriedades e restrições das abstrações de associação e composição (tais como representação de estrutura, identificação e dependência existencial), normalmente usadas de maneira indistinta no tratamento dado à abstração de associação;
- Distinção entre relacionamentos de cardinalidade fixa e relacionamentos com número fixo de papéis;
- Representação de tipos de objetos como objeto persistente;
- Ampla gama de estruturas de dados para atributos multivalorados, com a inclusão do tratamento de atributos identificadores em estrutura;
- Suporte à representação de estrutura de atributos multivalorados como atributo independente dos valores atribuídos na estrutura.

Em SIRIUS, são indistintamente chamados de atributos de um objeto tanto os estáticos quantos os dinâmicos. Assim, em SIRIUS, o termo atributo engloba valores e estruturas normalmente tratados como atributos, e também os métodos dos objetos e os parâmetros de interfaceamento do objeto com o exterior do sistema.

Em vistas ao presente trabalho, é suficiente considerar-se que objetos são armazenados pelo gerenciador SIRIUS com o uso de sequências de blocos de memória de tamanhos variáveis e/ou fixos, onde são armazenados os diversos elementos que compõem um objeto, tais como atributos e identificadores, bem como sua estrutura interna.

No desenvolvimento desse trabalho, procurou-se analisar características do modelo que pudessem beneficiar-se de suporte especial no núcleo, com vistas sempre ao custo/benefício de dispor de determinado suporte, e qual o impacto que a existência desse suporte no núcleo, em situações nas quais ele não era necessário, poderia causar no desempenho do sistema. O suporte às três abstrações básicas foi o principal elemento para definir os recursos a serem suportados.

Um conceito importante do modelo SIRIUS, que afeta a estrutura geral do núcleo, é o de objetos compostos[Tra_94][Kim_87], oriundo da abstração de composição: os objetos podem ser fisicamente compostos por outros. Isto é feito com o uso de construção semântica de alto nível, a qual permite agrupar objetos segundo diversos critérios definidos pelo usuário e controlar eficientemente a acesso e a recuperação de informações, baseada em critérios significativos para a aplicação. Fisicamente, objetos compostos são uma maneira de agrupar, num mesmo registro, objetos com alta probabilidade de serem acessados conjuntamente. Todo objeto forma exatamente um objeto composto, o que pode ser aproveitado pelo fato de que fisicamente cada objeto é armazenado em apenas um local. A base de dados é considerada um objeto composto, que é composta (direta ou indiretamente) por todos os objetos nela armazenados.

O suporte à abstração de classificação corresponde basicamente ao reconhecimento de que os tipos de objeto são também objetos. Isto produz dois requisitos para o núcleo: primeiro, surge do fato que, por serem também objetos, os tipos são altamente susceptíveis à atualização. Como o tipo define a estrutura de todos os objetos desse tipo, a variabilidade do tipo induz à possibilidade de restruturação freqüente em grandes conjuntos de objetos. O segundo requisito surge do fato que, para acessar um objeto, é necessário acessar as informações de seu tipo (como em qualquer gerenciador). No entanto, nesse escopo o tipo é um objeto que por sua vez também tem um tipo numa sequência que pode repetir-se diversas vezes. As informações de um objeto tipo influenciam as ações do núcleo. Assim, embora normalmente, o núcleo não interprete o conteúdo dos blocos de memória que ele gerencia, restringindo-se a supri-las para as camadas superiores, no caso de objetos tipo, a atualização deverá ocorrer com o uso de dois canais de informação: a usual, que trata as operações sobre um objeto tipo como um objeto igual aos demais; e as informações interpretadas sobre a atualização do objeto tipo, para permitir que os dados de um tipo se tornem disponíveis no núcleo sem que o objeto tipo seja necessariamente acessado sempre que um objeto qualquer desse tipo seja acessado. Isto pode ser feito com relativa facilidade, pois apesar de as informações contidas em um tipo serem abundantes e variadas, o subconjunto delas, necessárias à operação do núcleo, são apenas as que se referem ao tipo (identificador, atributo, tamanho fixo ou variável) e tamanho dos blocos de memória necessários para o seu armazenamento.

O suporte à abstração de generalização por parte do núcleo está concentrado na manipulação de atributos. A generalização atua na avaliação de valores e na execução de métodos de teste de propriedades, para determinar se um objeto atende aos critérios de determinada especialização e, uma vez constatada a ocorrência da especialização, determinar a existência de atributos, freqüentemente agrupados em tuplas, existentes em função da especialização.

4.3 A arquitetura do sistema SIRIUS e o suporte do núcleo para os aspectos semânticos

O Núcleo Gerenciador de Objetos - NuGO efetua todas as operações de alocação de espaços em e para os objetos, gerenciando o uso de espaço em disco e memória e cuidando das operações de transferência dos objetos de e para a memória. Provê o gerenciamento dos identificadores de objetos, gerenciamento de *buffers*, cuida dos aspectos de controle de transação, recuperação, aspectos de segurança lógica, e coopera com as camadas semânticas para os controles de acesso e concorrência, garantindo a consistência no acesso a registros.

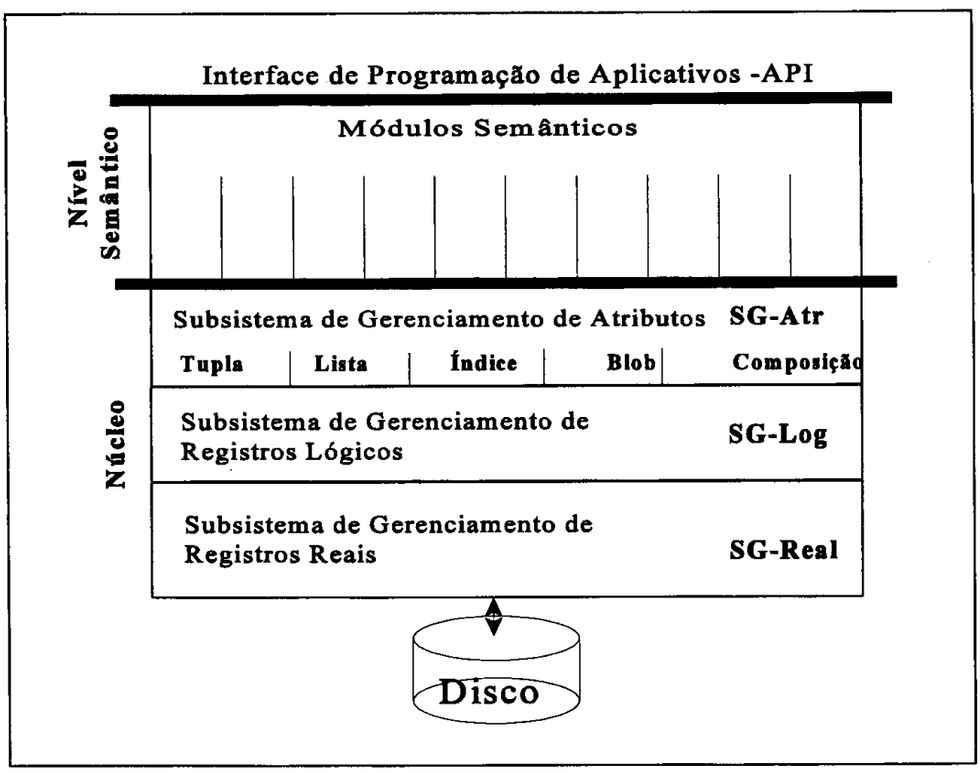


Figura 4.1: Estrutura em camadas do Gerenciador SIRIUS, com realce para o seu núcleo.

O Núcleo é montado numa estrutura de camadas, a qual aparece detalhada na figura 4.1. O nível semântico, que não é tratado no presente trabalho, utiliza o núcleo e implementa os elementos de um modelo de dados em particular. No caso do modelo

SIRIUS, os módulos implementariam por exemplo os conceitos de objeto, colônia, número etc[Bia_96]. No caso do modelo relacional, os módulos implementariam os conceitos de tuplas (segundo a acepção do termo no modelo relacional), as várias formas de chave e demais restrições de integridade, tabelas etc.

A camada superior do núcleo (**SG-Atr**) é a única que pode interfacear com as camadas implementadoras da semântica do modelo de dados do gerenciador. Provê suporte para manipular atributos concatenados em formato de tupla ou em estrutura de lista, com a possibilidade de proporcionar suporte para estruturas de índice sobre esses atributos, suporte para campos longos não estruturados e suporte para composição de objetos identificáveis um a um.

A camada seguinte (**SG-Log**) suporta a alocação e a recuperação de espaços contínuos, em registros físicos, para a camada SG-Atr. A estrutura em camadas é construída de maneira que as camadas inferiores reconheçam cada vez menos a semântica nos dados e o conteúdo dos blocos de memória manipulados. A camada SG-Log já não reconhece a estrutura interna dos blocos de memória manipulados, apenas cuida de alocar e manter coesos e recuperáveis os blocos de memória solicitados, bem como permitir a recuperação de espaços liberados.

A camada Subsistema de Gerenciamento de Registros Reais (**SG-Real**) efetua as operações de controle de acesso e alocação de espaço físico em disco para os elementos dos objetos; executa as operações de *buffering* dos registros físicos do disco em memória através de um algoritmo de emulação de memória *cache*, suprindo também as operações físicas de acesso ao disco; auxilia na execução do gerenciamento dos identificadores de objeto através da manutenção de dados de recuperação dos blocos de memória; efetua o suporte ao gerenciamento de transações; oferece o suporte ao controle de concorrência e recuperação de falhas provocadas por operações de *abort*.

O Gerenciador SIRIUS representado na figura 4.1 é na realidade o "motor" de acesso aos dados, sendo que posteriormente deverão ser incluídos utilitários como compiladores de linguagem de consulta, módulos de *backup* e recuperação de falhas por *hardware*, etc, de modo a completar-se o sistema SIRIUS com a formação da Interface de Programação de Aplicativos (API). A definição e implementação desses recursos está sendo executada em projetos paralelos a este e, portanto, não fazem parte do escopo do presente trabalho. A descrição de cada camada do núcleo é efetuada a seguir.

4.3.1 Subsistema de Gerenciamento de Atributos

Objetos são armazenados como uma concatenação de atributos que possuem um código de acesso coletivo. Esse código é o OId do objeto. Os atributos que o compõem podem ser concatenados em tupla ou lista. Recuperar atributos armazenados como tupla pré-definida (ocupando posições consecutivas de memória acessadas numa única operação) é mais eficiente que recuperar atributos armazenados como lista. Por outro lado, a armazenagem por meio de listas é mais flexível[Ber_94].

Na prática, é comum que os objetos do 'mundo real' tenham: uma parte "fixa", comum a todos os objetos do mesmo tipo, composta apenas de atributos monovalorados e uma parte "variável", onde ocorrem os atributos opcionais, os atributos não previstos no esquema e os atributos multivalorados. A camada SG-Atr foi construída com o objetivo de tornar disponíveis recursos para que os módulos de nível semântico possam armazenar objetos em "partes", de maneira a buscar o melhor compromisso entre as duas possibilidades. Esses recursos são implementados através de 5 módulos: SG-Atr-Tupla, SG-Atr-Lista, SG-Atr-Índice, SG-Atr-Blob e SG-Atr-Composição. Esse trabalho ateve-se à implementação dos módulos: SG-Atr-Tupla, SG-Atr-Lista e SG-Atr-Composição. Os demais módulos: SG-Atr-Índice e SG-Atr-Blob serão objetos de trabalhos futuros. Os principais serviços implementados por esses módulos aparecem descritos a seguir.

4.3.1.1 Suporte à armazenagem de tipos e de objetos

Em SIRIUS, a definição de objeto e de atributo é feita por intermédio do esquema de dados, através da definição de objeto, tipos de objeto, atributo, tipos de atributo etc. O sistema reconhece *a priori* alguns objetos especiais que permitem a definição dos demais. Um deles, o objeto denominado "Meta-Tipo", representa o tipo de objeto que não tem outros tipos já definidos pelo usuário, o que permite iniciar a hierarquia de tipos do esquema. Outro objeto reconhecido *a priori* pelo sistema é a colônia[1] de objetos denominada "Colônia Global" que representa a coleção de todos os objetos que compõem diretamente a base de dados (ou seja, aqueles que não são parte de outros já existentes na base). Criação e manipulação de um objeto e de um tipo de objeto são operações realizadas em cooperação entre o núcleo do gerenciador SIRIUS e as camadas de nível semântico.

Todo tipo de objeto definido em SIRIUS é por sua vez um objeto. Um objeto pode ser indicado como sendo ele próprio um tipo para outros objetos, habilitando-se a sua capacidade de ser instanciado. Essa habilitação é solicitada pelos módulos de nível semântico, e é efetivada pela camada SG-Atr. A operação de habilitação de instanciação parametriza como as instâncias devem ser geradas. Tal parametrização passa a fazer parte do registro do objeto como estrutura reconhecida pela camada SG-Atr. Assim, o SG-Atr pode utilizar a estrutura de instanciação de um tipo, para gerenciar a armazenagem dos objetos que têm esse tipo. O objeto "Meta-Tipo" já têm a parametrização para instanciação reconhecida *a priori* pelo sistema, permitindo que o usuário possa iniciar a definição de seus próprios tipos.

É interessante notar aqui que SIRIUS suporta as abstrações de instanciação/classificação (tipo e instância) e de especialização/generalização (subtipo e supertipo) como abstrações distintas. Usualmente as ocorrências de abstração de generalização criam hierarquias de diversos níveis, enquanto as ocorrências da abstração de classificações criam hierarquias com apenas dois níveis definidos pelo usuário (tipo e instância). SIRIUS suporta a criação de qualquer hierarquia com qualquer número de níveis.

---

[1]Colônia: são partições do conjunto de objetos da base de dados. Cada colônia é constituída de objetos de diversos tipos, denominados de *objetos habitantes* da colônia.

O subsistema de gerenciamento de atributos do Gerenciador SIRIUS assume que um objeto (que tem um OId gerado pelo próprio núcleo) é composto por uma lista de atributos e por uma lista de tuplas com estruturas pré-definidas no esquema. Como o gerenciamento feito nesse nível preocupa-se com as características de armazenagem dos objetos e não com suas características semânticas, denominam-se essas listas e tuplas de i-listas e i-tuplas (de implementação), para distinguí-las das listas e tuplas semânticas dos níveis de aplicação. Cada objeto tem suas próprias i-lista e i-tupla.

A i-lista armazena todos os atributos de um objeto, mesmo aqueles que fazem parte de alguma tupla. No entanto, nem todo atributo faz parte de alguma tupla, existindo assim apenas na lista de i-tuplas.

Quando um objeto é criado, solicita-se ao núcleo a criação de um objeto de determinado tipo. Nessa operação devem ser fornecidos os atributos monovalorados obrigatórios (que não significa necessariamente apenas os que não podem ser nulos). Se o tipo desse objeto não for subtipo de nenhum outro tipo definido pelo usuário, esses atributos irão compor a primeira i-tupla da lista de i-tuplas do objeto. Essa primeira i-tupla armazena portanto, todos os atributos básicos de um objeto, permitindo unificar num único acesso a recuperação de todos os atributos que uma base de dados equivalente, ao utilizar o modelo relacional, recuperaria. Se o tipo desse objeto for subtipo de um ou mais outros supertipos definidos pelo usuário (estruturados segundo um grafo acíclico de especialização), esses atributos serão decompostos em tantas i-tuplas quantos são os supertipos diretos e indiretos desse tipo (incluindo o próprio tipo natural). Em todos os casos, cria-se a i-lista de atributos do objeto, em que cada atributo em qualquer i-tupla é uma célula da lista. Atributos multivalorados, opcionais e extras poderão ser incluídos depois, e serão inseridos apenas na i-lista de atributos.

Essa concepção procura aliar o melhor dos dois mundos possíveis para a armazenagem de atributos (por lista ou por tupla)[Ber_94]. Embora isso cause sobrecarga nas operações de insersão de objetos, agiliza as operações de recuperação e modificação que são maioria durante a operação normal da base. Essa sobrecarga é minimizada ao mesmo tempo que aumenta a efetividade da operação das demais camadas do Gerenciador SIRIUS, as quais com isso podem flexibilizar as possibilidades de armazenagem, mantendo as partes coesas o suficiente para aumentar a taxa de acertos (*hit ratio*) do *cache*.

4.3.1.2 Suporte para atributos

Além de prover os mecanismos básicos para a definição de objetos e tipos, o SG-Atr provê suporte para os atributos. Esse suporte oferece aos módulos de nível semântico a possibilidade de manipular atributos individualmente, como tuplas ou listas. Além disso, permite indicar um ou mais atributos para a criação de estruturas de indexação, para agilizar buscas. Permite também tratar atributos longos (Blobs - *Binary large objects*), numa estrutura que autoriza a recuperação por partes seqüenciais. Isso permite otimizar as operações segundo a estrutura de dados mais adequada para representar os objetos e seus componentes, caso a caso. Note-se que listas ou tuplas que passam pela interface entre a camada SG-Atr e o nível semântico são consideradas estruturas de nível semântico e não de

implementação, pois não correspondem às i-tuplas ou i-listas que armazenam objetos, mas sim às estruturas semânticas que serão apresentadas às aplicações. Na camada SG-Atr existem 4 módulos dedicados ao tratamento de atributos:

SG-Atr-Tupla - manipula um conjunto não vazio de atributos monovalorados, segundo uma estrutura de tupla. Esse módulo implementa a manutenção da estrutura de i-tuplas;

SG-Atr-Lista - manipula atributos segundo uma estrutura de lista, onde cada célula da lista pode ser um atributo monovalorado ou uma estrutura de dados (que poderia ser por exemplo uma i-tupla, um vetor ou outra lista). A figura 4.2 exemplifica a utilização desta estrutura por parte de um objeto, onde as listas de atributos compõem-se de células de tuplas. Atualmente não está implementado o suporte para outras estruturas de dados além de lista (como por exemplo conjuntos ou vetores). No entanto a expansão para suportar outras estruturas de dados foi considerada, com a implementação do módulo através de uma estruturas de classes. Esse módulo implementa a manutenção da estrutura de i-tuplas;

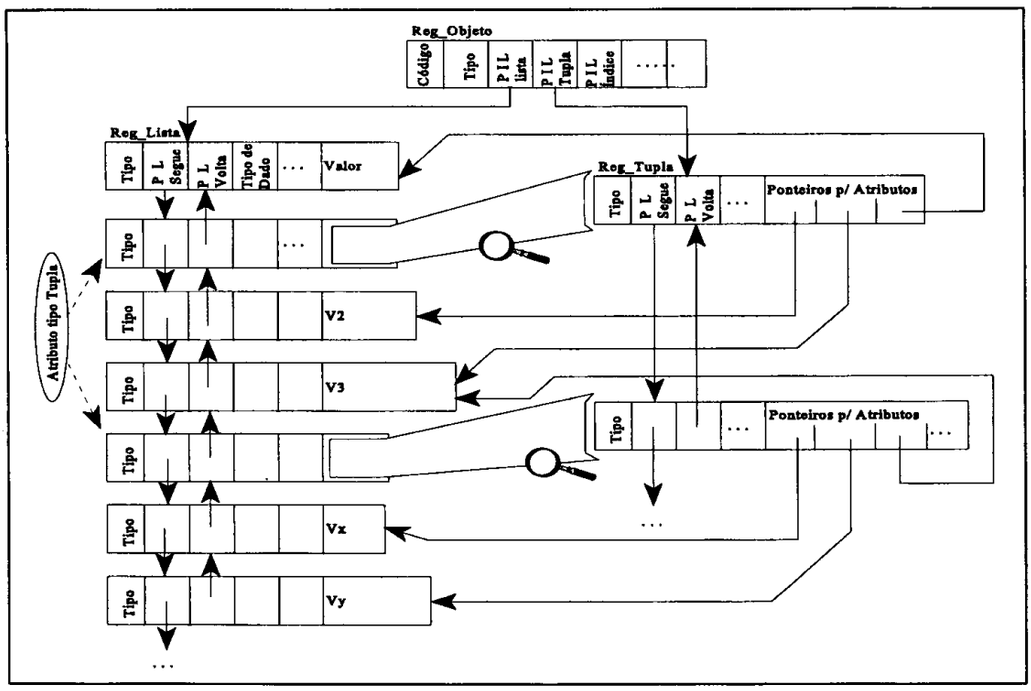


Figura 4.2: Estrutura de listas e tuplas para atributos de objetos.

SG-Atr-Índice - permite definir e manter uma estrutura de índices sobre um atributo ou uma tupla de atributos (tuplas indexadas devem ser definidas como tuplas na estrutura de instanciação do tipo do objeto). A implementação desse módulo não está prevista no presente trabalho. Uma estrutura de indexação espacial está sendo definida para ser suportada pelo NuGO;

SG-Atr-Blob - permite que um atributo longo possa ser recuperado por partes, seqüencialmente. As operações de transferência ocorrem através de *buffers* de tamanho definidos pelo nível semântico. Atributos definidos como BLObs não podem ser indexados, e portanto não são suportados pelo módulo SG-Atr-Índice. Para que sejam acessados, devem antes ter um "identificador de BLOb" - BLOId, temporário, solicitado ao sistema. Através desse identificador, os BLObs são tratados de forma homogênea pelos módulos SG-Atr-Tupla e SG-Atr-Lista. Cada BLOId representa uma área de transferência independente para acesso aos valores dos Blobs. Este módulo também não está implementado e, portanto, não é atividade prevista para o escopo desse trabalho.

4.3.1.3 Suporte para composição de objetos

Um quinto módulo da camada SG-Atr, o módulo SG-Atr-Composição realiza todas as operações vinculadas ao conceito de objeto em si. Incluem-se nele as operações de criação de objetos e tipos definidas na seção 4.3.1.1. Note-se que, apesar de os atributos poderem ser concatenados ao formar tuplas, isso não caracteriza a "composição de objetos". Os serviços que propiciam o suporte a objetos compostos estão incluídos nesse módulo, uma vez que, para permitir a definição de objetos, deve-se indicar a posição do objeto na estrutura de composição de objetos da base de dados como um todo. Isso é feito através de um particionamento físico do espaço de armazenagem da base de dados, de maneira que cada objeto "habite" em exatamente uma partição. A essa partição, que pode ser física ou lógica, denomina-se de colônia no modelo SIRIUS[Bia_96].

Esse módulo foi desenvolvido especificamente para suportar os conceitos advindos das abstrações de classificação e composição, e não teria um equivalente num núcleo que suportasse apenas o modelo relacional, pois tais abstrações não são reconhecidas nesse modelo. Assim, este módulo atua na instanciação de objetos e na alocação de espaços físicos na região onde o objeto habita. Com esse recurso suporta-se o conceito de composição física de objetos eventualmente usada por um modelo de dados (como é o caso do modelo SIRIUS), para cujo suporte se pretenda utilizar o NuGO. Num caso limite, o suporte para composição de objetos do núcleo pode ser desativado, definindo-se toda a base como uma única partição. Assim, procura-se preservar a independência do núcleo em relação ao modelo de dados suportado pelo gerenciador.

4.3.2 Subsistema de gerenciamento de registros lógicos

Na camada SG-Log os objetos e seus componentes são manipulados como porções menores de memória, denominadas de registros lógicos - RegLog, distribuídas logicamente através de páginas de memórias, denominadas de blocos lógicos. Nessa camada, um bloco lógico é armazenado em um registro real - RegReal obtido da camada inferior SG-Real, segundo uma formatação adequada para o armazenamento dos registros lógicos.

Os tipos de RegLog manipulados pela camada SG-Log são os seguintes:

RegObj: Armazena um registro de objeto. Todo objeto possui exatamente um registro desses, o qual mantém as informações da estrutura interna de armazenagem de objetos, tais como seu tipo, OId e ponteiros de acesso de i-tuplas e i-listas;

RegAtr: Armazena uma célula da i-lista de atributos de um objeto. Cada RegAtr armazena um valor de atributo de um objeto ou uma estrutura de dados (sem os atributos dessa estrutura, que são por sua vez individualmente mantidos em outras células da i-lista);

RegTupla: Armazena uma célula de i-tupla. Cada RegTupla armazena os ponteiros de acesso às i-tuplas e i-listas. Essa estrutura integra fisicamente todas as células das i-listas que armazenam os atributos que compõem esta i-tupla;

RegEstr: Armazena uma célula de estrutura de dados. Pode ser visto como conjunto de dados que organiza uma coleção de atributos segundo uma estrutura de dados. A estrutura fundamental do Núcleo é a lista de atributos de objetos, porém poderá ser utilizada para a armazenagem de vetores, árvores, etc. É a estrutura básica para a manutenção das tuplas, atributos multivalorados e índices;

RegÍndice: Armazena um nó da estrutura de índice a ser suportada pelo NuGO;

RegCont: Registro de continuação. Se um RegEstr ou RegÍndice necessitar mais espaço do que o provido por um único RegFis, um RegCont poderá ser utilizado para a expansão da estrutura. Um RegCont poderá ser utilizado também para a armazenagem de valores de atributos muito grandes (Blobs).

RegSis: Registro de estrutura da própria base. Não é utilizado para armazenar dados do usuário, mas para suportar a própria estrutura da base de dados. O registro de índice zero é sempre um RegSis.

O agrupamento (*clustering*) de RegLog provido pela camada SG-Log destina-se a incrementar a taxa de acertos de buscas de registros no *cache* mantidos pela camada SG-Real. No entanto, para que os registros lógicos possivelmente necessários sejam os que efetivamente estarão juntos, é preciso que os registros lógicos sejam agrupados segundo algum critério dependente da semântica das aplicações. Como o Núcleo nesse nível não dispõe dessa semântica, a camada SG-Log provê o gerenciamento dos dados que compõem cada objeto, como os atributos se agregam, como os objetos se compõem de outros e como estão classificados, segundo a definição dos tipos de objeto.

Os registros de tipo RegObj são de tamanho fixo, portanto cada base possui um número fixo, pré-determinado, de RegObj em cada bloco lógico, o qual depende apenas do tamanho adotado para o RegFis daquela base. Os registros lógicos de tipo RegAtr, RegEstr, RegÍndice e RegCont são registros de estrutura de tamanho variável, mas mesmo, assim, é feita uma estimativa da quantidade máxima admissível de RegLog. Como todos os RegLog de cada bloco lógico têm todos o mesmo tipo, numa base com determinado tamanho de RegFis, todos eles têm um número pré-definido de RegLog de determinado tipo que pode ser, no máximo, armazenado em cada RegFis. Mais do que isso, esses números são decididos na criação da base, e permanecem constantes durante toda sua existência.

A identificação dos RegLog de qualquer tipo é feita imaginando-se que a base de dados toda é composta apenas de registros de um mesmo tipo contando-se sequencialmente os RegLog que podem ser armazenados. Assim, para se localizar o registro **x** do tipo **y**, divide-se **x** pela constante "número de RegLog de tipo **y** que cabe num RegFis". A parte inteira do resultado é o índice do RegFis em que esse RegLog está armazenado e o resto indica qual dos RegLog armazenados nesse RegFis corresponde ao do registro **x** sendo procurado. Deve-se notar que dois RegLog de tipos diferentes podem ter o mesmo número, porém a identificação de cada RegLog de determinado tipo é unívoca.

Um outro objetivo dessa camada é adotar estratégias capazes de proporcionar um índice de ocupação de memória satisfatória pelos objetos e seus atributos nos blocos lógicos, com um número mínimo de acessos ao *cache*. Assim, é desejável que os RegLog relacionados estejam o mais próximo possível um do outro e que, para isso, permitam provocar um crescimento racionalizado da base de dados. Isto deverá ocorrer de modo que os espaços disponíveis para os RegLog tenham prioridade na ocupação antes de se alocar um novo bloco lógico.

Com o objetivo de atender o agrupamenento dos RegLogs relacionados e proporcionar estratégias de ocupação racionalizada pelos RegLogs nos blocos lógicos, duas concepções para o SG-Log foram propostas e implementadas. Essas propostas basicamente afetam a estrutura que compõe o bloco lógico. Elas estão definidas do seguinte modo:

SG-Log1 - denominação dada a uma das propostas, na qual todo bloco lógico é capaz de receber os RegLogs de tamanhos variáveis e fixos, sendo que os RegLogs de tamanho fixo são especialização dos RegLogs de tamanho variável, ou seja, existe uma única estrutura de bloco lógico capaz de suportar qualquer tipo de RegLog. A figura 4.3 mostra a estrutura de um bloco lógico com registros lógicos de tamanhos variáveis. Nesse caso, existem dois tipos de blocos lógicos, um com a estrutura de controle e espaço para receber RegLogs denominado de Bloco de Controle - BC; e o outro denomindado de Bloco de *Overflow* - BO e utilizado para armazenar os RegLogs que excederam o espaço disponível num BC;

SG-Log2 - a outra proposta, que se utiliza de dois tipos de blocos lógicos, um que armazena somente RegLogs de tamanho fixo, e outro tipo que armazena somente RegLogs de tamanho variável. Portanto, uma estrutura de bloco lógico para atender cada um dos tipos de RegLogs. A estrutura usada para

receber os RegLogs de tamanhos variáveis é a mesma que em SG-Log1. A diferença está na estrutura para o armazenamento de RegLogs de tamanho fixo. A figura 4.4 mostra um conjunto de blocos lógicos com quatro RegVirts, e esses com dois RegLogs de tamanhos fixos. Os blocos lógicos e os RegVirt têm uma estrutura de controle (cabeçalho) que servem, respectivamente, para controlar os RegVirts armazenados num bloco e os RegLogs armazenados num RegVirt.

Em ambas as concepções, utiliza-se do conceito de "Linha" que permite formar uma coleção de informações lógicas inter-relacionadas e armazenadas nos vários blocos lógicos. Ao utilizar-se o SG-Log2, tem-se adicionalmente o conceito de "Registros Virtuais" que consiste no agrupamento de RegLogs armazenados num bloco lógico, ou em metade, um quarto, um oitavo etc. de um bloco lógico. Uma linha, no SG-Log1, é uma coleção de um ou mais RegLogs e, no SG-Log2, uma linha é uma coleção de registros virtuais - RegVirts.

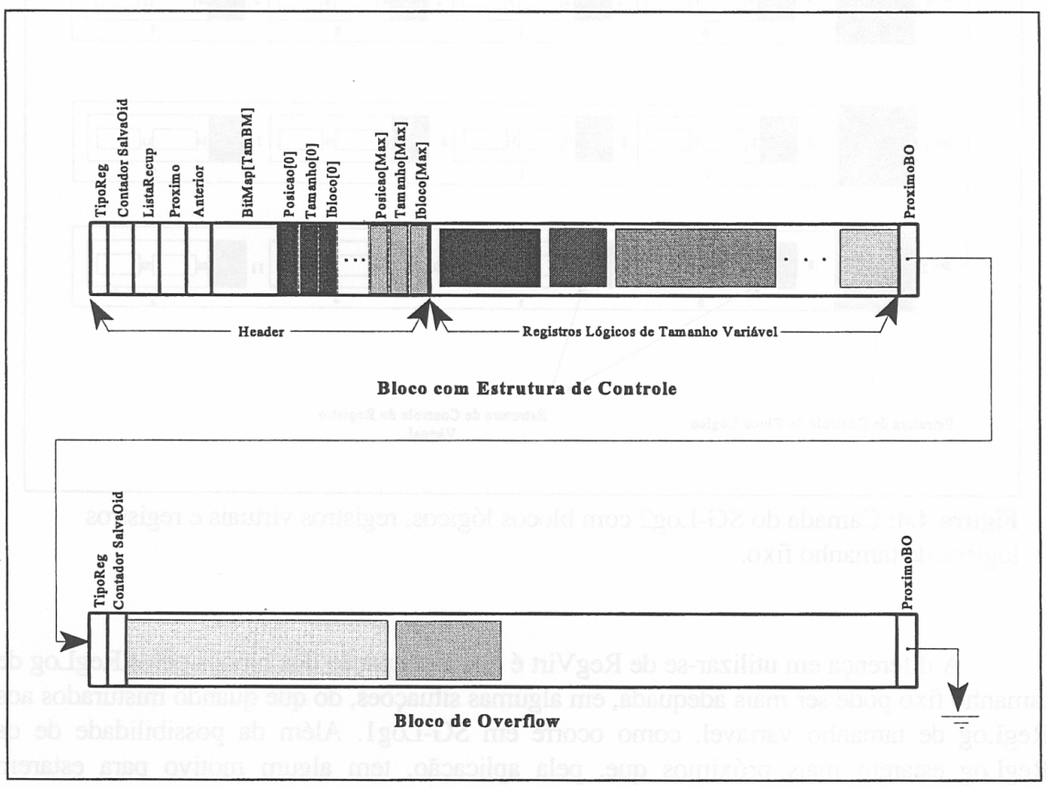


Figura 4.3: Blocos com registros lógicos de tamanho variável

Uma linha armazena um número qualquer de RegLogs, todos do mesmo tipo, que têm algum motivo para estarem juntos, e isto vale para ambas as concepções de SG-Log. Entre esses motivos estão os seguintes:

- Todos os RegObj que formam uma dada colônia definem a linha de objetos que compõem essa partição, ou coleção de objetos;
- Todos os Atributos de um mesmo objeto compõem a linha de atributos desse objeto;
- Todas as i-tuplas de um mesmo objeto compõem a linha de i-tuplas desse objeto;
- Cada nó da estrutura de Índices deverá correspondender a uma linha de RegÍndice;
- Cada atributo Blob deverá corresponder a uma linha de atributo longo.

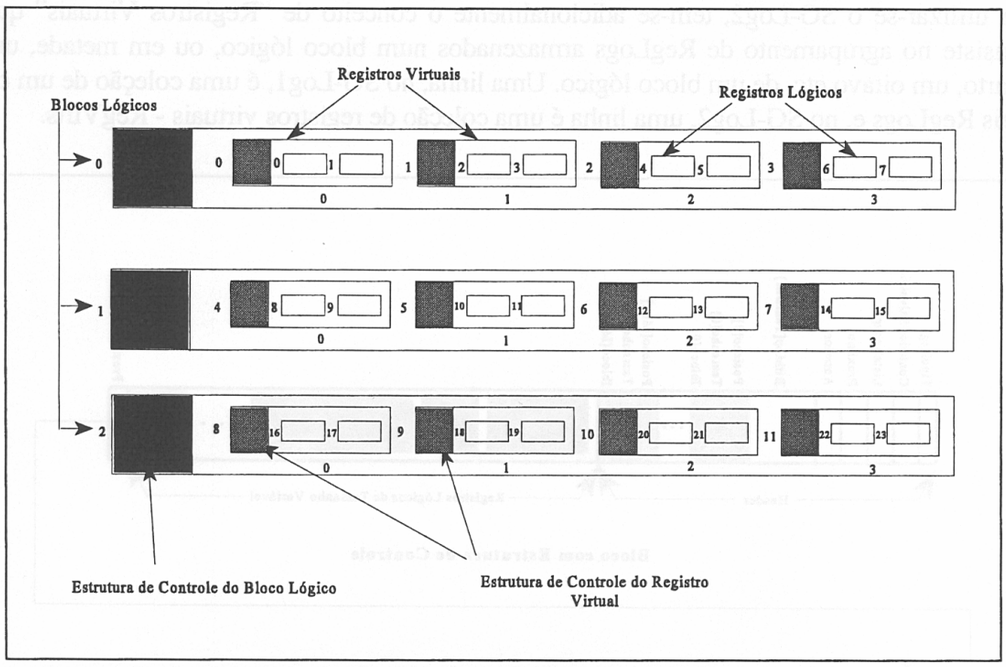


Figura 4.4: Camada do SG-Log2 com blocos lógicos, registros virtuais e registros lógicos de tamanho fixo.

A diferença em utilizar-se de RegVirt é que a ocupação dos blocos pelos RegLog de tamanho fixo pode ser mais adequada, em algumas situações, do que quando misturados aos RegLog de tamanho variável, como ocorre em SG-Log1. Além da possibilidade de os RegLog estarem mais próximos que, pela aplicação, tem algum motivo para estarem relacionados. No entanto, em termos de implementação, a adição de RegVirt e a separação no tratamento dos tipos de blocos lógicos acarreta um aumento de complexidade no código necessário para esse suporte. Com o objetivo de medir o desempenho das duas possibilidades de implementação para o SG-Log, no capítulo 5 é apresentado um conjunto de testes que as envolve.

A implementação dos dois módulos **SG-Log1** e **SG-Log2** é também um exemplo de como a arquitetura do ambiente SIRIUS permite a experimentação de diversas alternativas para uma mesma funcionalidade, pois ambos correspondem as alternativas (mutuamente

exclusivas para uma base de dados durante toda sua existência) para a realização de uma só operação.

4.3.3 Subsistema de gerenciamento de registros reais

Essa camada é a mais básica do Núcleo Gerenciador de Objetos do Gerenciador SIRIUS, onde estão as estruturas de dados de suporte às principais funcionalidades dum sistema gerenciador de banco de dados orientado a objetos, como, por exemplo, o gerenciamento de registros físicos, o gerenciamento de *buffers*, o gerenciamento de transações, bloqueio e recuperação. A figura 4.5 mostra as principais estruturas e funcionalidades suportadas pelo SG-Real. Cada uma dessas atividades desempenhadas pelo SG-Real são descritas em detalhes a seguir.

4.3.3.1 Gerenciamento de registros físicos

Fisicamente, uma base de dados armazenada pelo Núcleo Gerenciador de Objetos é uma coleção indexada de registros físicos. Isso permite que a base seja armazenada numa partição em disco acessada diretamente pelo Núcleo, ou num arquivo gerenciado pelo sistema operacional. Atualmente, dado o caráter de protótipo do sistema, apenas acha-se em operação o suporte para bases armazenadas em arquivos gerenciados pelo SO.

Todos os registros físicos (RegFis) de uma base têm o mesmo tamanho, embora diferentes bases possam tê-los de tamanhos diferentes. Para agilizar e permitir diminuir a carga de trabalho imposta sobre o sistema operacional, esse tamanho é sempre um múltiplo do tamanho do setor físico do disco utilizado, e cada registro deve ocupar setores consecutivos no disco, de maneira que uma operação de leitura/escrita de um RegFis possa ser feita sem interrupção. Os RegFis são numerados de zero em diante. Toda informação da base é armazenada apenas numa seqüência de RegFis. Pode-se utilizar um ou mais RegFis para armazenar cada unidade de informação ou concatenar diversas unidades de informação num mesmo RegFis.

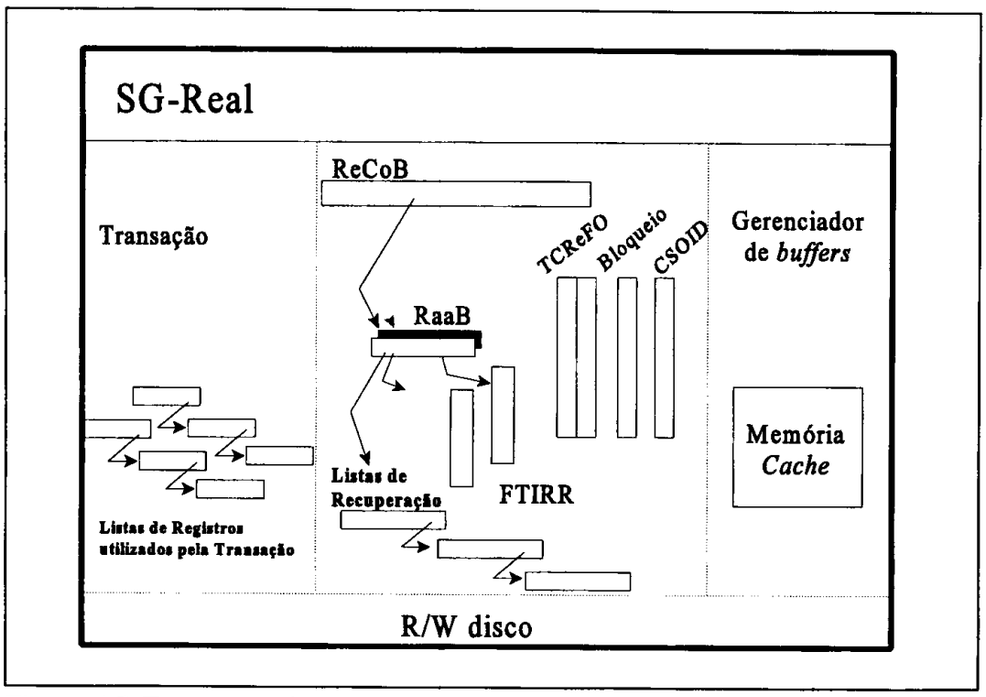


Figura 4.5: Estruturas e funcionalidades suportadas pelo SG-Real

O acesso à base de dados é feito através da leitura/escrita de um RegFis inteiro. Essa operação somente pode ser executada por esta camada, a qual traz os RegFis necessários para a memória, numa estrutura de *cache*. Existem critérios para que cada RegFis armazene sempre informações de um mesmo tipo. A camada SG-Real apenas reconhece que existem diversos tipos de informação, mas não quais são esses tipos, o que é reconhecido pela camada SG-Log. Cada tipo de informação é armazenada num tipo de bloco lógico, cuja estrutura é definida *a priori* pelo sistema. Para a camada SG-Real, um RegFis é identificado através do seu 'índice real". A associação endereço de RegFis e índice real é efetuada através de estruturas necessárias para suporte ao gerenciamento de transações. Ao receber um índice real, o registro físico passa a ser denominado de registro real - RegReal. A associação entre um índice real e o número sequencial de um RegFis em disco varia conforme as transações vão sendo executadas, permitindo que uma mesma informação (por exemplo, um objeto) possa mudar de lugar onde é gravada em disco.

4.3.3.2 Gerenciamento de transação

É a atividade responsável pelo controle das transações efetuadas durante a utilização do gerenciador SIRIUS pelas aplicações. Enfeixadas numa transação, um conjunto de operações pode provocar atualizações ou não nos diversos RegReal e respectivos RegFis em disco. Para isto, utiliza-se como suporte básico, de um conjunto de estruturas de dados e componentes que permitem garantir as propriedades ACID para as transações em execução. Estas estruturas e componentes são: uma estrutura contendo informações da configuração da

base de dados; uma estrutura de armazenamento do estado atual e anterior da base de dados; o gerenciador de *buffers*; estruturas globais do SG-Real, que consistem nas listas de recuperação de RegFis e RegReal disponíveis, a cópia em memória dos fragmentos de tabelas de indexação de registros reais - FTIRR e a tabela de conversão de registros reais para físicos e *overflow* - TCReFO, explicadas a seguir.

Além desses elementos, existem nesta camada uma estrutura de controle de bloqueio para os RegReal e uma estrutura de suporte ao gerenciamento de Oid para os objetos utilizados pelo Núcleo.

Com o propósito de entender como funciona o SG-Real, deve-se explicar como são utilizados os RegFis para armazenar a estrutura da base de dados. Um RegFis, usualmente o de índice zero (registro de tipo RegSis), denominado de ReCoB - Registro de Configuração da Base, define a estrutura da própria base e é sempre armazenado numa posição pré-definida do arquivo/partição. Esse registro contém informações sobre a versão do sistema, o tamanho da base, referências a outros dois registros de controle da base etc. Os dois registros mencionados, denominados de Registros de Gerenciamento do Estado Atual e Anterior da Base de Dados - RaaB, possuem seus endereços armazenados pelo ReCoB e contém informações sobre a estrutura corrente da base de dados. Ambos representam a mesma informação, mas um deles representa a informação atual, e o outro a informação imediatamente anterior. Esses registros controlam também os locais onde se encontram em disco os Fragmentos das Tabelas de Indexação dos Registros Reais - FTIRR, que armazenam os índices reais dos RegFis utilizados pelas transações. Um índice real de um RegFis não define a posição em que esse registro é armazenado fisicamente, mas sim onde, nos FTIRR, está guardado o endereço dessa armazenagem. Assim, para cada RegFis obtido pelo Gerenciador de *buffers*, há um índice real associado: se ele for válido, o correspondente RegReal existirá e armazenará dados da base. Se for inválido, corresponderá a um registro que teve seu lugar alterado, e que está atualmente disponível na lista de recuperação. Como nessa camada a manipulação de todas as informações pelas transações são efetuadas através dos índices reais, denomina-se registro real - RegReal ao respectivo RegFis associado a esse índice.

Assim, sempre que é solicitado um RegReal, verifica-se se ele está associado à transação corrente. A transação está associada a um ou mais objetos compostos que têm determinadas suas restrições de acesso. Se esse RegReal estiver associado à transação corrente, a operação será liberada. Caso o RegReal não esteja associado à transação corrente, a operação será negada, retornando-se um código de violação de acesso.

Se a operação tiver sido liberada, o gerenciador de *buffers* determinará se o respectivo RegFis solicitado já está em memória, no *cache*, utilizando uma estrutura de dados mantida por completo na memória. Caso esteja em memória, seu endereço no *cache* é devolvido para o SG-Real . Caso não esteja, utiliza-se um algoritmo para a localização e recuperação do RegFis em disco. Escolhe-se também um registro que esteja no *cache*, para ser removido e abre-se espaço para que o registro recém-solicitado seja trazido para o *cache*. O RegFis solicitado é então obtido em disco e colocado na posição recém liberada. O sistema de gerenciamento de *buffers* administra uma região de memória, com capacidade

para um número pré-determinado de RegFis, trazidos do disco sob solicitação da camada SG-Real. A solicitação de um RegFis é feita através de seu índice.

O registro eleito para ser retirado do *cache* é avaliado para determinar se, durante sua permanência no *cache,* ele foi modificado na memória. Se foi, ele será regravado num novo RegFis do disco, mantendo-se a cópia original intocada e continuará sendo ainda a versão do registro indicada para a TCReFO. O novo endereço em disco é mantido apenas em memória, onde será localizado no caso de solicitações subseqüentes. O gerenciamento desses registros é efetuado através da TCReFO, essencialmente um par de vetores paralelos, cujo tamanho corresponde ao do maior índice de registro real utilizado na base. Um desses vetores contém os índices físicos dos registros lidos e/ou requisitados pela transação; o outro contém os respectivos índices dos registros alterados. Portanto, para os registros alterados verifica-se no mesmo índice da TCReFO as referências de suas versões anterior e modificada. Os registros não modificados têm o mesmo valor em ambos os vetores.

Quando uma transação é finalizada com êxito (*commit*), os novos endereços dos RegReal associados a essa transação são passados para novas posições na TCReFO, indicando a versão definitiva dos RegReal. Isto ocorre ao efetuar-se a cópia dos índices dos registros alterados armazenados num vetor para outro na TCReFO. Assim, ao se efetuar uma operação *commit,* todos os registros dessa transação presentes no cache e que tinham sido alterados são re-escritos em disco, em novos RegFis. Não acontece nada quanto aos registros lidos pela transação e os alterados (no *cache* ou já removidos dele anteriormente) têm seus endereços re-escritos na TCReFO. A cópia desse vetor da TCReFO é efetuada para o correspondente FTIRR, procedendo-se a atualização dos respectivos endereços reais alterados. Em caso de falha da transação, as alterações não serão consideradas e os RegFis em disco que contêm as atualizações serão colocados em disponibilidade na lista de recuperação. Os antigos registros, ou seja, os registros anteriores à modificação são colocados na lista de recuperação, inclusive os que compunham o FTIRR anteriormente.

Como apenas a operação de gravação final dos registros RaaB redireciona de fato os endereços dos RegReal para suas novas posições, a atomicidade das transações é garantida. Em caso de cancelamento (*abort*) de transações, a estrutura mantida em disco é recuperada para a memória e os registros em disco que estavam sendo utilizados para a armazenagem dos novos RegReal são colocados na lista de recuperação. O papel das listas de recuperação de registros físicos livres ou liberados é evitar o crescimento da base com "buracos". Assim, conforme abram-se as transações, utilizam-se os registros das listas, reaproveitando-os e evitando-se o crescimento da base com o desperdício de espaços físicos.

Para cada transação aberta, têm-se sob seu controle um conjunto de listas, entre elas destacam-se quatro: uma lista com todos os registros lidos por esta transação; uma lista dos registros escritos; uma de registros alocados (novos registros incluídos na base) por essa transação; e outra de registros liberados. Assim, ao efetuar as operações de *abort* ou *commit* por uma transação, efetua-se o devido tratamento para as suas respectivas listas, além da TCReFO, dos FTIRR e das listas de recuperação.

Note-se que, sempre que uma transação é abortada, os registros escritos no *cache* são re-lidos de sua posição original. Quando uma transação é finalizada com êxito, os registros modificados que já tinham sido removidos do *cache* não são mais afetados. Os que ainda estão no *cache* são escritos em disco em novas posições (equivalente a serem removidos do *cache* para serem substituídos), e as novas posições de todos esses novos registros são atualizadas numa nova FTIRR que representa todos os registros da base dessa transação, das demais em andamento (mas ainda sem modificar nenhum dado, mesmo os já alterados pela transação em andamento) e dos que não estão participando de nenhuma transação atual. Essa nova FTIRR será a indicada pelo RAAB a ser escrito em disco, como o mais atual.

4.3.3.3 Recuperação de endereços para a armazenagem física

A camada SG-Real efetua uma associação de endereços físicos para índices reais que representa a conversão de endereços de armazenamento em disco (permanente) para índices utilizados para a referenciação pela camada SG-Real (transientes). O trabalho de conversão é imediato, pois se resume a um acesso ao vetor pelo índice real do correspondente RegFis desejado. Esse vetor, denominado de TCReFO, tem tamanho limitado pelo caráter de protótipo dado a essa implementação do NuGO, mas facilmente extensível para uma dimensão apenas limitada pela capacidade disponível de memória. A escolha dessa estratégia de recuperação de endereços para RegFis está atrelada às políticas utilizadas para a manutenção das atualizações pelas transações, que são as páginas *shadow* e o gerenciamento de OIds não recuperáveis suportados pelo Núcleo. Todo RegReal requisitado pelas camadas superiores deve ter seus endereços físicos obtidos através dessa recuperação sobre a estrutura TCReFO e, posteriormente solicitado ao gerenciador de *buffers* que se encarregará de torná-lo disponível. Após a obtenção do conteúdo no RegFis do RegReal desejado, uma cópia é efetuada e transferida de modo a atender às solicitações efetuadas pelas camadas superiores.

4.3.3.4 Gerenciamento de *buffers*

O NuGO efetua todos os seus acessos a disco através de requisições ao gerenciador de *buffers* mantido pela camada SG-Real. Um número limitado de páginas de memória é mantido pelo gerenciador de *buffers*, onde cada uma delas deverá conter informações armazenadas nos RegFis em disco. O tamanho dessas páginas é o mesmo especificado para os RegFis para a base de dados corrente. As dimensões para o funcionamento do gerenciador de *buffers*, assim como para a base de dados, poderão ser configuradas de acordo com as aplicações no momento da criação duma base de dados.

Com o objetivo de otimizar o acesso ao disco, o gerenciador de *buffers* utiliza-se de uma política de escolha para a retirada de blocos definida através duma adaptação dos critérios LRU (*Least Recently Used*) e LFU (*Least Frequently Used*) existentes na literatura. O algoritmo e a estrutura de dados para suportar esta adaptação foram propostos e implementados inicialmente como suporte ao gerenciamento de um sistema multiusuário distribuído[Tra_87], passando posteriormente a ser denominado Algoritmo Seleção Natural[San_98]. A partir dessa adaptação, o critério passa a ser o seguinte: substituir a

página acessada há mais tempo, dentre as que tiveram menor freqüência de acesso, gerando um critério para substituição para páginas com mesma freqüência e aperfeiçoando o critério LRU, retirando-lhe apenas as páginas com pouco acesso. Em outras palavras, é feita a seleção natural da página com menor importância definida como sendo a página acessada há mais tempo dentre as menos freqüentadas.

Antes de se efetuar uma substituição de páginas, o gerenciador de *buffers* verifica se a página a ser retirada foi alterada por alguma transação. Se isso ocorreu, a página deverá ser gravada em disco antes de ser retirada do *buffer*, viabilizando-se a persistência dessas alterações numa posterior operação *commit* pela respectiva transação. Uma outra situação envolvendo as páginas escritas por uma transação é na ocorrência de uma operação de *commit*. Nesse caso, todas as páginas alteradas pela transação deverão ser retiradas dos *buffers* após serem armazenadas em disco. Em qualquer caso, um novo RegFis é alocado para a gravação da página. A substituição do endereço de armazenagem do RegReal para o novo RegFis é feito durante a operação *commit*.

4.3.3.5 Bloqueio e controle de concorrência

A camada SG-Real oferece o suporte ao bloqueio físico, ou seja, uma operação de bloqueio é efetuada por RegReal sempre que uma transação desejar alterar algum RegLog. Se a camada SG-Log requisitar uma operação de atualização num RegLog, essa informação será passada para a camada SG-Real. Nessa camada é efetuado o tratamento adequado para que o RegReal com o respectivo RegLog seja bloqueado com êxito e consequentemente possa atender à operação desejada pela camada superior. Tal procedimento evita que duas transações possam efetuar operações de escrita simultâneas nesse mesmo RegReal ou que uma transação possa acessar as informações que estão sendo alteradas por outras transações, afetando-lhe a propriedade de isolamento e impedindo que as transações possam ser executadas concorrentemente de modo a garantir a capacidade de serialização. Como os critérios de concatenação de informações num mesmo registro, através do sistema de linhas implementado pela camada SG-Log, garante a inclusão num RegFis apenas de dados semanticamente agregados, o bloqueio executado dessa maneira é consistente e significativo para os requisitos de alto nível da aplicação.

Os modos de bloqueio utilizados pelo Núcleo e implementados pela camada SG-Real são o de leitura (compartilhado) e o de gravação (exclusivo) e sua utilização é implementada através de uma estrutura que armazena o tipo de bloqueio requisitado pela respectiva transação num RegReal. Assim, quando uma transação desejar efetuar uma operação de escrita num determinado RegReal, primeiro verifica-se se o mesmo não se encontra bloqueado para a escrita ou a leitura por outra transação. Portanto, uma operação de escrita só estará liberada para uma transação se o RegReal que contém o RegLog a ser alterado não estiver sendo utilizado, de qualquer modo por outra transação. Uma operação de leitura por uma transação em um RegLog terá êxito se o RegReal que o contém não estiver bloqueado para a escrita por outra transação.

4.3.3.6 Recuperação

A camada SG-Real proporciona o suporte necessário à recuperação de falhas dos seguintes tipos: falhas de transação (*abort*) e queda do sistema. Para isso, utilizam-se estruturas de dados que garantam a atomicidade das transações caso ocorra qualquer uma dessas falhas. Se uma transação for cancelada através duma operação de *abort*, ou se ocorrer queda no sistema, de modo que as informações (*buffers*, listas de recuperação etc.) da memória principal, utilizadas pelas transações, se percam, o Núcleo, por intermédio desta camada, deverá garantir a capacidade de reiniciar a base de dados colocando-a num estado consistente. Assim, em ambos os casos, as atualizações que não foram concluídas não deverão ser propagadas de modo persistente na base de dados, o fato que, de modo indesejável, refletiria situações parciais das transações que estavam em execução no momento da falha.

Essa camada utiliza-se de diretório *shadow* para a manutenção das atualizações executadas pelas transações. Através do armazenamento das atualizações em espaços diferentes dos quais encontra-se armazenado o item lógico original, é possível preservar o estado anterior à atualização e, apenas no momento da efetivação, ou seja, numa operação *commit*, as atualizações passariam a serem consideradas definitivas, assim, tomando o lugar do até então original. No SG-Real, o processo de efetivação dos RegReal com as atualizações é concretizado através da alteração dos endereços dos respectivos RegFis, a tal modo que aquele que contém as atualizações passa a ser o definitivo e o antigo é liberado para ser reaproveitado pela base de dados. Note-se que o sistema pode "cair" mesmo durante a fase de finalização de uma transação. Portanto, todas essas operações de substituição de endereços físicos têm de ser executadas numa única operação, indivisível. Se o sistema cair antes dessa operação, nenhuma das substituições, e nenhuma atualização solicitada pela transação será passada para a base de dados. Se essa operação for executada, todas as alterações da transação serão passadas para a base.

Essa operação é realizada da maneira descrita a seguir. Os respectivos endereços de RegFis, o que contém a atualização e o original, são ambos mantidos no correspondente índice real da estrutura de dados TCReFO. Quando uma transação é terminada, os registros anotados nas quatro listas internas da transação são processados e passados para a estrutura TCReFO e para a cópia em memória (ativa) das estruturas FTIRR. Nessas FTIRR encontra-se a associação efetuada pela SG-Real entre um endereço físico e índice real, cuja preservação conserva a consistência das informações manipuladas na base de dados. No momento da efetivação (*commit*) de uma transação, os respectivos endereços da FTIRR ativa (na memória) com os RegReal atualizados por essa transação, são armazenados num novo RaaB em disco denominado RaaB futuro, pois conterá informações das futuras atualizações efetivadas agora na base de dados. No caso de falha, o outro FTIRR já gravado em disco denominado corrente será usado para recuperar a base de dados no estado anterior às atualizações perdidas. Como esse FTIRR está gravado na base, qualquer problema no sistema fará com que a base recupere o FTIRR corrente.

A preparação do FTIRR futuro na memória corresponde à substituição dos endereços dos registros anotados nas listas da transação encerrada. Um novo FTIRR inteiro (com os endereços de todos os registros da base de dados) é gravado então em disco, num

novo endereço, distinto dos utilizados pelos dois FTIRR (corrente e anterior) já existentes em disco. Note-se que um FTIRR pode ser maior do que o espaço disponível num único RegFis. Assim, um FTIRR é composto por um conjunto de RegFis, segundo uma estrutura em lista sequencial mantida nos RegFis utilizados. Depois de preparado o novo FTIRR futuro, todos os RegFis necessários são gravados em disco e o endereço do registro cabeça da lista é gravado na cópia corrente do Raab futuro em memória, a seguir, gravado em disco. Assim, se até o registro do RaaB ser gravado a base "cair", o registro do RaaB em disco estará apontando para o FTIRR antigo, que esta utilizando outros registros físicos, que não foram ainda substituídos. Quando o novo RaaB é gravado no mesmo registro apontado pelo ReCoB, essa única operação de escrita em disco efetiva toda a substituição de dados da transação que encerrou.

O único ponto falho nessa estrutura é que a base ainda pode cair durante a escrita do RaaB, uma vez que a gravação de um RegFis em disco é uma operação (relativamente) demorada, apesar de indivisível. Para detectar se esse problema veio a ocorrer, e evitá-lo, o sistema mantém duas estruturas FTIRR e dois registros RaaB em disco permanentemente. Além disso, cada registro RaaB mantém um esquema de verificação de consistência, baseado na gravação de padrões de bytes com os valores 0, 1 ou 2, e um esquema de *check sum*. Os valores 0, 1 ou 2 são gravados sempre iguais num RaaB. Ou seja um RaaB grava zeros em várias posições do registro físico, outro RaaB grava uns nas mesmas posições, e o seguinte grava 2. O próximo RaaB grava zeros novamente, e assim por diante. Como existem somente dois RaaB em disco a cada instante, lendo ambos as combinações possíveis são 0 e 1, 1 e 2, ou 2 e 0. No primeiro caso, sabe-se que o mais recente é o que tem 1, no segundo o que tem 2, e no terceiro o que tem 0.

Assim, sempre que um gerenciador de base de dados é iniciado sobre uma base existente, a leitura de seus dois RaaB indica qual é o mais recente. Se a base foi antes desligada de modo correto, ou se ela não caiu a gravação de um registro RaaB, essa estrutura permite re-iniciá-la consistentemente, com o uso das estruturas apontadas pelo RaaB mais recente. Se ela caiu durante a gravação de um RaaB, é quase certeza que o registro que estava sendo gravado estará fisicamente danificado, e sua leitura será rejeitada pelo controlador do disco. Nesse caso o outro RaaB restante ainda permitirá que a base possa prosseguir normalmente. No caso improvável em que a base tenha caído durante a gravação do registro RaaB e se, ainda assim, o registro for legível, a gravação dos valores 0, 1 ou 2 repetidos em diversos pontos do registro, e o esquema de *check sum* serão então a barreira final que impedirá que qualquer inconsistência na base seja efetivada.

Embora o esquema de páginas *shadow* seja uma tecnologia já adotada em gerenciadores de dados, a maneira como ela é empregada aqui em conjunto com o esquema de memória cache e as estruturas RaaB e FTIRR, é inédito e traz uma significativa melhora de desempenho, quando comparada às implementações existentes. Tradicionalmente, o emprego de páginas *shadow* acarreta aumento na carga do sistema, pois implica em duplicar todas as páginas atualizadas por uma transação e, quando a transação termina, implica em ler essas páginas e re-escrevê-las nas posições adequadas. Nesse caso, a garantia de consistência é dada pela existência de cópia completa das atualizações da transação em disco, antes que a primeira página da base seja substituída: se a base cair, na re-inicialização o sistema pode prosseguir com o esquema de copiar todas as páginas já gravadas para as posições corretas.

No entanto, para cada página atualizada na transação existem pelo menos 3 operações em disco: 1) gravação da cópia, 2) leitura da cópia e 3) gravação no local correto.

Na implementação efetuada nesse trabalho, a operação 1) é feita integralmente, mas, como muitas das páginas já podem ter sido gravadas em disco durante operações anteriores que a removeram do *cache*, essa operação tende a ser distribuída durante a execução da transação, e isso, portanto, permitirá o sistema operar de modo mais homogêneo, incluindo o processamento paralelo do disco, das operações de processamento do *cache* e do restante do núcleo. A operação 2) não é necessária em absoluto, e a operação 3) é substituída pela gravação de um novo FTIRR. Como um único registro físico para o FTIRR pode-se gravar no mínimo milhares de ponteiros (dependendo do tamanho e estrutura da base de dados e dos RegFis), o número de registros a ser gravados tende a ser significativamente menor que o número de registros atualizados numa transação. Portanto, a técnica aqui desenvolvida representa significativo avanço para a implementação da técnica de memória *shadow*.

4.4 O esquema de gerenciamento de OIds no gerenciador SIRIUS

Pode-se definir que dois objetos são os mesmos de duas maneiras: se seus atributos possuem os mesmos valores - denominada Identificação Lógica ou LOId - *Logical Object Identification*; ou quando ambos são representações do mesmo objeto - denominado Identificação Física, Identificação por Referência, ou OId - *Object Identification*[Kho_89] [Kim_92][Cat_94]. OIds são códigos de acesso unívoco, não reaproveitáveis, gerados automaticamente pelo sistema, para identificar e localizar objetos. A maneira mais eficiente de utilizar um OId para localizar um objeto é utilizar o endereço de armazenagem do objeto como seu OId - no caso de objetos armazenados em disco, seu endereço físico em disco. No entanto, essa alternativa viola a regra de que o OId não pode ser reaproveitado se o objeto deixar de existir, pois a memória sem dúvida tem de ser reaproveitada.

As alternativas usualmente empregadas consistem em utilizar-se pares <OId, endereço> armazenados em estruturas de busca, tais como árvores do tipo B-*tree*. Estruturas em *hash*, embora aplicáveis em casos particulares, não são adequadas para Gerenciadores de Objetos genéricos, pois a quantidade de objetos é em geral muito grande e as chaves de partição não são eficientes. As árvores propiciam acesso rápido, até por que a chave de busca é um campo numérico, porém não evitam um determinado número de iterações para que a localização seja obtida. Dada a elevada taxa de acesso a objetos que solicitam de um gerenciador de objetos, esse mecanismo causa sobrecarga significativa em seu desempenho.

O núcleo do gerenciador SIRIUS procura resolver a incompatibilidade entre os requisitos de não reaproveitamento dos OIds e o uso do endereço de armazenagem de maneira a poder utilizar esse eficiente mecanismo e, ao mesmo tempo, tornar a reutilização de OIds impossível. Basicamente, a solução desenvolvida neste trabalho consiste em utilizar como OId uma estrutura que agrega dois atributos: um é o índice real correspondente ao endereço físico de armazagem do objeto em disco (EndObj), e o outro é um contador do número de vezes que esse endereço foi utilizado (ContObj). Na base, armazena-se como parte da estrutura interna de manutenção de cada objeto o contador de utilização ContObj dessa área de memória. Assim, sempre que um OId for passado como referência ao objeto,

seu endereço EndObj é diretamente utilizado para acessar o objeto. A seguir, o ContObj do OId é comparado com o armazenado na estrutura do objeto. Sendo o mesmo, considera-se o OId válido e o acesso é atendido. Sendo diferentes, considera-se que o OId é inválido, e rejeita-se o acesso.

Essa solução não seria universal, se qualquer endereço em disco pudesse ser o endereço de um objeto. No entanto, no caso da estrutura do núcleo do gerenciador SIRIUS, essa solução é exeqüível e prática. Isso porque o que se utiliza como o EndObj de um OId é o índice do RegLog que armazena o registro desse objeto - o atributo ContObj é mais uma informação contida nos RegLog e se refere a quantas vezes esse regLog foi reaproveitado para armazenar um objeto após ter sido eliminado o objeto anterior que ele armazenava. Sendo uma estrutura de tamanho fixo, se um objeto for removido, seu espaço será mantido, e a estrutura de RegVirt ou o bloco lógico, em que o RegLog está inserido, passa a indicar que o objeto acha-se disponível para a reutilização, mas o atributo ContObj permanece disponível, para ser incrementado e utilizado pelo próximo objeto que vier a fazer uso desse RegLog.

O gerenciamento dessa estrutura acrescenta sobrecarga à atividade do Núcleo, porém ela consiste em alguns poucos testes, sempre sobre valores numéricos, sem que ocorram iterações. Além disso, todas essas operações, embora rápidas, não ocorrem quando um objeto é acessado, mas apenas quando algum objeto é criado ou eliminado. As tarefas de alocação de espaço para novos objetos ou sua liberação são usualmente bastante mais custosas do que as necessárias para acessá-los. Assim, a sobrecarga causada pela necessidade adicional de gerenciamento dos valores de ContObj não causa impacto no desempenho do sistema. Mais do que isso, todas essas operações ainda são muito menores do que as necessárias em qualquer dos esquemas existentes de OId alternativos, como árvores B ou estruturas de *hash*.

4.5 Considerações finais

Os mecanismos adotados para a implementação do NuGO aqui descritos destinam-se a ser utilizados no Gerenciador SIRIUS, e, assim, diversas decisões de projeto foram tomadas com vistas a atender às necessidades do desenvolvimento dos módulos semânticos dos níveis superiores do Gerenciador.

O núcleo gerenciador de objetos implementado é genérico para atender ao desenvolvimento de ampla gama de gerenciadores de bases de dados, orientados a objeto ou não, sem restrições para que um particular modelo de dados seja adotado. Por exemplo, um gerenciador relacional pode ser construído com o uso apenas dos módulos de tupla e índice da camada SG-Atr, e as camadas inferiores. Esse enfoque é interessante, pois permite o suporte a gerenciadores puramente orientados a objetos, ao mesmo tempo que permite utilizar toda a tecnologia de construção de gerenciadores relacionais que atualmente possui a maior gama de algoritmos desenvolvidos. Assim, a solução apresentada pode ser utilizada para suportar o desenvolvimento de outros gerenciadores de bases de dados, mantendo a eficiência na recuperação de objetos armazenados.

Esse trabalho também teve o objetivo de servir como uma "bancada de experimentação" para o desenvolvimento de algoritmos e técnicas que contribuíssem para a criação de gerenciadores de dados orientados a objetos de nível industrial eficientes e confiáveis, sem contudo pretender-se que essa implementação resultasse em si num produto. Assim, a modularidade, expansibilidade e portabilidade do sistema sempre foi considerada essencial. Embora ainda não estejam disponíveis, diversos módulos foram projetados de maneira a que venham ter mais do que uma implementação, que podem ser plugadas individualmente para compor uma versão do gerenciador. Pôde-se efetuar a constatação disso ao executar o NuGO com as duas propostas de implementação para o SG-Log, em que os conjuntos de testes foram aplicados, analisando-se o comportamento do NuGO para cada uma das possibilidades, bastanto para isto "plugar" uma entre as propostas de cada vez. No que tange à funcionalidades já desenvolvidas, pretende-se que esse gerenciador possa ser utilizado como um produto, se não industrial, para ser utilizado por outros grupos de pesquisa dentro da comunidade acadêmica.

Além disso, outras contribuições também foram feitas, em particular a implementação de um núcleo de um gerenciador de base de dados que poderá ser usado efetivamente para a construção de um sistema gerenciador de base de dados orientado a objetos completo.

Capítulo 5

ASPECTOS DA IMPLEMENTAÇÃO

5.1 Considerações iniciais

Neste capítulo são apresentados detalhes da implementação dos módulos componentes do Núcleo Gerenciador de Objetos desenvolvido. Em cada um desses módulos mostra-se como estruturas e conceitos são utilizados através da descrição das primitivas que implementam as suas funcionalidades. São apresentados também alguns testes efetuados com o objetivo de validar a adequação dos resultados obtidos no trabalho.

Especificamente, são mostradas as primitivas e as estruturas que componentes dos módulos SG-Atrib, SG-Log (as duas propostas de implementação), SG-Real, além de um módulo simplificado para a criação duma base de dados SIRIUS. São apresentados os testes efetuados sobre as duas propostas para o módulo SG-Log. Também são apresentadas comparativamente as opções adotadas para as tecnologias implementadas neste trabalho em relação às existentes na literatura, ou seja, utilizadas por outros sistemas gerenciadores de dados.

O gerenciador de objetos SIRIUS adota a arquitetura cliente/servidor. A implementação do Núcleo Gerenciador de Objetos - NuGO vem sendo executada em linguagem C++, numa estrutura em camadas. Seu desenvolvimento tem sido efetuado utilizando os sistemas operacionais Windows e UNIX. O NuGO, parte fundamental do Gerenciador SIRIUS, atua como servidor de objetos para as camadas semânticas implementadoras do modelo de dados.

5.2 Criação da base de dados

Uma base de dados SIRIUS é inicialmente configurada através de um conjunto de parâmetros especificados durante a operação de sua criação. Essa configuração inicial é interpretada pelo Núcleo Gerenciador de Objetos - NuGO e utilizada para a parametrização do conjunto de estruturas de suporte de funcionamento de cada camada que o compõe. Uma vez definidos os parâmetros dessa configuração, as informações armazenadas em disco também permanecem no formato especificado por essa definição, o que impede que os valores para tais parâmetros possam ser modificados após a sua criação. Isso é feito para simplificar as operações de acesso e agilizar ao máximo essas operações. Como esses parâmetros referem-se a configurações de software e hardware (naturalmente estáveis durante a operação da base de dados), essa imposição não causa nenhuma restrição real para o sistema.

Uma base de dados SIRIUS é armazenada em disco, a qual pode ser, ou um arquivo único gerenciado pelo sistema operacional ou uma partição do disco. Embora a estrutura desenvolvida possa ser aplicada igualmente a um arquivo ou a uma partição, a flexibilidade de desenvolvimento oferecida por um arquivo orientou o desenvolvimento da versão protótipo aqui descrita para a utilização apenas de um arquivo gerenciado pelo sistema operacional. No entanto, como pode ser verificado pela descrição da estrutura feita a seguir, essa estrutura pode ser facilmente empregada para o gerenciamento de uma partição específica, numa versão de produção, dado que toda a base de dados é mantida num único arquivo, acessado exclusivamente através de blocos de dados de tamanho fixo endereçados através de um índice contínuo. Todos os blocos de dados têm o mesmo tamanho, e são chamados de registros físicos ou páginas. O índice de acesso de um registro físico é um número inteiro, a começar de zero, denominado endereço físico do registro. Para aumentar a eficiência do acesso ao disco, as estruturas foram desenvolvidas levando em conta que esse tamanho é múltiplo do tamanho do registro físico de acesso ao disco, tipicamente valores de potências de dois de 1 Kbyte (1K, 2K, 4K, etc.).

As informações a respeito da configuração duma base de dados SIRIUS ficam num registro físico pré-definido, numa estrutura denominada "Registro de Configuração da Base de Dados" - ReCoB. Esta estrutura, mostrada na tabela 5.1, contém informações a respeito da versão da base de dados, sobre a localização de outros registros de controle importantes da base, o tamanho definido para os registros físicos, o número de páginas a serem utilizadas pelo gerenciador de *buffers*, o número máximo de colônias possíveis de serem criadas nessa versão da base, o tamanho máximo do maior registro já criado na base de dados e outras informações.

Para que uma base de dados SIRIUS seja criada, devem-se definir os parâmetros de sua configuração. Isto é efetuado através de um utilitário que inicialmente oferece um conjunto de valores *default* para esses parâmetros e permite ao usuário redefiní-los de acordo com a aplicação e o ambiente de execução desejado. A definição desses valores sintonizam as estruturas de suporte do NuGO às diferentes aplicações, ao otimizar seu desempenho às várias características das aplicações.

A base é criada segundo os parâmetros de configuração, através de uma das primitivas do módulo de manipulação de base de dados descritas a seguir. Essa primitiva cria a estrutura inicial de uma base de dados SIRIUS vazia. A figura 5.1 mostra a estrutura inicial de uma base de dados SIRIUS composta de registros iniciais de controle utilizados para o funcionamento do NuGO.

ReCoB - Registro de Configuração da Base de Dados

Campos	Tipo	Número de bytes	Descrição
TipoReg	byte	1	Determina o tipo de registro
ContadorSalvaOid	upal4	4	Utilizado para o gerenciamento de OIds
Sentinela	upal4	4	Utilizado para se verificar a integridade do registro
Versao	byte*	5	Contém a versão da base
TamanhoBloco	upal4	2	Contém o tamanho dos registros utilizados pela base
TamanhoCache	upal2	2	Número máximo de páginas manipuladas pelo *cache*
RaaB1	upal4	4	Endereço do Registro de armazenamento do estado atual da Base
RaaB2	upal4	4	Endereço do Registro de armazenamento do estado atual da Base
TamBlob	upal4	4	Contém o tamanho em bytes da estrutura manipulada pela base para um BLOB
MaxCol	upal2	4	Número máximo de colônias na base de dados
MaxRegBase	upal4	4	Maior registro da base de dados
nrvpb	upal2	2	Número de registros virtuais por bloco
edpvb	byte	1	Espaço máximo desejável para os valores num bloco
edpcb	byte	1	Espaço máximo desejável para controle num bloco

Tabela 5.1: Apresentação das informações componentes do ReCoB

As primitivas componentes do módulo de manipulação de uma base de dados SIRIUS são as seguintes:

- **Criar uma base de dados** - Permite criar uma base de dados com as estruturas iniciais necessárias para o funcionamento e armazenamento dos dados efetuados pelo NuGO;
- **Ler uma base de dados** - Permite verificar o conteúdo armazenado nas estruturas de controle da base de dados criada;
- **Eliminar uma base de dados** - Permite eliminar uma base de dados SIRIUS.

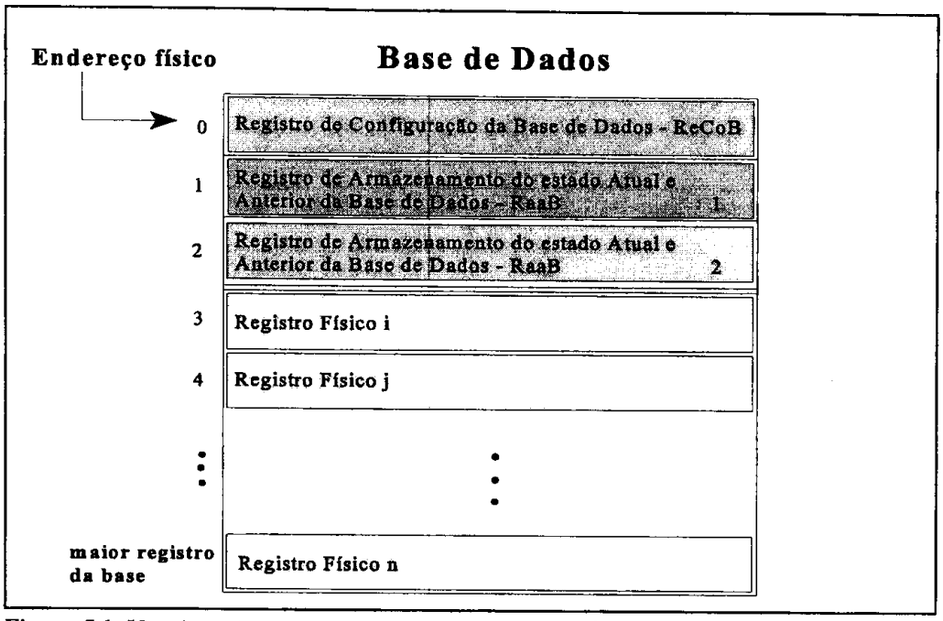


Figura 5.1: Uma base de dados SIRIUS com os registros de controle no início.

5.3 O Núcleo gerenciador de objetos

Pretende-se que o NuGO seja portável para diferentes máquinas ou plataformas. Para isso, uma característica necessária é a padronização dos tipos básicos usados na implementação. Essa tarefa é efetuada com a preservação do número de bytes e a ordem dos bytes (*big* ou *little endians*) utilizados para esses tipos independentemente do ambiente no qual se deseja executar o NuGO. Esse recurso é necessário, pois os vários compiladores ou máquinas podem utilizar-se de tamanhos diferenciados quanto ao número de bytes ou estrutura para um mesmo tipo básico. A tabela5.1 apresenta um conjunto dos tipos definidos para o NuGO e os respectivos tipos aos quais estão associados nos compiladores C++Builder, VisualC++ e G++/Sun. Esta normalização dos tipos básicos é efetuada através de um arquivo de configuração especial, denominado *máquina.def*. Como para esses três ambientes a ordem dos bytes é a mesma, adotou-se a representação *little endian* como padrão para uma base SIRIUS. Caso a máquina ou compilador, no qual o NuGO possa ser submetido não esteja definido neste arquivo, ele poderá ser acrescentado com a definição dos seus tipos básicos.

Tipos Básicos para o NuGO	C++ Builder Borland	Visual C++ Microsoft	G++ Sun
byte	unsigned char	unsigned char	unsigned char
cadeia	unsigned char *	unsigned char *	unsigned char *
pal2	short	short	short
upal2	unsigned short	unsigned short	unsigned short
pal4	long	long	*******
upal4	unsigned long	unsigned long	unsigned long

Tabela 5.2: Exemplos de tipos básicos nos diferentes compiladores

5.3.1 Subsistema de gerenciamento de atributos

A implementação do módulo SG-Atrib utiliza-se do conceito de sobrestante [Gaz_97], um conjunto de espaços na memória principal controlados pelo SG-Atrib, para cada uma das peças[1] lógicas utilizadas pelo NuGO como suporte à criação e navegação nas informações armazenadas na base de dados. Um sobrestante mantém na memória um espelho dos dados mantidos em disco, minimizando o acesso a disco e habilitando a implementação de objetos persistentes (armazenados em disco) e objetos transientes (não são armazenados em disco, existem apenas durante a execução duma aplicação). Objetos transientes e persistentes apresentam a mesma sintaxe e comportamento para a aplicação, a menos do fato de serem ou não persistentes.

Para o funcionamento do SG-Atrib, obtém-se inicialmente, da camada inferior, as informações de controle sobre a base de dados SIRIUS sobre a qual esse gerenciador está atuando, e que deve estar aberta. Se essa base de dados estiver inicializada, com instâncias de objetos armazenados decorrentes de execuções anteriores de aplicações, o SG-Atrib receberá, da camada SG-Log, as informações criadas e utilizadas pelo NuGO. Essas informações de controle encontram-se na colônia global, uma peça que contém informações onde se encontram, na base de dados, os registros lógicos iniciais que permitem localizar as demais informações armazenadas. Se a base de dados estiver sendo aberta pela primeira vez após a sua criação, então esse subsistema se encarregará de criar a colônia global.

Com as estruturas iniciais necessárias ao funcionamento desse subsistema é oferecido para as camadas superiores, através do uso das primitivas descritas a seguir, a possibilidade de particionar logicamente a base de dados, através das definição de colônias ou manipular objetos e seus valores.

---

[1]Peças: denominação dada a qualquer um dos elementos lógicos tratados pelo NuGO na camada SG-Atrib e nas superiores. Esses elementos são: colônia, objeto, lista e tupla.

5.3.1.1 Primitivas do módulo SG-Atrib-Composição

Esse módulo é composto pelas primitivas que permitem definir as partições lógicas na base de dados através de colônias. Uma colônia é composta de seus objetos e os atributos definidos para a partição lógica. A seguir são apresentadas as primitivas utilizadas para a manipulação de colônias e objetos.

- **Cria uma colônia**: Esta primitiva permite inserir uma colônia de código específico armazenada em disco no seu respectivo sobrestante ou apenas criar o sobrestante para uma colônia sem nenhuma referência em disco. Toda colônia é constrita por um objeto, portanto, para que esta tarefa seja executada, o objeto que a constringe deve existir e estar posicionado como um sobrestante para objetos;
- **Lê uma colônia**: O sobrestante é sincronizado com a colônia armazenada em disco, cujo código é o especificado no parâmetro;
- **Insere valores para a colônia**: Permite definir o tipo da colônia e o objeto que a constringe no sobrestante;
- **Obtém valores da colônia**: Retorna os valores solicitados para o sobrestante, como código da colônia global, código do objeto que a constringe e o código do primeiro objeto da sua lista de objetos;
- **Grava a colônia:** A colônia cujas informações encontram-se no sobrestante é armazenada na base de dados. O SG-Atrib mantém o histórico das atualizações efetuadas em sobrestantes (dados na memória). Se o estado atual da colônia no sobrestante não tiver sido alterado, a colônia não será efetivamente gravada em disco;
- **Cria um objeto**: O sobrestante do objeto é criado na memória sem referência a um objeto armazenado em disco, ou então cria-se o sobrestante com um objeto armazenado em disco, cujo código é especificado como parâmetro.;
- **Grava o objeto**: Se o objeto foi alterado, o conteúdo do sobrestante é gravado em disco;
- **Lê um objeto**: O sobrestante é sincronizado com o objeto armazenado em disco, cujo código é passado como parâmetro;
- **Obtém valores do objeto:** Obtém um conjunto de valores do objeto armazenado no sobrestante. Esses valores são o tipo do objeto, o código do objeto que constringe esse objeto e a colônia em que tal objeto habita;
- **Insere valores para o objeto:** Coloca valores no sobrestante do objeto, como o tipo do objeto, o código do objeto que o constringe e a colônia que tal objeto habita;

5.3.1.2 Primitivas do módulo SG-Atrib-Lista

Através das primitivas que são componentes desse módulo permite-se definir os atributos do tipo lista para os objetos numa base de dados. Uma atributo lista pode ser definido independentemente, ou estar associado a um objeto. As primitivas desse módulo são descritas a seguir.

- **Cria uma lista**: Um sobrestante é criado na memória sem referência a um objeto ou um atributo armazenado em disco. Se um código de objeto é passado como parâmetro, então o sobrestante da lista criada na memória estará associado ao referido objeto;
- **Grava a lista**: A lista contida no sobrestante é gravada em disco. O objeto do sobrestante associado a essa lista tem seu estado marcado como alterado e se torna persistente;
- **Insere uma lista**: Uma nova lista é inserida, após a atual armazenada no sobrestante. São passados como parâmetros as informações do tipo, o tamanho do valor da lista e o próprio valor;
- **Lê uma lista**: Obtém valores como tipo, tamanho do atributo, tipo de dado e o valor do atributo armazenado no sobrestante;
- **Avança na lista**: Avança um elemento da lista, o corrente será gravado em disco caso tenha sido alterado, o próximo é carregado para o sobrestante;
- **Volta na lista**: Caso exista, retrocede um elemento da lista. Se a lista corrente foi alterada, ela é gravada e o elemento anterior é lido;
- **Posiciona numa lista**: Obtém o elemento da lista, cujo código é passado como parâmetro. A lista corrente será gravada caso tenha sido alterada.

5.3.1.3 Primitivas do módulo SG-Atrib-Tupla

Esse módulo é composto pelas primitivas que permitem criar e manipular as tuplas numa base de dados. Uma tupla pode ser criada associada ou não a um objeto e os atributos que fazem parte de uma tupla, também estão armazenados na lista de atributos de um objeto. As primitivas que compõem esse módulo são apresentadas a seguir.

- **Cria uma tupla**: O sobrestante é criado na memória sem referência a um objeto ou a uma tupla em disco, como transiente. Se o código de um objeto for passado como parâmetro, o sobrestante criado estará associado a esse objeto e será considerado persistente ou transiente, segundo o estado do objeto associado;
- **Grava a tupla**: O sobrestante da tupla é gravado em disco. O sobrestante do objeto associado a essa tupla terá seu estado marcado como alterado;
- **Insere a quantidade da tupla**: Esse método permite especificar o número de atributos referenciados por essa tupla;
- **Insere uma lista na tupla**: Permite inserir no sobrestante da tupla a lista definida no seu respectivo sobrestante;

O módulo SG-Atrib representa uma implementação simplificada de uma base de dados SIRIUS, utilizada para a validação de estruturas internas, bem como um exemplo de como uma versão completa de uma camada semântica suportando o modelo SIRIUS completo deverá vir a ser desenvolvido, utilizando os módulos internos.

5.3.2 O subsistema de gerenciamento de registros lógicos

Nesse trabalho são apresentadas duas propostas de implementação para o subsistema de gerenciamento de registros lógico, a saber: SG-Log1 e SG-Log2. A diferença básica entre essas propostas consiste no tratamento específico dado aos registros lógicos de tamanho fixo efetuado pelo SG-Log2. A versão que implementa a primeira proposta (SG-Log1) trata todos os registros lógicos independentemente de serem de tamanho fixo ou variável. A versão que implementa a segunda proposta (SG-Log2) utiliza um módulo para o tratatamento dos registros lógicos de tamanho fixo denominado SG-Log21 e um outro módulo, SG-Log22, apenas para o tratamento dos registros lógicos de tamanho variável.

Em ambas as versões são utilizadas estruturas auxiliares que permitem efetuar o agrupamento das informações e o aproveitamento dos espaços livres nos blocos por colônias. Para cada colônia, tem-se um conjunto de listas de recuperação de blocos usados com espaços livres para o armazenamento de registros lógicos, sendo que cada uma dessas listas contém os blocos lógicos que armazenam um determinado tipo de registro lógico. Assim, para cada colônia existe uma lista de recuperação contendo os blocos lógicos com espaços livres para o armazenamento de objetos, outra lista de recuperação contendo blocos com espaços livres para armazenamento de tuplas e assim por diante, tendo-se uma lista de recuperação para cada um dos tipos lógicos suportados por essa camada. Além das listas de recuperação, para cada colônia também são mantidos o controle sobre o último bloco usado para cada tipo lógico, os blocos de início e fim da linha lógica de cada tipo lógico, isto é, a linha contendo todos os blocos que armazenam objetos de uma colônia, a linha contendo todos os blocos que armazenam as tuplas e todos os que armazenam as listas de uma colônia. As linhas de blocos lógicos são baseadas no conceito de arquivos lógicos implementados no GEO[Tra_91]. Essas estruturas são importantes nesta camada, pois suportam o esquema de *clustering* utilizado pelo NuGO e facilitam o acesso aos registros lógicos.

Além do controle interno efetuado para cada colônia, há o controle externo às colônias. Esse controle é composto por uma lista de recuperação dos blocos contendo espaços livres para o armazenamento de registros lógicos de colônias, o bloco inicial e o bloco final da linha de blocos lógicos contendo as colônias e o último bloco de colônias usado pela aplicação. A figura 5.2 mostra um bloco contendo todas as colônias da base de dados, contidas numa linha de colônias, os objetos de cada colônia e as tuplas e listas de cada objeto. A figura 5.3 mostra a estrutura de controle externo às colônias, na qual se armazenam os blocos contendo espaços livres para novas inserções, o último bloco usado e a linha de blocos do tipo colônia. A figura 5.4 exibe as estruturas controladas internamente a cada colônia da base de dados. Essas estruturas armazenam, para cada tipo de registro lógico (objeto, lista e tupla), informações referentes aos blocos contendo espaços livres, o último bloco usado, o início e o fim da linha de blocos contendo registros lógicos. Esses módulos são descritos a seguir nas subseções.

5.3.2.1 Proposta para uso de apenas um tipo de bloco lógico

Esse módulo define inicialmente o número máximo de registros lógicos que podem ser armazenados num bloco lógico, sendo que esse cálculo é efetuado para cada um dos tipos lógicos (colônia, objeto, tupla e lista). O cálculo é executado tomando-se como parâmetro inicial a porcentagem desejada de ocupação nos blocos para os valores (registros lógicos) e estrutura de controle (cabeçalho do bloco), valores esses obtidos do Registro de Configuração da Base de Dados - ReCoB.

As primitivas básicas que compõem o módulo SG-Log1 são as seguintes:

- **Obter endereço de registro lógico** - Essa primitiva efetua a leitura dum novo endereço de registro lógico de determinado tipo lógico (lista, tupla, objeto ou colônia). Verifica-se inicialmente se o último bloco lógico utilizado pela aplicação que contenha registros lógicos desse tipo possui algum livre. Caso isso não ocorra, verifica-se se a lista de recuperação de blocos lógicos contém algum bloco com registros lógicos disponíveis. Se a lista de recuperação estiver vazia, um novo bloco lógico será requisitado de modo que seja possível fornecer um registro lógico do tipo desejado. Um novo endereço de registro lógico é obtido num bloco lógico novo ou num bloco lógico em uso com registros lógicos livres, ou seja, ainda não usado. Para os blocos lógicos com registros lógicos do tipo objeto e requisitados pela primeira vez é efetuado um tratamento de suporte para o gerenciamento de Identificadores de objetos - OIds. Essa tarefa, representada na figura 5.5 situação I, é composta pelo repasse do valor em ContadorSalvaOId, um dos campos do bloco, para os registros

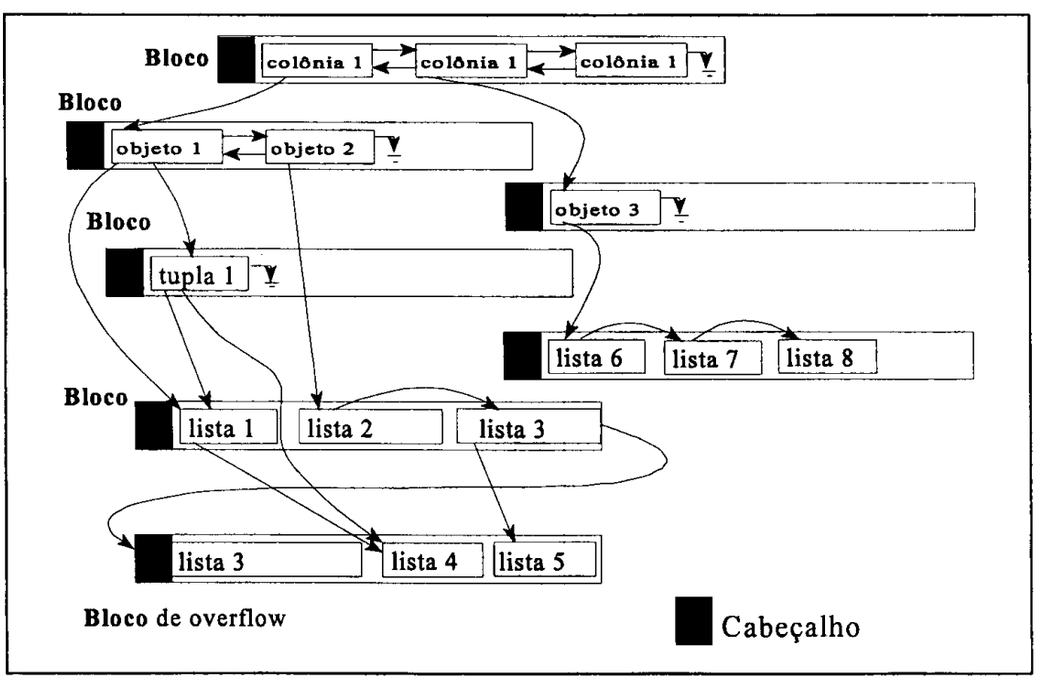


Figura 5.2: Blocos lógicos com os respectivos tipos lógicos agrupados por colônias e objetos.

lógicos de objetos ainda não utilizados, especificamente para o campo ContOid em cada um desses registros. No caso, o valor 5 é repassado para todos os registros lógicos, antes de ser utilizados;

- **Obter o registro lógico** - O conteúdo de um registro lógico é obtido com o fornecimento de seu tipo lógico e endereço lógico. Antes, porém, são requisitados à camada SG-Real os blocos lógicos com o registro lógico desejado. Para que isto seja efetuado, o endereço do bloco lógico de controle é calculado baseado no endereço do registro lógico procurado e transferido para a camada inferior ao SG-Log do NuGO. Um registro lógico pode ocupar mais que um bloco lógico, e nesses casos, a recuperação é efetuada através de várias requisições para a camada SG-Real, a fim de serem obtidos os vários blocos lógicos utilizados nesse armazenamento. O conteúdo do registro lógico é obtido e retornado por essa primitiva;
- **Escrever o registro lógico** - São passados como parâmetros para essa primitiva o endereço e o conteúdo do registro lógico a ser armazenado. O conteúdo dum registro lógico pode ser escrito em mais que um bloco lógico. O endereço do bloco lógico de controle, o qual deverá conter as informações desse registro lógico, é calculado a partir do endereço do registro lógico passado como parâmetro. A tarefa de alocar um registro lógico num ou mais blocos é composta de atividades como abrir espaço entre dois outros registros lógicos já armazenados, caso o espaço existente entre esses registros não seja suficiente para receber o conteúdo do registro lógico a ser armazenado;
- **Liberar o registro lógico** - Esse método permite colocar em disponibilidade um registro lógico de endereço conhecido, armazenado num ou mais blocos lógicos. Através do endereço do registro lógico determinam-se em quais blocos lógicos ele está armazenado em disco, e efetua-se a atualização dos campos de controle do bloco que o contém a fim de representar tal liberação. Se o registro lógico ocupar mais que um bloco lógico, a sua liberação poderá acarretar na disponibilização de blocos lógicos utilizados em sua alocação. No caso da liberação do último registro lógico do tipo objeto num bloco, este método percorrerá todos os registros lógicos para determinar o maior ContOid pertencente a todos os registros lógicos liberados desse bloco. O maior valor de ContOid ficará armazenado em ContadorSalvaOid, um campo do bloco lógico. A figura 5.5 mostra essa operação na situação II, na qual ocorre a eliminação do registro lógico cujo ContOid é 6. Todo registro lógico do tipo objeto tem seu

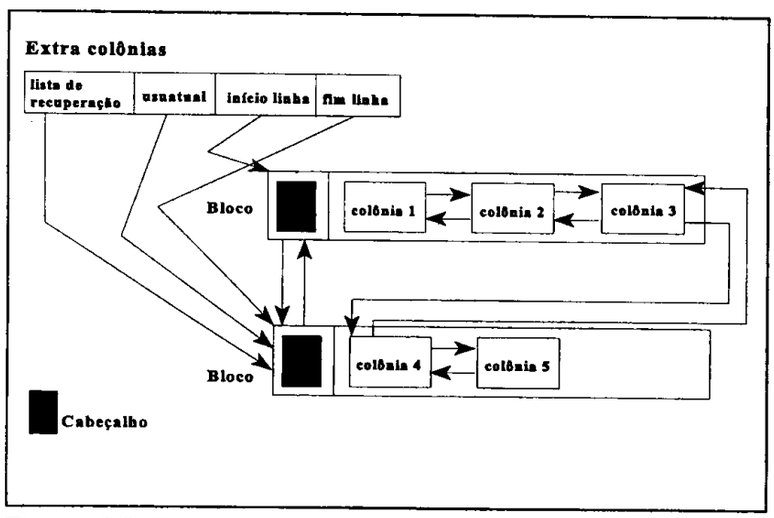


Figura 5.3: Estrutura de controle externo às colônias

ContOid incrementado numa unidade ao ser eliminado. Por fim, determina-se o maior dentre os ContOid dos registros lógicos deste bloco, no caso o de valor 8, que é colocado em ContadorSalvaOid, um campo do bloco.

5.3.2.2 Proposta com o uso de dois tipos de blocos lógicos

A camada SG-Log2 é composta pelos módulos SG-Log21 e SG-Log22. O módulo SG-Log21 efetua o gerenciamento dos registros lógicos de tamanho fixos (colônias e objetos) e o SG-Log22 efetua o gerenciamento dos registros lógicos de tamanhos variáveis (listas e tuplas). Nesse caso, a implementação usada para o SG-Log22 é a mesma para o SG-Log1, diferenciando-se apenas por se restringir ao tratamento apenas de dois de tipos registros lógicos, ou seja, listas e tuplas. Cada uma destas camadas são explicadas a seguir através da apresentação das primitivas que as compõem.

O módulo SG-Log21 efetua um conjunto de cálculos básicos para definir a formatação a ser utilizada para os blocos lógicos, registros virtuais e registros lógicos. A base para tais cálculos é um valor pré-definido para o número de registros virtuais por bloco, armazenado no ReCoB. Em seguida define-se o número de registros lógicos por registro virtual para cada tipo de registro lógico de tamanho fixo. Nessa versão são considerados registros lógicos de tamanho fixo apenas os registros de colônias e objetos. As primitivas desse módulo são descritas a seguir.

- **Lê novo endereço de registro lógico** - Essa primitiva obtém um endereço de registro lógico dado o seu tipo lógico, assim como a primitiva que lê um endereço de registro lógico do SG-Log1. A diferença é que o endereço de registro lógico é obtido num registro virtual, numa etapa adicional, em vez do acesso direto a um bloco lógico como é feito no SG-Log1. A busca por um registro lógico disponível é efetuada primeiramente no registro virtual indicado em *usuatual*, um campo para o tipo de registro lógico em *intracolônia*. Caso não tenha nenhum registro lógico disponível, a busca será efetuada nas células seguintes da linha corrente nesse registro virtual. Se, mesmo assim, nenhum registro lógico disponível for encontrado,

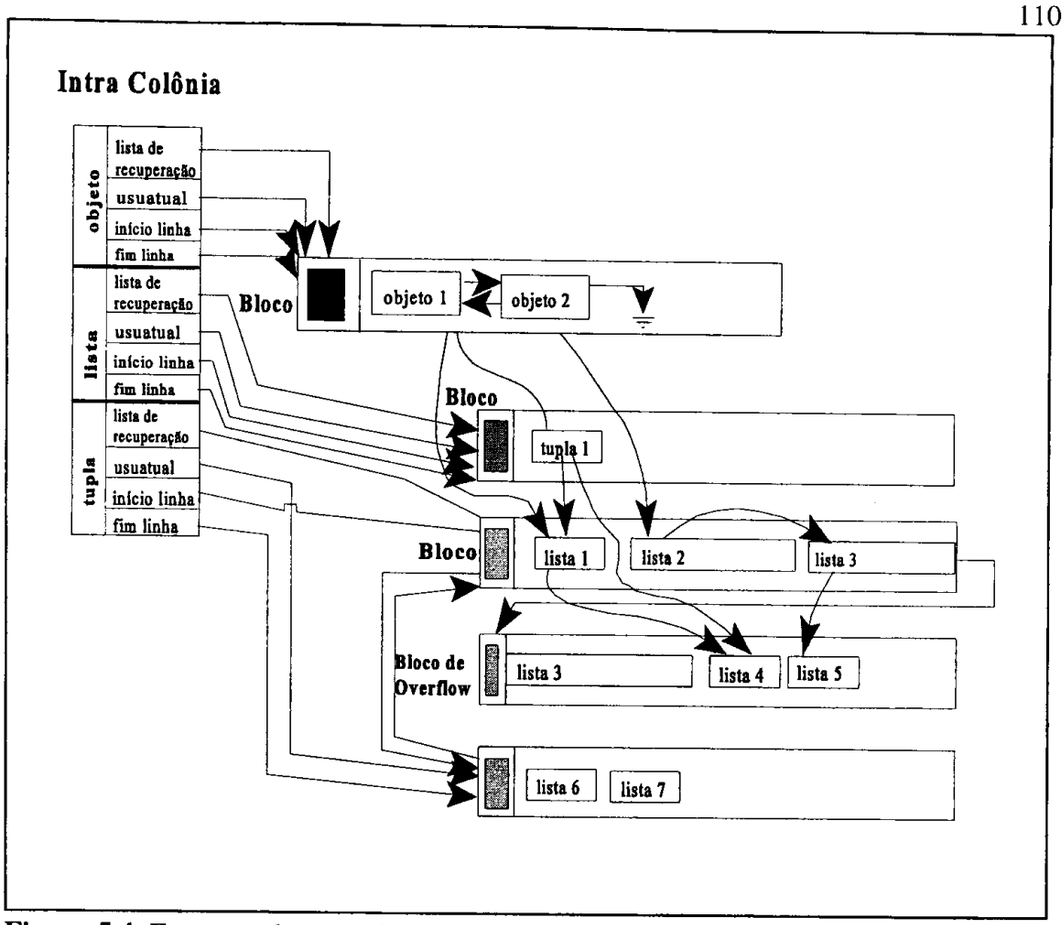


Figura 5.4: Estrutura de controle interno às colônias

a busca será executada nos blocos lógicos pertencentes à lista de recuperação global de blocos lógicos desse tipo, com registros virtuais livres. Um novo bloco lógico será alocado se não existir qualquer registro virtual livre. Caso esse novo bloco seja do tipo objeto, será efetuada a transferência do valor do campo ContadorSalvaOid para os respectivos campos ContOid dos registros lógicos do tipo objeto desse bloco, do mesmo modo como no SG-Log1, essa operação é parte do suporte na definição dos OIds para os objetos que posteriormente poderão estar armazenados neste bloco lógico;

- **Lê um registro lógico** - Obtém o conteúdo de um registro lógico através de seu endereço lógico e seu tipo. Através desse endereço determina-se o registro virtual e o bloco que contém esse registro lógico. É obtido do bloco lógico adequado o conteúdo do registro virtual que contém o registro lógico solicitado. Posteriormente, o conteúdo do registro lógico é recuperado nesse registro virtual e retornado por essa primitiva;
- **Escreve um registro lógico** - Essa primitiva localiza o registro virtual adequado e efetua a inserção do registro lógico. A operação de escrita é acompanhada da atualização das informações que indicam a presença do registro lógico no registro

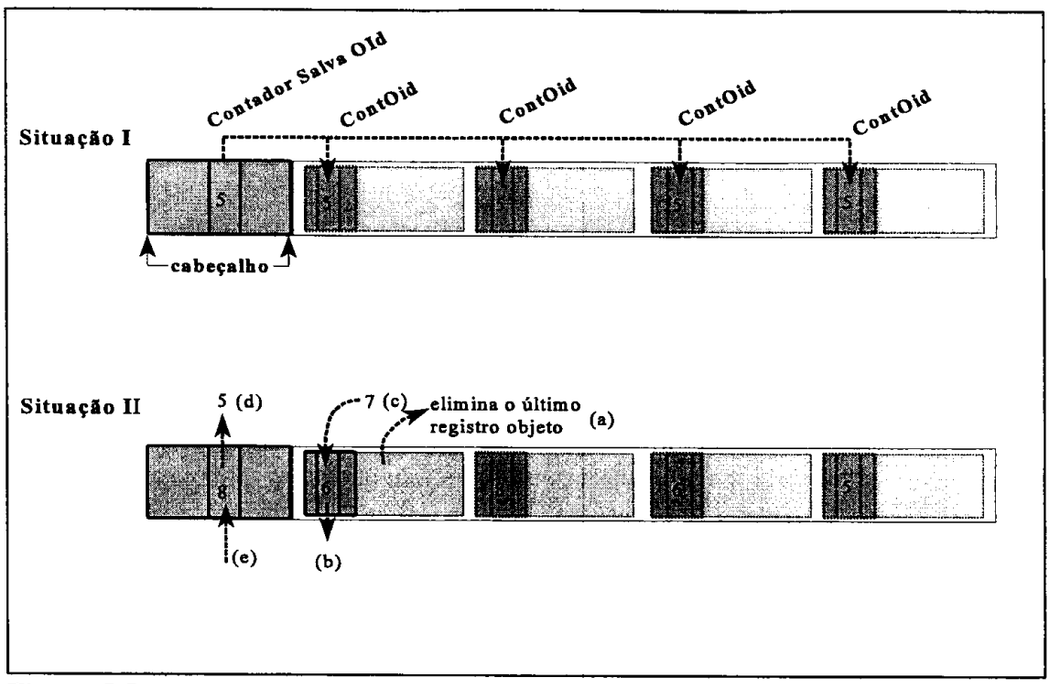


Figura 5.5: Atualização do valor de controle de Oid em registros lógicos do tipo objeto

virtual, como, por exemplo, atualização do *BitMap*[2] do registro virtual que indica o armazenamento do respectivo registro lógico;

- **Libera um registro lógico** - Permite que um registro lógico seja colocado como disponível. Através do endereço do registro lógico, localiza-se o registro virtual que o contém, no qual serão efetuadas as operações que lhe indiquem a liberação. A posição referente a esse registro lógico no *BitMap* do registro virtual deverá indicar a sua eliminação. Além disso, se o registro lógico for o último que estava sendo ocupado nesse registro virtual, o registro virtual também será colocado como disponível no bloco que o contém. Isto é efetuado através da atualização do *BitMap* do bloco, na correspondente posição associada ao registro virtual, de modo a representar o seu novo estado. Conseqüentemente, se o bloco agora possui um registro lógico disponível, deverá estar presente na lista de recuperação de seu tipo lógico. Se nenhum registro lógico estiver sendo ocupado no registro virtual, e essa situação representar que o bloco todo está vazio, então esse bloco será liberado. Se o bloco liberado possuía registros lógicos do tipo objeto, será efetuado o tratamento de suporte para o gerenciamento de OId. Esse tratamento consiste em percorrer os registros lógicos (vazios) existentes e obter o maior entre os valores para ContOId. O maior valor entre os ContOId é então passado para o campo do bloco ContadorSalvaOid. Essa operação libera o bloco para ser utilizado para outras estruturas de dados, não necessariamente objetos. No entanto, preservando-se o

[2]*BitMap*: uma estrutura interna de controle da existência ou não dos registros lógicos em um bloco. No SG-Log1, esta estrutura encontra-se no bloco lógico, no SG-Log2, encontra-se no registro virtual.

campo ContadorSalvaOId (existente em todos os blocos, independentemente de armazenar objetos ou não), caso futuramente esse mesmo bloco venha a ser utilizado novamente para armazenar objetos, então pode-se garantir que um mesmo ContOId não será reaproveitado para um objeto armazenado no mesmo endereço.

5.3.2.3 Testes

Um conjunto de testes foi feito de modo que se pudesse analisar o comportamento do NuGO perante as duas possibilidades de implementação para o subsistema de gerenciamento de registros lógicos propostas por esse trabalho. Os resultados mostrados a seguir foram obtidos pela execução do NuGO para quatro conjuntos diferentes de valores de entrada. Os valores introduzidos em cada uma das execuções a que foi submetido o NuGO estão descritos na tabela 5.3. Em todas as situações foram utilizados atributos com valores de tamanho 30 bytes e páginas de tamanho 1Kb.

Entradas	1a.	2a.	3a.	4a.
Número de colônias	1	2	4	8
Número de objetos por colônia	10	100	1000	10000
Número de tuplas por objeto	1	2	4	8
Número de atributos por tupla	1	2	4	8

Tabela 5.3: Valores de entrada para os testes aplicados no SG-Log1 e SG-Log2

Alguns dos gráficos obtidos para os valores de entrada especificados na trabela 5.3 são apresentados a seguir. Na figura 5.6 é apresentado o gráfico que descreve a quantidade de espaço não utilizado na base de dados em relação ao espaço total gasto. Na figura 5.7 mostra-se a quantidade gasta com as estruturas de controle nos blocos em relação ao total de espaço gasto na base de dados. Nos dois gráficos verificam-se os comportamentos para o

SG-Log1 e SG-Log2 referentes a cada uma das 4 execuções para os valores de entrada especificados na tabela 5.3.

Pode-se verificar que a estratégia de *clustering* adotada para o SG-Log1 ocupa mais espaço para as suas estruturas de controle em relação ao SG-Log2 em aplicações menores, no entanto, efetua um melhor espalhamento dos registros lógicos, pois ocupa mais os espaços alocados na base de dados.

As duas estratégias se equiparam para aplicações, cuja a quantidade de dados de entrada é considerável.

5.3.3 Subsistema de gerenciamento de registros reais

O subsistema de gerenciamento de registros reais trata os registros reais com o uso de várias estruturas de dados que fornecem suporte a algumas das funcionalidades do NuGO, como, por exemplo, o gerenciamento de transações, controle de concorrência, bloqueio, recuperação, gerenciamento de OIds e gerenciamento de *buffers*. Este subsistema também efetua a interface entre os dispositivos de armazenamento e a camada superior, SG-Log. Um registro físico - RegFis é requisitado através do gerenciamento de *buffers*, pelo SG-Real, e a partir daí recebe um índice real obtido por sua inserção em TCREFO - Tabela de Conversão de Registros Reais para Físico e de Overflow. Portanto, um registro real - RegReal é um RegFis com endereço numa das entradas da tabela TCREFO.

Um registro físico ou uma página de memória tem o tamanho definido durante a criação da base de dados SIRIUS, cujo valor deverá ser um múltiplo na potência dois de 1024 bytes, ou seja, 1Kb, 2Kb, 4Kb, etc. Uma vez definido o tamanho da página, o NuGO dimensionará o tamanho das estruturas de controle sobre esse valor, fato que inviabiliza qualquer alteração posterior desse tamanho.

Essa versão do NuGO utiliza 4 bytes para o armazenamento de endereços de registros físicos em disco, o que implica um máximo de 2^{32} (~4 bilhões de) endereços de registros físicos que poderão ser referenciados em disco. A tabela 5.4 mostra o tamanho

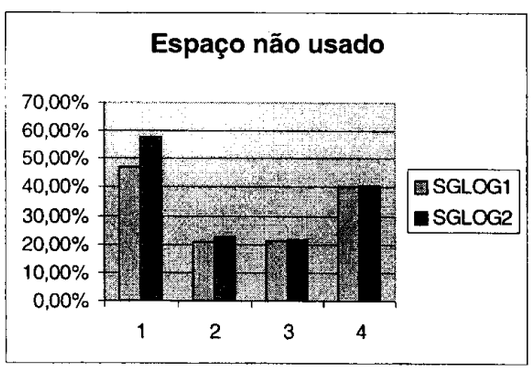


Figura 5.6: Gráfico que descreve a quantidade relativa de espaço não utilizado na base de dados

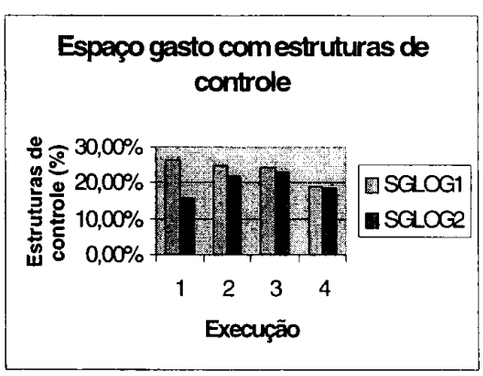


Figura 5.7: Gráfico que descreve a quantidade relativa de espaço usado para as estruturas de controle nos blocos

máximo em bytes, passíveis de serem endereçados, numa base de dados SIRIUS para cada tamanho de página adotado na sua criação.

Tamanho da página	Tamanho máximo da base de dados
1Kb	4Tb
2Kb	8Tb
4Kb	16Tb

Tabela 5.4: Proporção entre o tamanho de uma página e o tamanho máximo da base de dados em disco, baseado em endereços de 4 bytes.

A TCReFO permite implementar a estratégia de *shadowing* para o suporte ao gerenciamento de transações e recuperação de registros. Nessa versão do NuGO, essa estrutura é implementada numa matriz de 2 colunas com um número de linhas, cujo cálculo para a sua definição depende do tamanho adotado para a página. A tabela 5.5 mostra a relação entre o tamanho da estrutura TCReFO e o total de espaço endereçado em disco através dessa estrutura. No caso da utilização de páginas de tamanho de 4Kb, o espaço máximo endereçado pelo TCReFO seria de 4Gb (considerando-se uma base de dados totalmente ocupada). Embora seja uma capacidade significativa, ela ocupa espaço relativamente pequeno quanto ao total de bytes passíveis de serem armazenados em disco. No entanto, serve suficientemente para o caráter dessa implementação do NuGO, pois permite demonstrar o correto funcionamento do suporte para o qual foi definido. Por outro lado, essa estrutura pode ser facilmente estendida, como é proposto no capítulo 6, sem que comprometa as funcionalidades que a utilizam.

Tamanho das páginas	1Kb	2Kb	4Kb
Número máximo de FTIRRs referenciados no EFTIRR	242	498	1010
Número máximo de endereços reais contidos num FTIRR	254	510	1022
Máximo de end. Reais referenciados no TCReFO	61,468	253,98	1.032.220
Tamanho máximo da Base de Dados	62Mb	500Mb	4Gb
Tamanho do TCReFO em memória	484Kb	1,9Mb	4Mb
Proporção Tamanho do TCReFO x Q. e.r.d.[3]	0,79%	0,4%	0,1%

Tabela 5.5: Relação entre o tamanho de uma página, tamanho da estrutura TCReFO em memória e tamanho máximo em disco referenciado por essa estrutura.

A camada SG-Real faz interface com os módulos Base e Transação. O módulo Base contém o gerenciador de *buffers* (ou *cache*) e executa todos os acessos ao disco, seja diretamente, seja através do *cache*. O módulo Transação faz interface com o SG-Real e SG-Log e é definido sempre que se inicia uma aplicação. Esses dois módulos são apresentados com detalhes mais à frente no presente capítulo.

5.3.3.1 As primitivas do módulo SG-Real

O SG-Real obtém através do módulo Base, as informações armazenadas no registro de controle da base de dados - ReCoB, como, por exemplo, o tamanho em bytes dos registros físicos a serem utilizados durante todo o funcionamento dessa base de dados e a localização, em disco, dos registros de armazenamento do estado atual e anterior da base de dados - RaaBs. É efetuada a recuperação de ambos os registros RaaBs, onde, ao basear-se nos valores de estado de seus respectivos campos Ntriangulo, pode-se determinar em qual desses registros estão as informações mais recentes e consistentes para a recuperação em disco de todas as estruturas da base de dados e em particular para a operação desse subsistema. A maneira como essas estruturas são administradas durante o processo de construção e gerenciamento de transações, permitem que, a qualquer instante, a base de dados esteja consistente, permitindo também a recuperação da base de dados independentemente se houve ou não falha no sistema. Inicialmente são restabelecidas as estruturas de suporte a essa camada que são: a tabela de conversão de registros reais para

[3]Q.e.r.d.: Quantidade de espaço (em bytes) referenciado em disco.

físicos e de *overflow*; a tabela de contadores para o salvamento de OIds; a tabela de bloqueio; a lista de recuperação de registros físicos e a lista de recuperação de registros reais.

A lista de recuperação de registros físicos está armazenada no disco em duas versões, as quais deverão representar os seu conteúdo na memória no momento da submissão dos dois últimos *commits* aplicados na base de dados. Duas outras listas também são armazenadas, as contendo os endereços físicos que foram requisitados e liberados pelas duas últimas transações que sofreram *commit*.

O mesmo ocorre para a lista de recuperação de registros reais, o que exige também o armazenamento em disco das duas últimas versões desta lista obtidas no *commit*. Deverão ser mantidas também armazenadas em disco as listas de registros reais requisitados e liberados pelas transações que realizaram tais operações de *commit*.

No processo de recuperação, que ocorrerá sempre que esse módulo é iniciado, é efetuada uma análise comparativa envolvendo essas listas, a qual deverá acusar se houve ou não alguma falha no sistema. Se houve falha, a lista de recuperação de registros físicos (e reais) deverá ser reconstruída através do uso das demais listas. Se não ocorreu falha, a integridade da lista de registros físicos está garantida, portanto apta a ser utilizada por essa camada.

As primitivas básicas que compõem o SG-Real são descritas a seguir, apresentando-se suas funcionalidade e estruturas manipuladas.

- **Obtém um novo registro real** - Essa primitiva permite que um novo endereço de registro real seja obtido da lista de recuperação de registros reais. Essa lista de recuperação é mantida na estrutura TCReFO, com o uso de suas células para implementar o encadeamento dos elementos. Essa lista tem a particularidade de crescer em porções equivalentes ao tamanho de um registro físico, a medida que os endereços armazenados nessas porções vão sendo consumidos. Isso se reflete em economia de memória ao armazenar os endereços de registros reais ainda não usados. As operações são efetuadas de modo que o registro real possa tornar-se disponível no final de cada transação. Por exemplo, a cada endereço de registro real é associado um endereço de registro físico e efetuada a cópia do valor correspondente ao índice real em CSOID para o registro físico associado ao registro real e o bloqueio no modo leitura para o real liberado desta lista;
- **Libera um registro real** - Devolve para a lista de recuperação de registros reais um endereço de registro real. A inserção é efetuada numa das células da lista encadeada implementada na estrutura TCReFO. O registro físico associado ao registro real é devolvido para a lista de recuperação de registros físicos e o valor de ContadorSalvaOid no registro físico é copiado para o respectivo índice em CSOID;
- **Escreve um registro real** - Escreve o conteúdo de um registro real usando a estrutura de páginas *shadow* implementada no TCReFO. Para que essa operação seja efetivada, a transação corrente deve antes conseguir uma conceção de bloqueio para escrita neste registro real. Obtida a conceção, o conteúdo do registro real é copiado para um novo registro físico, cujo endereço é armazenado na célula de

overflow da estrutura TCReFO, gerando uma duplicação de páginas, na qual essa última página criada receberá as alterações executadas pela transação;

- **Obtém uma página de registro real** - Obtém o conteúdo de um registro real. Antes de habilitar a leitura, verifica se esse registro real possui bloqueio compatível com essa operação de leitura. Caso seja permitida, a operação será executada. Isso é feito ao obter-se a página adequada, a qual pode ser a página *shadow* ou a página original. A escolha em qual das páginas ocorrerá a leitura dependerá se a transação tenha ou não efetuado modificações neste registro real;
- **Atualiza um FTIRR** - Essa primitiva efetua a atualização numa porção do TCReFO denominada FTIRR que contém o registro real passado como parâmetro. Uma nova página é criada para receber a versão atualizada deste FTIRR. Essa página é armazenada em outro espaço do disco e seu endereço colocado na respectiva posição no EFTIRR, pertencente ao RaaB e que conterá o novo estado da base de dados. O registro físico que contém a versão antiga desse FTIRR será tornado disponível na lista de recuperação de registros físicos;
- **Cancela uma operação de escrita num registro real** - Um registro real que sofreu uma operação de escrita deverá retornar ao estado anterior à modificação. Essa operação é conseqüência de uma operação de *abort* efetuada pela transação que retém o bloqueio sobre o registro real passado como parâmetro. Essa primitiva efetua a liberação da página com a atualização e colocando o endereço de registro físico original em ambas as colunas do TCReFO. Essa nova composição de endereços nas colunas do TCReFo indica que o registro real contém apenas uma versão em disco e que o registro está liberado para novas operações de escrita ou leitura;
- **Encerra uma operação de escrita num registro real** - O registro real que sofreu uma operação de escrita deverá concretizar as modificações na base de dados, numa operação de *commit* executada pela transação que retém o bloqueio nesse registro. Essa primitiva efetua tal tarefa ao colocar a página contendo as modificações como a versão oficial do registro real. O endereço de registro físico com a versão original do real é liberado para a lista de recuperação de registros físicos, o endereço na segunda coluna do TCReFO é copiado para a primeira coluna, tornando-se a única versão em disco para o real. Assim como a primitiva que cancela um real, a nova composição de endereços físicos no TCReFO indica que o real estará liberado para novas operações de leitura ou escrita;
- **Finaliza parcialmente uma operação de escrita num registro real** - As operações dessa primitiva são equivalentes às executadas pela anterior. A diferença consiste, nessa primitiva, em não liberar o bloqueio no registro real passado como parâmetro. Portanto, o endereço físico da segunda coluna do TCReFO é copiado para a correspondente posição na primeira coluna. Antes, porém, o endereço do registro físico com a versão original do real é tornado disponível na lista de recuperação de registros físicos. Desse modo, a única versão armazenada em disco será a versão modificada pela transação para o real. O bloqueio permanece o mesmo, portanto mantido ainda sob o controle da transação que o modificou;
- **Grava a lista de recuperação de registros físicos** - Efetua a gravação em disco da lista de recuperação de registros físicos. Essa primitiva é acionada sempre que é executada uma operação de *commit*, refletindo na base os endereços físicos disponíveis até o momento dessa operação. Para o armazenamento dessa lista de

endereços, são utilizados registros físicos cujos endereços estão na própria lista de recuperação. Assim, um cálculo prévio deve ser efetuado para verificar quantos registros serão requisitados junto à lista de recuperação para esse armazenamento. O endereço do local no disco onde está gravada a lista de recuperação é armazenado no RaaB que conterá o novo estado da base de dados;

- **Grava a lista de recuperação de registros reais** - Essa primitiva executa a gravação em disco da lista de recuperação de registros reais. Como a implementação dessa lista ocorre na estrutura do TCReFO, a gravação só será efetuada para os trechos ainda não armazenados pela execução da primitiva que atualiza um FTIRR;
- **Solicita bloqueio** - Efetua uma solicitação de bloqueio para um registro real. A solicitação será atendida se o tipo de bloqueio requisitado for compatível com o tipo atual de bloqueio do registro;
- **Libera bloqueio** - Executa a liberação do bloqueio efetuado pela transação corrente para um registro real.

5.3.3.2 O módulo transação

A criação desse módulo, em tempo de execução, é efetuada ao iniciar cada aplicação e poderá se encerrar através de uma operação de *abort* ou *commit*. Através da operação de *checkpoint* é possível efetivar na base de dados as alterações executadas pela transação permitindo que essa continue aberta, tal recurso é importante no suporte a transações longas.

Várias transações podem ser criadas em um mesmo período e executadas concorrentemente. As estruturas definidas no SG-Real possibilitam que isso ocorra garantindo-se as propriedades ACID.

Em um módulo transação é criado um conjunto de listas utilizadas no suporte das suas operações. As listas de cada transação são:

- lista para o armazenamento de todos os registros reais lidos pela transação;
- lista para o armazenamento de todos os registros reais escritos pela transação;
- lista para o armazenamento de todos os registros reais requisitados, ou seja, criados pela transação;
- lista para o armazenamento de todos os registros reais liberados, ou eliminados pela transação;

Além das listas acima, uma transação também controla os registros físicos liberados e requisitados. Tal controle é implementado através da manutenção de mais duas outras listas: uma lista de registros físicos requisitados pela transação e outra de registros físicos liberados.

As primitivas básicas do módulo transação são:

- **Encerra** - Essa primitiva implementa a operação de *commit* para uma transação. Executa a atualização das estruturas de suporte ao gerenciamento de transação e recuperação proporcionados pela camada SG-Real, a saber: TCReFO, FTIRR, lista

de recuperação de registros físicos, lista de recuperação de registros reais, bloqueio e RaaB. A execução dessas operações é efetuada para cada um dos registros reais pertencentes às listas internas (listas de registros lidos, escritos, requisitados e liberados) da transação. Assim, todos os registros reais lidos e escritos pela transação têm seus bloqueios liberados, as páginas *shadow* com as alterações executadas pelas operações de escrita devem ser as únicas versões dos registros reais gravadas em disco, as listas de recuperação de registros físicos e registros reais são gravadas em disco, efetua-se a atualização do RaaB com o novo estado da base de dados e finaliza-se a transação;

- **Cancela** - Essa primitva executa a operação de *abort* para uma transação. É efetuada a atualização do TCReFO para cada um dos registros reais escritos pela transação, eliminando-se as páginas *shadow*; liberam-se todos os bloqueios requisitados, sejam de escrita ou leitura. Em seguida a transação é finalizada;
- **Encerra parcialmente** - Idêntica à primitiva que implementa a operação de *commit*, mas não efetua a liberação dos bloqueios efetuados pela transação e mantém a transação aberta.

5.3.3.3 O módulo base

Esse módulo é controlado pelo SG-Real e tem como funções: abrir a base, cuja identificação é transferida como parâmetro; obter o registro de controle da base - ReCoB e interagir com o módulo gerenciador de *buffers* (ou *cache*). As informações obtidas no ReCoB são transferidas por esse módulo para o SG-Real. As primitivas básicas componentes do módulo base são as seguintes:

- **Obtém novos registros** - Obtém um conjunto de novos endereços de registros físicos, provocando o crescimento físico da base de dados em quantidades equivalentes ao incremento definido no ReCoB. A gravação do endereço do último registro obtido na base de dados no ReCoB é também executada por essa primitiva. As páginas referentes aos novos registros são colocadas no *cache*;
- **Lê** - Obtém do *cache* a página referente ao endereço de registro passado como parâmetro;
- **Lê registro de controle** - Obtém um registro de controle diretamente da base de dados sem colocá-lo no *cache*. Esse registro é utilizado para o gerenciamento global da base, podendo ser o registro ReCoB ou um registro RaaB;
- **Escreve** - Escreve a página corrente no *cache*. Essa operação não grava diretamente no disco a página corrente, apenas marca-a como escrita e no momento em que ela deva ser substituída no *cache* é que será efetivamente armazenada;
- **Escreve registro de controle** - Escreve o registro de controle especificado diretamente no disco, sem colocá-lo no *cache*;
- **Descarrega o *cache*** - Esse método executa a gravação em disco de todos os registro físicos marcados como escritos pela transação corrente.

5.4 Conclusões sobre a implementação

5.4.1 Armazenagem eficiente e flexível para atributos

O NuGO utiliza-se de dois tipos de estruturas de dados para o armazenamento dos valores de atributos dos objetos. Essas estruturas são as listas e tuplas. Assim como no capítulo 4, é usada a notação de i-lista e i-tupla para diferenciar tais estruturas utilizadas na implementação das listas e tuplas semânticas das aplicações.

As implementações encontradas na literatura[Ber_94] [Cat_94] para os atributos dos objetos são executadas na forma de listas (ou com característica de listas), compondo uma seqüência de campos onde se tem um campo *idatrib*, uma espécie de identificador que indica em qual objeto esta associado o atributo, o *tamanho* armazena a quantidade em bytes do valor do atributo e o *valor* que contém o valor do atributo. Essa forma de armazenamento de atributos é mostrada na figura 5.8 representação (a). Os detalhes de como essa composição dos atributos são implementados pelo sistema são diferenciados, no entanto a idéia de lista é a mesma. Por exemplo, o ORION armazena o *idatrib* no começo do registro e o *valor* no final desse mesmo registro, como está representado na figura 3.3. A herança de atributos é também implementada utilizando-se a característica de lista para atributos, como pode ser observado na figura 5.8 representação (b), na qual inicialmente armazena-se os atributos do tipo concatenados com os atributos do subtipo e assim por diante, dependendo da composição hierárquica de tipos a que pertence o objeto.

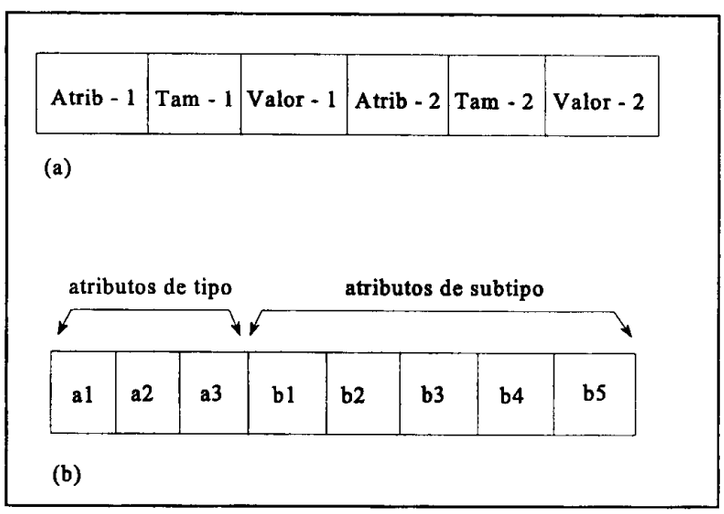


Figura 5.8: Uma sequência de campos na forma utilizada pelos atributos com característica de listas

A proposta apresentada nesse trabalho para o armazenamento de atributos dos objetos também inclui o suporte a listas. Essa forma é a mais flexível, pois um mesmo atributo pode ter vários tamanhos em diferentes objetos, os atributos podem ser armazenados em diferentes lugares físicos e todos os objetos não necessitam ter o mesmo conjunto de atributos[Cat_94]. No entanto, diferencia-se pelo modo em que as i-listas são implementadas e pela utilização de i-tuplas para o suporte ao armazenamento dos atributos que compõem uma instância de abstração de especialização/generalização ou instanciação/classificação.

Através das i-listas é possível especificar as informações pertinentes a cada atributo agregado ao objeto, compondo-se em cada i-lista de informações referentes ao tipo do atributo, tipo de dado armazenado e valor do atributo. Nessa estrutura, os atributos de um objeto são organizados na forma de uma lista duplamente encadeada e armazenada separadamente dos objetos, cujo objetivo é facilitar a manipulação e a navegação sobre esses itens de dados.

Ao utilizar-se da estrutura de i-listas pode-se facilmente conceber os atributos do tipo lista para os objetos, nos quais é importante a ordem entre os seus valores, além da unicidade, compondo-se como conjunto de valores, assim como na teoria de conjuntos. A restrição de unicidade pode ser relaxada, caso o usuário deseje tal facilidade.

A representação através dos registros i-lista, permite incorporar as ocorrências de atributos multivalorados, nos quais basta inserir na lista de atributos do objeto novos registros lógicos lista de mesmo tipo de atributo e com os seus respectivos valores. Deste modo, para a recuperação de qualquer dos valores para qualquer tipo de atributo, basta obter o conteúdo de todos os nós desta lista encadeada, cujo tipo de atributo é o mesmo.

Embora os sistemas orientados a objetos também utilizem estruturas de lista, o NuGO adiciona a elas as estruturas de i-tuplas, utilizando tecnologia originada na construção de gerenciadores relacionais. Para isso, alguns atributos predeterminados dos objetos também são agrupados numa estrutura de i-tupla. A estrutura de dados de suporte para as i-tuplas, assim como para as i-listas, também é uma lista duplamente encadeada que permite definir uma lista de i-tuplas para um único objeto. As i-listas armazenam todos os atributos de um objeto, mesmo aqueles contidos nas i-tuplas. Portanto, podem existir atributos nas i-listas que não pertencem a nenhuma i-tupla.

Durante o processo de criação de um objeto, os atributos básicos (ou obrigatórios) especificados através do esquema que define a sua classe, são também agrupados através de uma i-tupla, embora cada um individualmente também permaneça ligado à i-lista de atributos do objeto. Se esse objeto é subtipo de algum supertipo, então a primeira i-tupla será a contendo os repectivos atributos do supertipo. Este processo se repete para todos os supertipos existentes na hierarquia a que esse objeto possa pertencer, formando-se a lista das primeiras i-tuplas nesse objeto.

A estrutura de i-tuplas utiliza a tecnologia de concatenação de atributos do modelo relacional e, dessa maneira pode aproveitar as técnicas eficientes de recuperação de dados desse modelo. A idéia é basicamente encontrar uma maneira extremamente rápida de recuperar os dados mais comumente acessados de cada objeto utilizando as estruturas de i-tuplas, ao mesmo tempo em que as estruturas de i-listas permitem toda a flexibilidade necessária aos conceitos de orientação a objetos, tais como atributos extras, multivalorados etc.

5.4.2 Gerenciamento de OIds

O sistema de gerenciamento de OIds implementado pelo presente trabalho apresenta como característica inovadora a capacidade de localizar diretamente um objeto no disco e, ao mesmo tempo, permite a reutilização do espaço utilizado para o armazenamento desses objetos, sem que para isso tenha de efetuar qualquer sobrecarga operacional no sistema ao oferecer tais recursos.

O código de identificação de um objeto é formado pela concatenação dos valores ContOid e o endereço do registro lógico que armazena o objeto no disco. O valor ContOid é obtido num dos campos do registro lógico de tipo objeto e o endereço lógico é fornecido com base na posição onde se encontra o registro no bloco lógico e em qual bloco acha-se armazenado. Ambos os valores, ContOid e endereço lógico são basicamente definidos pela camada SG-Log e, portanto, definem também os identificadores de objetos.

A definição do OId para um objeto é determinada através das operações descritas a seguir:

- se o bloco corrente possuir um registro lógico do tipo objeto disponível, então esse será requisitado e o seu valor em ContOid vai ser utilizado, junto com a sua localização lógica, para compor seu OId;
- se não houver bloco com registros lógicos do tipo objeto disponível, então um novo bloco lógico é requisitado. A camada SG-Real aloca uma nova página para compor o registro real que receberá o respectivo valor em CSOID para o seu campo ContadorSalvaOid. Na camada SG-Log, o valor em ContadorSalvaOid é repassado para os campos ContOid dos futuros registros lógicos contidos nesse bloco lógico. Executado esse repasse de valor, o novo registro do tipo objeto é liberado e seu valor em ContOid é utilizado para compor o OId.

A eliminação de um objeto influi na composição do OId dos próximos objetos que reutilizarão o mesmo espaço para o armazenamento. As etapas que compõem a eliminação dum objeto são descritas a seguir, ao enfocar-se os aspectos de gerenciamento de Oids:

- todo objeto eliminado tem seu ContOid acrescido numa unidade. Se esse objeto eliminado for o último registro lógico do tipo objeto do bloco, após o incremento do seu ContOid, será determinado o maior entre todos os valores para o ContOid desse bloco e repassado para ContadorSalvaOid um campo do bloco;
- o bloco liberado, como conseqüência da eliminação do último objeto contido nesse bloco, terá sua eliminação executada na camada SG-Real, liberando-se o registro físico associado ao respectivo índice real e o valor em ContadorSalvaOid repassado para a correspondente posição da estrutura CSOID.

A localização de um objeto, com o conhecimento do seu Oid, é efetuada nas seguintes etapas:

- obter do OId o valor que determina o endereço do registro lógico do tipo objeto;
- efetuar a operação aritmética que determina o bloco lógico que contém o registro lógico do tipo objeto e sua posição nesse bloco lógico;
- obter o bloco lógico através de mensagem à camada SG-Real;
- a camada SG-Real localiza no TCReFO a página que contém o endereço real correspondente ao bloco lógico desejado;
- obtém-se do *cache* a página com o item de dado objeto e repassa-a para as camadas superiores do NuGO.

Portanto, a recuperação de um objeto é imediata, pois através do seu OId determina-se o local onde está armazenado em disco. O processo de criação e eliminação de um objeto permite garantir a unicidade do OId durante todo o ciclo de vida do objeto e a capacidade de reutilizar o mesmo espaço de armazenamento por outros objetos sem perder tal unicidade na identificação.

O sistema de gerenciamento de registros físicos e o sistema de *cache* das camadas internas dão suporte à realocação de registros e a manutenção dos valores no ContOId, habilitando ao mesmo tempo acesso imediato ao endereço real do objeto a partir de seu OId, sem depender de qualquer estrutura de indexação. No entanto, ao mesmo tempo que impede que um OId seja reaproveitado, habilita que o endereço físico do objeto em disco possa mudar e habilitando-o a recompactação da base.

5.4.3 Esquema de *Clustering*

O esquema de *clustering*, proposto pelo NuGO, utiliza-se de características semânticas para definir a distribuição dos objetos em regiões físicas da base de dados. Através da abstração de composição, os objetos com seus atributos são alocados em partições físicas na base de dados, cuja proximidade física beneficia amplamente sua recuperação e armazenamento.

Um suporte é oferecido para que a implementação desse esquema de *clustering* seja efetuado de modo eficiente, não acarretando sobrecarga operacional na execução dessa tarefa. Para cada partição semântica formada pelas aplicações e constituída pelas instâncias na base de dados, existe um conjunto de estruturas de apoio que permitem a distribuição adequada das peças lógicas que as compõem. Essas estruturas são as seguintes:

- endereço do último bloco usado - é armazenado o último bloco lógico utilizado pela transação, isto para cada tipo lógico e em cada partição. Com isso, permite-se dar prioridade à alocação de novos registros lógicos nos blocos que têm maior possibilidade de serem encontrados no *cache*, otimizando a tarefa de armazenamento;

- uma lista de recuperação para cada tipo lógico - essas listas de recuperação são compostas, cada uma delas, pelos blocos lógicos que contêm espaços para o armazenamento de registros lógicos de determinado tipo lógico. Levando em conta que, existe um conjunto de listas de recuperação para cada partição na base de dados e uma lista de recuperação para apoio na confecção das próprias partições. Com isto, em vez de alocar novos blocos lógicos, efetua-se sempre a alocação dos registros lógicos nos blocos requisitados, com espaço disponível e maior probabilidade de serem encontrados no *cache*, efetuando-se a distribuição de modo mais rápido, com maior aproveitamento dos espaços disponíveis e próximos dos demais registros lógicos semanticamente relacionados;
- arquivos lógicos e linhas de blocos lógicos do mesmo tipo - agrupa os atributos, tanto em estruturas de i-listas quanto de i-tuplas, permitindo ao mesmo tempo o acesso eficiente aos atributos mais freqüentemente acessados e acesso flexível para suportar qualquer estrutura de dados multivalorada e/ou dados semi-estruturados.

5.4.4 Armazenagem de atributos de tamanho variáveis nas páginas

Um registro lógico, seja do tipo lista ou tupla, denominados de tamanho variável, é armazenado em dois tipos de blocos lógicos: bloco lógico de controle ou bloco de *overflow*. Esses blocos são páginas que recebem formatação adequada para receber os registros lógicos. Utilizando-se dessa estrutura, um registro lógico poderá ser composto de grande quantidade de bytes ou crescer com liberdade, pois a operação de gravação, através dessa estrutura é facilitada pela composições desses blocos e da atuação de um algoritmo que determina a ação a ser tomada. No sistema O2, se um registro aumenta de tamanho e o local onde está armazenado não permite tal crescimento, é efetuado um deslocamento para um outro bloco, atualizando-se os indicadores da nova localização. O que implica em aumentar mais um acesso às páginas. Esse processo é mostrado na figura 3.10. No caso do NuGO, em nenhuma das situações haverá a necessidade de deslocar para novo local de armazenamento, até mesmo se o espaço onde o registro lógico alvo estiver alocado for pouco para receber o novo tamanho. A estrutura de blocos lógicos utilizada pelo NuGO permite que novos blocos sejam inseridos entre os já existentes, permitindo qualquer tipo de crescimento. Essa estrutura é ideal para o tratamento de dados multimídia, nos quais o tamanho desses tipos de dados é considerável exigindo-se uma estrutura bastante flexível para recebê-los.

5.4.5 Uso de páginas *shadow* no suporte à recuperação e gerenciamento de transação.

Algumas desvantagens na utilização da estratégia de páginas *shadow* são apontadas [Elm_00][Sil_99], dentre as quais são ressaltadas as seguintes: a necessidade em ambientes multi-usuários da utilização complementar de arquivos de *logs* e *checkpoints*[4]; a constante alteração das páginas das respectivas posições no disco, dificultando as estratégias de *clustering* físico para os itens de dados relacionados; a sobrecarga no sistema para se manter atualizados no disco os diretórios com as páginas modificadas em cada operação de *commit*; a dificuldade em manter-se a coleta de lixo quando uma transação sofre *commit*; e a dificuldade em tornar atômica a operação de migração entre o diretório corrente e o diretótio de páginas *shadow*. A proposta de implementação de páginas *shadow* através do TCReFO, no NuGO, combinado com outras estruturas, como o Bloqueio e RaaBs, permitem amenizar os problemas citados quanto ao uso dessa estratégia, além de apresentar melhorias significativas em alguns casos.

Nesse trabalho, o suporte dado pelo NuGO à recuperação, gerenciamento de transação e controle de concorrência deve-se à estrutura TCReFO, onde são efetuados os controles das páginas originais e das páginas contendo as atualizações executadas pelas transações. Apoiado nesta estrutura também é executada a validação quanto aos tipos de bloqueio efetuados pelas transações. Se o bloqueio é de escrita, os endereços dos registros físicos (ou páginas) são diferentes em ambas as colunas do TCReFO e, se os endereços, são iguais, o bloqueio efetivado, caso exista, é destinado a operações de leitura. Portanto, não há a necessidade de uma estrutura auxiliar para armazenar o tipo de bloqueio efetuado na respectiva página pela transação como é utilizado pelo sistema ORION. Em conjunto com o TCReFO, são armazenados na estrutura Bloqueio a lista de transações que requisitaram bloqueios nos respectivos índices reais. Apenas uma transação por vez poderá efetivar um bloqueio de escrita num registro real e, somente após a liberação desse bloqueio pela correspondente transação, é que este registro real estará disponível para outras transações. Um bloqueio de leitura pode ser compartilhado por mais de uma transação. Para isso, utiliza-se uma lista ligada com as transações que requisitaram esse tipo de bloqueio no respectivo registro real. Esse esquema garante o controle de concorrência sobre as páginas utilizadas pelo TCReFO.

As requisições das páginas referenciadas pela estrutura TCReFO são executadas via gerenciador de *buffers*, fato que reduz o número de acessos ao disco durante a utilização da estrutura. A requisição duma nova página para receber as alterações é efetuada via lista de recuperação de registros físicos, o que também implica numa redução aos acessos ao disco, pois os endereços de registros disponíveis nessa lista incluem páginas liberadas pelas transações ou páginas novas e, portanto com grandes possibilidades de serem encontradas no *cache*. Esse fato contorna um dos problemas provocados pela utilização da estratégia de

---

[4]*Checkpoint*: são pontos de controle armazenados como entradas no arquivo de *log* que indicam a ocorrência de escrita no disco de todas as páginas alteradas do gerenciador de *buffrs* naquele momento[Elm_00].

páginas *shadow*, ou seja, um número excessivo de acessos ao disco para a manipulação das páginas originais e de atualizações.

O tamanho do TCReFO, nesta versão, é limitado pelo tamanho das páginas definidos na criação duma base de dados SIRIUS, que conseqüentemente define o número de FTIRR possíveis de ser representados no TCReFO. Assim, como já foi descrito no capítulo 4, esse tamanho pode evoluir de 484 Kb, 1,9 Mb, 4 Mb e assim por diante, dependendo do tamanho adotado para as páginas. No entanto, o aumento do tamanho da estrutura do TCReFO na memória implica numa proporcional redução na relação com sua capacidade na quantidade de informações em disco, capaz de referenciar, como é apresentado na tabela5.6. Por exemplo, para páginas de tamanho 4Kb, o tamanho do TCReFO representa apenas 0,1% do total de informações que é capaz de referenciar em disco.

Tamanho das páginas	**1Kb**	**2Kb**	**4Kb**
Tamanho do TCReFO em memória	484Kb	1,9Mb	4Mb
TCReFO x Q.e.r.d.	0,79%	0,4%	0,1%

Tabela 5.6: Relação entre o tamanho de uma página, o tamanho do TCReFO e a capacidade do TCReFO em referenciar informações em disco baseado no seu tamanho na memória.

Um outro ônus da adoção dessa estratégia é a necessidade em efetuar a gravação da estrutura TCReFO a cada operação *commit*. No NuGO, isso ocorre, mas a gravação em disco é executada apenas dos trechos do TCReFO que sofreram alteração pela transação encerrada.

O SG-Real utiliza-se de uma lista de recuperação de registros físicos "a quente", na qual se efetua o gerenciamento de todos os registros liberados ou requisitados pelas transações em execução. A cada operação *commit*, essa lista de recuperação é armazenada em disco, junto com a lista de registros físicos liberados e requisitados. Através da análise sobre as listas contendo endereços físicos obtém-se o conjunto dos endereços que devem constar nessa lista independentemente da ocorrência ou não de falhas. A estrutura TCReFO utiliza-se dessa lista de recuperação no suporte às operações de requisição de novas páginas para conter as modificações efetuadas por operações de escrita e também como receptora das páginas liberadas, tanto em operações *abort*, quanto em operações de *commit*.A utilização da lista de recuperação em tempo de execução, permite evitar a utilização de operações de coleta de lixo, após cada operação de *commit*, como é feito no O2.

O SG-Real proporciona de modo confiável a atomicidade na operação de atualização do diretório de páginas *shadow* para o diretório corrente. Essa tarefa é efetuada sempre numa operação *commit*, na qual apenas os segmentos de FTIRRs com endereços de registros reais atualizados pela transação é que terão sua representação efetivada em disco.

Todos os FTIRRs contendo as atualizações do TCReFO são gravadas em disco e seus respectivos endereços são armazenados no EFTIRR do RaaB denominado futuro, pois representará o futuro estado da base de dados. Os demais endereços no EFTIRR permanecem com os mesmos endereços para os FTIRRs que não sofreram modificações pela transação corrente. Após finalizada essa etapa, o RaaB futuro, com o Ntriangulo mais atual, é gravado em disco e, só após a concretização dessa operação de escrita é que a base terá novo estado que incorporará tais atualizações. Se ocorrer alguma falha, o RaaB com o estado anterior da base será considerado na tarefa de recuperação. Portanto, ou se efetiva por completo o *commit* da transação para que a base de dados assuma o novo estado consistente, ou se assume o estado anterior refletido através do RaaB mais atual, desconsiderando-se qualquer etapa intermediária executadas pelas transações que foram interrompidas.

5.4 Considerações finais

Nesse capítulo fez-se uma exposição ampla da implementação executada para o Núcleo Gerenciador de Objetos, cujo objetivo maior é validar as propostas efetuadas para a confecção das funcionalidades suportadas por esse sistema gerenciador de objetos. Apresentou-se os módulos com suas primitivas e efetuou-se uma comparação, onde foi possível, entre as opções adotadas nesse trabalho em relação ao que se encontra na literatura em termos de técnicas de implementação para SGBDOOs.

O NuGO foi implementado através do uso de camadas com funcionalidades bem definidas e coesas. Isso permite que outras técnicas possam ser testadas pela substituição dos módulos que as implementam, como foi demonstrado nesse trabalho através das implementações do SGLog. Em um aspecto mais amplo, isso vale também para o NuGO como um todo, o qual pode ser utilizado como o "motor" gerenciador de dados em qualquer SGBDs, basta para isto que se deixe de utilizar os suportes desnecessários para um modelo mais restritivo em abstrações suportadas ou utilizando-o integralmente, quando o suporte exigido às abstrações é mais amplo.

Capítulo 6

CONCLUSÕES E FUTURAS PESQUISAS

6.1 Considerações iniciais

Motivado pela necessidade das aplicações atuais terem suporte eficiente e capaz de representar suas abstrações, cuja tarefa é proporcionada pelos Sistemas Gerenciadores de Base de Dados, o Núcleo Gerenciador foi definido de modo a propiciar um conjunto de tecnologias capazes de oferecer esse suporte de modo flexível e eficiente.

Esse capítulo apresenta as conclusões, sugestões de futuras pesquisas e discute as principais decisões de projeto tomadas durante o desenvolvimento do trabalho. Apresenta o relato das etapas que constituiram o desenvolvimento do Núcleo Gerenciador de Objetos.

6.2 Histórico do trabalho

O Grupo de Banco de Dados em Imagem - GBdI vem desde meados dos anos 80, desenvolvendo pesquisas em tecnologia para a implementação de sistemas de gerenciamento de base de dados. Nesse período, o GbdI produziu uma série de resultados, dentre eles o GEO[Tra_91], um Gerenciador de Objetos baseado no Modelo de Representação de Obejtos - MRO[Tra_92][Tra_88].

Como integrante desse grupo desde 1987 e participando das discussões sobre as tecnologias adotadas pelo GEO para o gerenciamento de objetos, pode-se ter uma visão tanto das dificuldades quanto dos benefícios proporcionados pelas tecnologias aplicadas a esse gerenciador de objetos.

Tendo em mente as dificuldades encontradas pelo GEO e as necessidades das atuais aplicações no gerenciamento de dados, fez-se um estudo bibliográfico da área com o objetivo de ampliar e atualizar o conhecimento em relação às técnicas utilizadas por outros SGBDs e as carências apresentadas por tais técnicas.

Após efetuado esse estudo, partiu-se para a definição da arquitetura a ser adotada pelo Núcleo, através da qual foi possível definir as estruturas de suporte às funcionalidades previstas para ser suportadas por esse trabalho.

A etapa de implementação do NuGO deu-se a partir do módulo de criação de base de dados SIRIUS. Posteriormente implementaram-se as camadas componentes da arquitetura do NuGO: o subsistema de gerenciamento de registros reais, o subsistema de registros lógicos e o subsistema de gerenciamento de atributos. Em conjunto com o desenvolvimento dos módulos componentes dos subsistemas foram sendo desenvolvidos módulos de testes para a verificação de estruturas de dados utilizadas. Durante essa fase, percebeu-se a possibilidade de se executar implementações alternativas para a estratégia de *clustering* adotada, onde acabou resultando nas propostas apresentadas para o subsistema de gerenciamento de registros lógicos.

Após o término da implementação das camadas que compõem o NuGO, efetuou-se um conjunto de testes com o objetivo de analisar o comportamento e o desempenho da arquitetura proposta.

6.3 Contribuições inovadoras

6.3.1 Suporte nativo às abstrações de generalização, classificação e composição

O NuGO proporciona um particular suporte integrado de maneira homogênea e coesa para as abstrações de generalização, classificação e composição. Através do módulo SG-Atrib permite-se definir essas abstrações.

Uma generalização/especialização é definida através dos SG-Atrib-Tupla e Lista, associando-se i-tupla de um objeto com as i-listas do seu supertipo, repetindo-se esse processo para quantos forem os níveis em uma hierarquia de tipos.

Os tipos e objetos são definidos através de um único módulo SG-Atrib-Composição utilizando-se das definições de colônias e objetos habitantes dessa colônia. Esse mesmo recurso permite definir a composicão de objetos, os quais definem a posição do objeto na estrutura de composição na base de dados toda.

O particionamento propiciado pela abstração de composição foi desenvolvido tendo em vista o modelo SIRIUS. No entanto, teoricamente pode ser aproveitado em

outros modelos como um recurso para suportar a implementação de versões de objetos em aplicações de CAD, gerenciamento de acesso, criação de "bases de dados virtuais", como é suportado atualmente por gerenciadores comerciais sob a designação de usuário responsável pelas tabelas (Oracle) ou múltiplas bases de dados (Sybase), com a vantagem que, ao contrário dessas implementações, não requisita a existência de arquivos físicos, ou partições em disco, distintos, com a otimização da utilização do *cache*. Essa nova tecnologia é uma das contribuições desse trabalho.

6.3.2 Armazenagem de atributos para os objetos

As implementações encontradas na literatura[Ber_94] [Cat_94] para os atributos dos objetos são executadas na forma de listas ou tuplas. Listas são flexíveis e, portanto, prediletas para a implementação de gerenciadores de dados orientados a objetos, porém bastante ineficientes. Tuplas são muito eficientes, principalmente por dispor de técnicas de gerenciamento altamente elaboradas, desenvolvidas para os gerenciadores relacionais. No entanto, até aqui, o tratamento de atributos sempre foi feito, ou através de listas, ou através de tuplas, impondo um compromisso de flexibilidade versus eficiência na decisão de como implementar um gerenciador de dados.

O NuGO utiliza os dois tipos de estruturas de dados para o armazenamento dos valores de atributos dos objetos, através de estruturas de dados i-listas e i-tuplas. Essas estruturas são mantidas concatenadas, de maneira que o suporte a ambas representa uma sobrecarga muito pequena ao sistema. Assim, pode-se preservar a eficiência do acesso através de tuplas para a busca dos atributos mais frequentemente acessados dos objetos, ao mesmo tempo que oferece a flexibilidade do acesso por listas para os demais objetos.

6.3.3 Gerenciamento de OIds

O sistema de gerenciamento de OIds implementado por esse trabalho apresenta como característica inovadora a capacidade de incluir diretamente no OId o endereço dos objetos em disco e, ao mesmo tempo, permite a reutilização do espaço utilizados para o armazenamento desses objetos evitando o reaproveitamento do OId.

Com isso, consegue-se a recuperação de um objeto de maneira imediata, pois através de seu OId determina-se o local onde está armazenado em disco sem depender de qualquer estrutura de indexação e a sobrecarga que essas estruturas introduzem. O processo de criação e eliminação dum objeto permite garantir a unicidade de seu OId durante todo o ciclo de vida do objeto e a capacidade de reutilização do mesmo espaço de armazenamento por outros objetos sem perder a unicidade na identificação.

Por outro lado, o sistema de gerenciamento de registros físicos e o sistema de *cache* das camadas internas dão suporte à realocação de registros, possibilitando que o endereço físico do objeto em disco possa mudar e permitindo a recompactação da base, ao mesmo tempo em que habilita o acesso imediato ao endereço real do objeto a partir de seu OId.

6.3.4 Esquema de *clustering*

O esquema de *clustering*, proposto pelo NuGO, utiliza-se de características semânticas para definir a distribuição de objetos em regiões físicas da base de dados. Através da abstração de composição, os objetos com seus atributos são alocados em partições físicas na base de dados, cuja proximidade física beneficia amplamente sua recuperação e armazenamento.

A maneira como esse suporte é implementado não acarreta sobrecarga operacional na execução dessa tarefa. Para cada partição semântica formada pelas aplicações existe um conjunto de estruturas de apoio que permitem a distribuição adequada das peças lógicas componentes. Dessa maneira, a própria semântica das aplicações é aproveitada nas estruturas sintáticas internas da representação dos objetos em disco, para agrupar informações semanticamente relacionadas e permitir agilizar a recuperação, através da manutenção contígua de informações que freqüentemente são acessadas conjuntamente.

Essa estratégia também favorece a concorrência entre as transações, pois como o bloqueio ocorre em RegReal e esses são compostos por RegLog semânticamente relacionados através do esquema de linhas implementadas pelo SG-Log, têm-se o agrupamento provocando o isolamento das páginas e, portanto em sintonia com os requisitos semânticos das aplicações.

6.3.5 Armazenamento de atributos de tamanho variáveis nas páginas

As estruturas de dados tradicionamente utilizadas para armazenagem de dados em bases de dados pressupõem que os dados sejam mantidos em tuplas, cujo tamanho é fixo. No entanto, aplicações mais recentes, como as que manipulam dados multimídia tem uma alta taxa de dados em tamanho variável. As técnicas tradicionais dependem da realocação de espaços de memória quando um dado já armazenado varia de tamanho, tornando custosas as operações de armazenagem e substituição desses dados em disco. A técnica adotada neste trabalho de separação entre endereços físicos e reais, e a existência de blocos de controle e de *overflow* praticamente eliminam a necessidade de realocações de dados, permitindo que a armazenagem e substituição de valores, mesmo quando implicam em variação de tamanho dos dados, se restrinja à mera re-escrita do novo valor.

6.3.6 Uso de páginas *shadow* no suporte a recuperação e gerenciamento de transação.

As operações de re-escrita de páginas durante uma transação obriga fazer-se a manutenção das versões corrigidas e anteriores, para permitir tanto que a transação se complete normalmente através de uma operação de *commit*, quanto que seja abortada e os dados anteriores recuperados. Isso implica na necessidade de se manter duas cópias de cada página modificadas por uma transação até que a transação se complete. Quando uma transação é finalizada com sucesso, normalmente é necessário que a página com a nova

versão corrigida seja lida de onde ela foi temporariamente escrita e, a seguir, escrita sobre a versão anterior, que agora é descartada. Isso implica no mínimo nas seguintes operações em disco para cada página modificada por uma transação que se completa com sucesso: escrita inicial da página modificada em um local temporário; leitura dessa página; e escrita sobre a versão anterior quando a transação se encerra. A integração entre a escrita temporária das páginas removidas do *cache* com a realocação dinâmica de endereços providas pelas estruturas TCReFO, EFTIRR e FTIRRs permite reduzir-se essas operações apenas à escrita inicial da página modificada e à escrita da estrutura EFTIRR e do registro RaaB. Levando em conta que a escrita de um registro da estrutura EFTIRR corresponde à substituição de muitos registros físicos (no mínimo 256 quando o tamanho dos registros físicos da base são os menores possíveis: 1KByte), percebe-se que o esquema aqui apresentado pode substituir $3N$ acessos ao disco (onde N é o número de páginas modificadas por uma transação) por $1+(1+\lceil 1/256 \rceil)N$ acessos a disco. Além disso, esse esquema torna natural a confirmação da transação por uma única operação atômica, a re-escrita da página RaaB. Considerando que o acesso ao disco é um dos gargalos dos sistemas de gerenciamento de dados, esse ganho é significativo para o desempenho do sistema como um todo. Em particular, é importante notar que essa técnica não se restringe a ser aplicada em gerenciadores orientados a objetos, mas pode ser aplicada a qualquer gerenciador de dados que suporte a existência de uma memória *cache*.

6.3.7 O NuGO como um gerenciador de dados para qualquer modelo de dados

O Núcleo Gerenciador de Objetos implementado nesse trabalho atende a uma gama de gerenciadores de base de dados, orientados a objetos ou não, sem restrição para que seja usado um particular modelo de dados. Por exemplo, um gerenciador relacional pode ser construído utilizando-se apenas dos módulos Lista e Tupla do SG-Atrib, mais as camadas inferiores do NuGO. Esse enfoque é interessante, pois permite o suporte a gerenciadores puramente orientados a objetos, ao mesmo tempo que permite utilizar toda a tecnologia de construção de gerenciadores relacionais que atualmente possuem uma gama maior de algoritmos desenvolvidos. Assim, a solução apresentada pode ser utilizada para suportar o desenvolvimento de outros gerenciadores de base de dados, mantendo a eficiência na recuperação de objetos armazenados.

6.4 Futuras pesquisas

6.4.1 Ampliação da capacidade de referenciação de registros físicos na base de dados através do diretório de páginas *shadow* - TCReFO

Na atual implementação do NuGO, o diretório de páginas *shadow* possui tamanho limitado pela capacidade da estrutura de armazenamento de endereços de porções desse diretório em disco, ou seja, pelo tamanho do EFTIRR - uma estrutura de endereços contida no RaaB. Embora esse limite não afete o funcionamento das tecnologias propostas no

presente trabalho, restringe a capacidade do NuGO quanto ao suporte, por exemplo, às aplicações cuja necessidade de armazenamento supera 4Gb para páginas de tamanho 1Kb.

Uma proposta para essa situação poderia ser implementada através da composição do EFTIRR em uma lista ligada de páginas, onde uma das células dessa estrutura no RaaB estaria referenciando uma lista de páginas encadeadas de novas células para o armazenamento de endereços de FTIRRs.

Como essa estrutura poderá crescer de acordo com a exigência das aplicações, pode-se utilizar um *cache* dedicado, de modo a minimizar o número de acessos a disco durante o processo de fornecimento de páginas para o seu funcionamento.

6.4.2 Esquema de recuperação

Implementação de propostas alternativas para o armazenamento e recuperação da lista de recuperação de registros físicos utilizada pela camada SGReal. A versão atual está preparada para abrir uma base de dados e recuperar as transações completadas em sessões anteriores, mantendo a integridade dos dados. No entanto, caso a última sessão tenha finalizado de maneira irregular, embora a integridade da base seja mantida, é possível que existam registros físicos no arquivo da base de dados que não estejam nem alocados para dados nem numa lista de recuperação, tornando-se portanto inacessíveis e aumentando desnecessariamente o tamanho do arquivo. Esquemas de recuperação desses registros podem ser desenvolvidos para operações tanto *off-line* (antes da base ser re-aberta) quanto *on-line*, integrado como uma transação especial junto à operação regular da base de dados.

6.4.3 Indexação e *data mining* em base de dados multimídia

O NuGO será estendido de modo a oferecer o suporte a aplicações multimídia em projeto já iniciado integrando o ICMC-USP, DCCE-IBILCE-UNESP, DC-UFSCar, e a Carnegie Mellon University em Pittsburgh, EUA. Esse projeto tem por objetivo o desenvolvimento de técnicas, algoritmos e ferramentas de software para suportar o tratamento de informações multimídia, tais como: imagens, vídeo, áudio e dados com estrutura interna complexa, como representação de móleculas e estruturas genéticas, em bases de dados multimídia. Todo o projeto é centrado em utilizar o conteúdo dos dados tratados para possibilitar a indexação e recuperação baseada em seu conteúdo, bem como a descoberta de padrões e aglomeramentos (*clusters*) nos dados. Para atingir tais objetivos, esse projeto integra as áreas de bases de dados, inteligência artificial e visualização de dados, sendo que conta com a participação de especialistas em todas elas.

6.4.4 Elaboração e aplicação de um *benchmark* para o NuGO

Elaboração de um *benchmark* baseado nos padrões estabelecidos para os sistemas de gerenciamento de base de dados orientados a objetos[Kim_95] e a ser aplicado ao NuGO.

As diretrizes para um projeto de benchmark deve considerar os seguintes elementos:

- testes para o conjunto de todas as operações que devem ser suportadas por um SGBDOO, como por exemplo, recuperar um objeto simples com o uso de seu OId, navegar num aninhamento de objetos usando seus OIds, recuperação e atualização de objetos grandes, executar consultas que envolvem classes, atributos, métodos e expressões, carregamento dinâmico de métodos, além de outras operações ;
- estabelecer características de desempenho dum grande banco de dados em um ambiente multiusuário;
- estabelecer parâmetros para a criação do *benchmark*, como, por exemplo, número de classes (por exemplo, de 100 a 200), níveis de hierarquia (20), número de sub-classes de uma classe (1-10), número de métodos em uma classe (0-5), número de valores em um atributo do tipo coleção (1-200), número de atributos BLOB em uma classe (0-1), tamanho de um BLOB (10-100 Mb), além de outros parâmetros;

Esses elementos devem ser aplicados para os vários parâmetros de ajuste oferecidos pelo sistema gerenciador de banco de dados, como por exemplo, o número de páginas no *cache* e o tamanho em bytes das páginas a serem utilizadas pelo sistema.

6.4.5 Estratégias para otimização de *clustering* físico

Na atual implementação do NuGO, não há a preocupação com as proximidades físicas entre os registros fornecidos pela lista de recuperação e a página usada mais recentemente pela aplicação. A otimização desse processo pode ser implementado sobre a atual estrutura utilizada pelo NuGO para a lista de recuperação de registros físicos, na qual a própria lista poderá decidir, entre os registros que possui, qual atende melhor a estratégia de agrupamento físico. A estrutura utilizada para o armazenamento dos blocos lógicos facilita esse processo, pois mantém armazenado os arquivos lógicos[1] na base de dados, o que possibilita saber quem são os blocos mais próximos do utilizado pela aplicação em relação aos que se encontram armazenados na lista de recuperação. Portanto, é possível, sem muito ônus, permitir que a lista de recuperação atenda as requisições de registros fisicos de modo a provocar a proximidade física entre os registros utilizados pela transação que executou tal pedido.

---

[1]Arquivo lógico: conjunto de blocos lógicos do mesmo tipo[Tra_91].

6.4.6 Outras propostas

Um conjunto de outras propostas de trabalhos futuros devem ser executados como continuação e extensão desse trabalho. Entre elas podem ser citadas as seguintes:

A utilização do NuGO como suporte a aplicações GIS;
Utilização do SIRIUS como gerenciador de documentos em aplicações para Web;
Suporte à evolução de esquemas;
Esquemas de replicação de dados;
Backup total e incremental;
Controle de versões;
Suporte à distribuição de objetos;
Suporte a dados semi-estruturados;
Implementação de uma Interface de Programação de Aplicações - API.

6.5 Conclusões gerais

Neste trabalho apresentou-se uma implementação para um Núcleo Gerenciador de Objetos, denominado NuGO. Esse Núcleo é parte de um projeto mais amplo, cujo objetivo é a construção dum Gerenciador de Objetos apoiado no modelo de dados SIRIUS e que permita o suporte a aplicações multimídia. Esse projeto vem sendo desenvolvido pelo Grupo de Base de Dados e Imagem e integra as instituições ICMC-USP, DCCE-IBILCE-UNESP, DC-UFSCar, e a Carnegie Mellon University em Pittsburgh, EUA.

Os sistemas gerenciadores de banco de dados tradicionais, especificamente os relacionais, não foram concebidos através de suas tecnologias para suportar as complexidades apresentadas pelas aplicações surgidas ao longo desses últimos anos. Embora os SGBDs relacionais tenham se apresentado de modo eficiente no suporte às aplicações comerciais, ele é rígido quando se trata de proporcionar o gerenciamento de dados para as aplicações multimídia, projetos de engenharia CAD/CAM/CASE e até mesmo a complexidade das atuais aplicações comercias. Os gerenciadores orientados a objetos têm-se apresentado habilitados para o suporte às novas aplicações.

Embora capazes de representar com naturalidade as complexidade apresentadas por essas aplicações, os Gerenciadores de Objetos têm se mostrado lentos. As técnicas de implementação têm considerado o uso das tecnologias desenvolvidas para o modelo relacional, o que tem originado a criação dos modelos relacionais-objetos. No entanto, esses gerenciadores são uma solução temporária, pois restringe muitas das funcionalidades dos conceitos de orientação a objetos "puros". Essa tese apresenta um conjunto de técnicas de implementação que resolve alguns dos principais problemas técnicos para a construção de um gerenciador de objetos eficiente e flexível.

O Núcleo Gerenciador de Objetos apresenta-se capaz de integrar a flexibilidade e a eficiência em sua construção. Através do suporte às abstrações como generalização/especialização, classificação/instanciação etc utilizando-se da integração entre esquemas e dados; atributos monovalorados, multivalorados, estruturados, longos e

extras apresenta-se capaz de oferecer a flexibilidade exigida pelas aplicações mais recentes. O NuGO demonstra-se eficiente, pois oferece, através do gerenciamento de OIds, a possibilidade de acessar os objetos diretamente sem o uso de estruturas de índices e mantendo ainda as propriedades que se esperam de um identificador de objetos; permite a composição de atributos dos objetos através de estruturas integradas de listas e tuplas com pouca sobrecarga na operacionalização pelo Núcleo; oferece esquemas de *clustering* lógico eficiente baseado na abstração de composição e suporte a transações longas com acessos reduzidos ao disco. Por outro lado, o NuGO pode também servir como "motor" para qualquer sistema gerenciador de base de dados independentemente do modelo de dados.

A implementação das características propostas para o NuGO permite contribuir para o avanço das tecnologias proporcionadas aos sistemas gerenciadores de base de dados com vistas aos objetivos propostos pelo presente trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

[Ara_98] Araújo, M. R .B. - *Representação de Construtores Semânticos em SIRIUS e Suporte através de um Editor de Esquemas*, Dissertação de Mestrado, ICMC-Universidade de São Paulo, São Carlos, Agosto de 1998.

[Atk_89] Atkinson, M.; Bancilhon, F; DeWitt, D.; et al - *The Object-Oriented Database System Manifesto*, In Proc. 1st DOOD Conference Japan, 1989.

[Aur_99] Aurélio Buarque de Holanda Ferreira - *Novo Dicionário Aurélio - Século XXI,* Ed. Nova Fronteira, Novembro de 1999

[Ber_94]Bertino, E.; L. Martino - *Object-oriented Database Systems: Concepts and Architectures*, Ed. Addison-Wesley, 1994.

[Ber_95] Bertino, E.; Foscoli, P. - *Index Organizations for Object-Oriented Database Systems*, IEEE Transactions on Knowledge and Data Engineering, vol. 7, No. 2, Abril de 1995.

[Bia_96] Biajiz, M. - *Representação de Modelos de Dados Orientados a Objetos Através da Parametrização de Abstrações,* Tese de Doutorado, IFSC - Universidade de São Paulo, São Carlos, 1996.

[Bor_97] Borges, K.A.V. - *Modelagem de Dados Geográficos: Uma extensão do Modelo OMT para Aplicações Geográficas*, Dissertação de mestrado, Fundação João Pinheiro, Belo Horizonte, 1997.

[Bro_90] Brodie, M.L.; Bancilhon, F.; Harris, C.; et al. - *Next Generation Database Management Systems Technology* - in Dedutive and Object-Oriented Databases, Elsevere Science Puplishers, Amsterdam, The Netherlands, 1990.

[But_91] Butterworth, P.; Otis, A.; Stein, J. - *The GemStone Object Database Management System*, Communications of the ACM, vol. 34, no. 10, Outubro 1991.

[Cat_94] R. G. G. Cattel - *Object Data Management: Object-Oriented and Extended Relational*, Addison-Wesley Ed., 1994.

[Cat_97] Cattell, R.G.G.; Barry, D.K. - *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publ., 1997.

[Che_76] Chen, P. - *The Entity Relationship Mode-Toward a Unified View of Data*, ACM Transaction on Database Systems, Março de 1976.

[Cod_70] Codd, E.F. - *A Relational Model for Large Shared Databanks* - Communications of the ACM, 13, 6, Junho de 1970.

[Cod_71] Cood, E. - *A Data Base Sublanguage Founded on the Relational Calculus*, Proceeding of the ACM SIGFIDET Workshop on Data Description, Acess, Control, Novembro de 1971.

[Cod_72] Cood, E. - *Relational Completenesss of Data Base Sublanguages,* in Rustin, 1972.

[Dat_93] Date, C.J.; Darwen, H. - *A Guide to the SQL Standard*, 3ed., Addison-Wesley, 1993.

[Deu_90] Deux, O. et al - *The Story of O_2,* IEEE *Transaction on Knowledge and Data Engineering*, pp 91-108, março de 1990.

[Deu_91] Deux, O. et al - *The O_2 System*, Communications of the ACM, pp. 34-48, Outubro de 1991.

[Dew_90] DeWitt, D.; Futtersack, P.; Maier, D.; Vélez, F. - *A Study of Three Alternative Workstation/Server Arquitetures for Object-Oriented Database Systems*, in: Proceeding of the 16th VLDB Conference, Brisbane, Austrália, Agosto de 1990.

[DTB_71] *Report of the CODASYL Data Base Task Group*, ACM, Abril de 1971.

[Elm_85] Elmasri, R.; Weeldreyer, J.; Hevner, A. - *The Category Concept: An Extension to the Entity-Relationship Model*, International Journal on Data Knowledge Enginnering, 1:1, Maio de 1985.

[Elm_00] Elmasri, R.; Navathe, S.B. - *Fundamentals of Database Systems*, 3a. Ed., Addison Wesley, 2000.

[Fis_87] Fishman, D.H.; Beech, D.; et al - *Iris: An Object-Oriented Database Management System,* Communications of the ACM, 1987.

[For_93] Fornari, M.; Golendziner, L. - *Evolução de Esquemas de DadosUtilizando Versões em BD Orientado a Objetos,* in: Simpósio Brasileiro de Banco de Dados. 8, Campina Grande, 1993.

[Gam_95] Gamache. A.; Sahraoui, N. - *Addressing Techniques Used in Database Object Managers O_2 and ORION*, SIGMOD Record, vol. 24, No. 3, pp. 50-55, Setembro de 1995.

[Gaz_97] Gazeta, W. - *Estudo sobre a Representação de Estruturas de Atributos Multivalorados em Base de Dados Orientada a Objetos*; Dissertação de Mestrado, ICMC-Universidade de São Paulo, São Carlos, Julho de 1997.

[Gem_00] www.gemstone.com, Abril de 2000.

[Gra_96] Gray, J. - *Evolution od Data Management* - IEEE Computer, Outubro de 1996.

[Gro_97] Grosky, W. I. - *Managing Multimedia - Information in Database Systems*, Communications of the ACM, Vol. 40, No. 12, Dezembro de 1997.

[Ham_81] Hammer, M.; McLeod, D. - *Database Description with SDM: A Semantic Data Model*, ACM Transaction on Database Systems, 6:3, Setembro de 1981.

[IBE_00] www.ibex.ch/techsum/index.htm, Abril de 2000

[Ish_96] Ishikawa, H.; Yamane, Y. et al - *An Object-Oriented Database System Jasmine: Implementation, Application, and Extension*, IEEE Transactions on Knwledge and Data Engineering, vol.8, n. 2, Abril de 1996.

[Jin_98] Jin, B.; Feng, Y. - *Adaptive Object Addressing*, in: Proceeding of the Technology of Oriented-Object Language e Systems Tools, 1998.

[Kho_94] Khoshafian, S. - *Banco de Dados Orientado a Objeto*, Livraria e Editora Infobook S.A., 1994.

[Kho_89] Khoshafian, S.; Copeland, D. - *Object Identity*, pp. 37-46 in: B. Zdonik; D. Maier - *Readings in Object-Oriented Database Systems*, Morgam Kaufmann Ed., 1989.

[Kim_87] W. Kim; H. T. Chou; J. Banerjee - *Operations And Implementation of Complex Objects*, in: 3rd IEEE International Conf. on Data Engineering. Addison-Wesley 1987.

[Kim_90] Kim, W.; Garza, J.; Ballou, N.; Woelk, D. - *Architecture of the ORION Next-Generation Database System*, IEEE Transactions on Knowledge and Data Engineering, 2(1), Março de 1990.

[Kim_92] Kim, W. - *Introduction to Object-Oriented Database*, The MIT Press, 3a. Ed., 1992.

[Kim_95] Kim, W. - *Modern Database Systems*, Addison-Wesley, 1995.

[Kor_97] Korth, H.F.; Silberschatz, A - *Database Research Faces the Information Explosion* - Communication of the ACM - Vol. 40, No. 2, Fevereiro de 1997.

[Lam_90] Lamb, C.; Landis, G.; Orenstein, J.; Weinreb, D. *The ObjectStore Database System*, Communications of the ACM, Vol 34, No. 10, Outubro de 1991.

[Mau_98] Mauro, R.C.; Zimbrão, G.; Brügger, T.S. et al - *GOA++: Tecnologia, Implementação e Extensões aos Serviços de Gerência de Objetos*, http://www.cos.ufrj.br/~goa, 1998.

[Moh_92] Mohan, C.; Haderle, D.; Lindsay, B.; Pirahesh, H.; Schwartz, P. - *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging*, Tecnical Object Database Systems, Março de 1992.

[Obj_00] www.objectdesign.com/objectstore, Abril de 2000.

[OOT_97] www.ootech.com.au - *Techinical Overview, O_2Technology*, Janeiro de 1997.

[San_79] Santos, C. dos; Neuhold, E.; Furtado, A. - *A Data Type Approach to the Entity-Relationship Model*, in: ER Conference, 1979.

[San_98] Santos F., R.F. - *Algoritmo para o Gerenciamento de Cache para a Memória externa baseado na política de substituição Seleção Natural e sua implementação como um tipo abstrato*, Relatórios Técnicos do ICMSC-Universidade de São Paulo, São Carlos, 1998.

[She_97] Shekhar, S.; Coyle, M.; et al - *Data Models in Geographic Information Systems*, Communications of the ACM, Vol. 40, No. 4, Abril de 1997.

[Sil_91] Silberschatz, A.; Stonebreaker, M.; Ullman, J. - *Database Systems: Achievements and Opportunities* - Communications of the ACM, Vol. 34, No. 10, Outubro de 1991.

[Sil_96a] Silberschatz, A.; Stonebreaker, M.; Ullman, J. - *Database Research: Achievements and Opportunities into the 21st Century*, ACM SIGMOD Record, Março de 1996.

[Sil_96b] Silberschatz, A.; Zdonik, S.; et al. - *Strategic Directions in Database Systems - Breaking Out of the Box* - ACM Computing Surveys, vol. 28, No. 4, Dezembro de 1996

[Sil_99] Silberschatz, A.; Korth, H.F.; Sudarshan, S. - *Sistema de Banco de Dados* - Makron Books, 3a. edição, 1999

[Smi_77] Smith, J.; Amith, D. - *Database Abstractions: Aggregation and Generalization*, TODS, 2:2, Junho de 1977

[Sto_89] Stonebraker, M. - *Future Trends in Database Systems* - IEEE Transactions on Knowledge and Engineering, Vol. 1, No. 1, Março de 1989.

[Sto_76] Stonebraker, M., Wonk, E., Kreps, P., Held, G. - *The Design and Implementation of INGRES*, ACM Transaction on Database Systems, 1:3, Setembro de 1976.

[Tak_93] Takai, O.K. - *Representação de Atributos com Características Gráficas em uma Base de Dados Orientada a Objetos,* Dissertação de Mestrado, ICMC - Universidade de São Paulo, São Carlos, 1993.

[Tra_86] Traina, C.T. - *Máquina e Modelo de Dados Dedicados para aplicações de Engenharia,* Tese de Doutorado, IFSC-Universidade de São Paulo, São Carlos, 1986.

[Tra_87] Traina, A. J. M. - *Desenvolvimento de Software Básico para Gerenciamento de um Sistema Multiusuário Distribuído*, Dissertação de Mestrado, ICMC-Universidade de São Paulo, São Carlos, Fevereiro de 1987.

[Tra_88] Traina Jr, C. - *Um Modelo de Representação de Objetos*, in: Anais de 3o. Simpósio Brasileiro de Banco de Dados, Recife, Março de 1988.

[Tra_91] Traina Jr, C. - *GEO: Um Sistema de Gerenciamento de Base de Dados Orientado a Objeto - Estado atual de desenvolvimento e implementação* - in: VII Simpósio Brasileiro de Banco de Dados, Manaus, 1991.

[Tra_92] Traina Jr, Caetano; Slaets, J.F.W. - *MRO: Um Modelo de Representação de Objetos,* Notas Técnicas do ICMC-Universidade de São Paulo, São Carlos, 1992.

[Tra_94] Traina Jr. C.; Traina, A.J.M.; Biajiz, M. - *O Papel da Abstração de Instanciação num Meta-Modelo de Abstrações para BDOO*, in: Anais do IX Simpósio Brasileiro de Banco de Dados, pg. 173-187, São Carlos - S.P., Setembro de 1994.

[Zlo_75] Zloof, M. - *Query By Exemple*, NCC, AFIPS, 44, 1975.

[Zlo_82] Zloof, M. - *Office By Exemple: A Business Language That Unifies Data, Word Processing, and Eletronic Mail*, IBM Journal, 21:3, 1982

[Zha_99] Zhang, Y.; Kambayashi, Y.; Yang, Y.; Sun, C. - *A Novel Timestamp Ordering Approach for Co-existing Traditional and Cooperative Transaction Processing*, IEEE Computer, 1999.

[Zan_95] Zand, M.; Collins; Caviness, D. - *A Survey of Current Object Oriented Databases*, The DATA BASE for Advances in Information Systems, Vol. 26, No. 1, Fevereiro de 1995.

[Wil_90] Wilkinson, K.; Lingbaek, P.; Hasan, W. - *The Iris Architecture and Implementation*, IEEE Transactions on Knowledge and Data Engineering - Special Issue on Database Prototype Systems, Vol. 2, No. 1, Março de 1990.