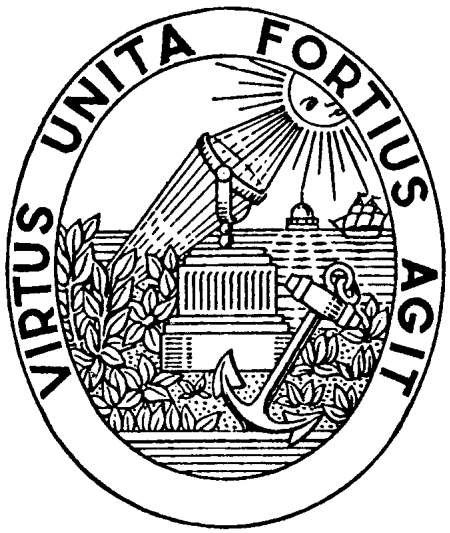
José Manuel de Castro Torres

Modelo Relacional versus Modelo Orientado Por Objectos

Estudo de uma Base de Dados de Descrição Arquivística



Faculdade de Engenharia da Universidade do Porto

Setembro de 1997

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

MODELO RELACIONAL versus MODELO ORIENTADO POR OBJECTOS
Estudo de uma Base de Dados de Descrição Arquivística

José Manuel de Castro Torres

Licenciado em Engenharia Electrotécnica e de Computadores pela Faculdade de Engenharia da Universidade do Porto

Dissertação submetida para satisfação parcial dos
requisitos do grau de mestre
em
Engenharia Electrotécnica e de Computadores
(Área de especialização de Telecomunicações)

Dissertação realizada sob a supervisão de

Professor Doutor Gabriel Torcato David,

da Faculdade de Engenharia da Universidade do Porto

Porto, Setembro de 1997

Resumo

As bases de dados orientadas por objectos, uma tecnologia emergente, têm tido como principais campos de utilização aqueles em que as bases de dados relacionais não se adequam. Este facto contraria algumas previsões do início da década de noventa que se referiam à substituição do modelo relacional pelo modelo orientado por objectos. No entanto, a explosão do uso da Internet, o surgimento da linguagem de programação orientada por objectos Java, o aparecimento e consolidação da norma ODMG para bases de dados orientadas por objectos e a evolução tecnológica em geral têm-se revelado factores potenciadores desta tecnologia. Os principais fabricantes de bases de dados relacionais têm respondido com a incorporação de características cada vez mais sofisticadas, convergindo no sentido da proposta de norma SQL3 para sistemas relacionais. Esta norma *de facto* dos sistemas de gestão de bases de dados relacionais incorpora conceitos herdados das bases de dados orientadas por objectos, introduzindo alterações profundas desde a última versão de 1992.

Nesta tese é apresentada uma aplicação que visa a automatização do processo de descrição arquivística. O modelo conceptual da aplicação é inspirado nas normas internacionais que regem a prática arquivística, disponibilizadas recentemente pelo Conselho Internacional de Arquivos. A aplicação, com uma componente de processamento de dados intensiva, foi desenvolvida usando um sistema de gestão de bases de dados orientado por objectos. A escolha desta tecnologia para aplicar ao problema em causa deve-se essencialmente à natureza intrinsecamente hierárquica do modo como está organizada a entidade central do modelo, a Unidade de Descrição, correspondente a um documento ou um conjunto de documentos relacionados.

Tendo como caso de estudo esta aplicação, procede-se no presente trabalho a um estudo comparativo entre as tecnologias de bases de dados citadas e das normas internacionais respectivas.

Abstract

Object Oriented Databases are an emergent technology which have been mainly used in applications where relational systems where not suitable. This fact, contradicts some prognostics made in the early 90's which antecipated the replacement of the relational model by the object oriented one. Nevertheless, the booming of Internet, the appearance of the Java object oriented programming language, the birth and consolidation of the standard ODMG for object oriented databases and, finally, the general technological evolution have been major stimulus to the growth of object oriented databases. Meanwhile, the main relational databases vendors included object oriented features in their products converging towards the SQL3 working draft standard for relational systems. This SQL3 standard includes concepts inherited from object oriented databases, being substantially different from its predecessor the SQL-92.

This thesis presents an application for the automation of the archivist description process. The conceptual model of the application is based on the international standards for the archivist practice which have been approved by the International Archival Council. In the development of the application an object oriented database system have been used. This technology was selected mainly due to the hierarchical nature of the organization of the *Description Unity* in this model. This Description Unity, which is the central entity of the model, represents a single document or a related set of documents.

Using this application as a case study, a comparative study between the two database technologies mentioned and the associated standards have been developed.

Agradecimentos

Agradeço ao meu orientador, Professor Gabriel Torcato David, pela disponibilidade sempre demonstrada, pela grande utilidade das suas sugestões, revisões, e criticas ao presente trabalho e pelas melhores condições de trabalho que me proporcionou.

À Lucy pela enorme ajuda na elaboração e revisão do trabalho, pelo incentivo, o apoio, o amor e a amizade.

Aos meus familiares e amigos, nomeadamente o meu Pai, minha Mãe, o Toni e a Ana pela paciência, apoio e compreensão que sempre demonstraram.

Ao Luís Paulo que sempre me incentivou e ajudou ao longo de todo o Curso de Mestrado, pela leitura deste texto, pelas valiosas sugestões.

Ao Professor Raúl Vidal pelo apoio e motivação ao longo de todo o curso de Mestrado.

Índice

1 INTRODUÇÃO 1

1.1 MOTIVAÇÕES 2
1.2 OBJECTIVOS 2
1.3 CONTRIBUIÇÃO ESPECÍFICA 3
1.4 ESTRUTURA DA TESE 3

2 MODELO RELACIONAL 4

2.1 CONCEITOS SOBRE O MODELO RELACIONAL 4
2.1.1 Domínios e Atributos 4
2.1.2 N-Tuplos e Relações 5
2.1.3 Características das Relações 6
2.1.4 Notação do Modelo 7
2.1.5 Atributos Chave de uma Relação 8
2.1.6 Esquemas de Bases de Dados Relacionais 9
2.1.7 Restrições de Integridade 10
2.1.8 Operações de Actualização em Relações 11
2.2 ÁLGEBRA RELACIONAL 12
2.2.1 Selecção 12
2.2.2 Projecção 13
2.2.3 Reunião 14
2.2.4 Produto Cartesiano 14
2.2.5 Diferença 14
2.2.6 Operações Derivadas 14
2.2.7 Extensão das operações: funções de agregação e junção externa 15
2.2.8 Leis Algébricas 16
2.2.9 Álgebra Relacional como Linguagem de Interrogação 16
2.3 CÁLCULO RELACIONAL: CÁLCULO SOBRE N-TUPLOS, QUANTIFICADORES E CÁLCULO SOBRE DOMÍNIOS 17
2.4 LIMITAÇÕES DO PODER EXPRESSIVO DE LINGUAGENS RELACIONAIS 19
2.5 SQL - LINGUAGEM PARA BASES DE DADOS RELACIONAIS 19
2.5.1 Definição de Dados em SQL 20
2.5.2 Manipulação de Dados 22
2.5.3 Uso do SQL em aplicações 28
2.6 DEPENDÊNCIAS FUNCIONAIS E NORMALIZAÇÃO 31
2.6.1 Definições 32
2.6.2 Dependências Triviais e Não Triviais 33
2.6.3 Regras de Inferência para Dependências Funcionais 33

2.6.4 Normalização 34

3 MODELO DE DADOS ORIENTADO POR OBJECTOS 36

3.1 CARACTERÍSTICAS DE UM SGBD ORIENTADO POR OBJECTOS 36
3.1.1 Conceitos básicos 36
3.1.2 Arquitectura 39
3.1.3 Características ligadas a SGBDOOs 40
3.1.4 Características Optativas 50
3.1.5 Outras Características 52
3.2 STANDARD ODMG93 V2.0 55
3.2.1 Modelo Objecto ODMG 57
3.2.2 Linguagem de Definição de Objectos - ODL 67
3.2.3 Linguagem de Especificação OIF - Object Interchange Format 70
3.2.4 Linguagem de Interrogação de Objectos 73
3.2.5 Ligação às Linguagens de Programação 77
3.2.6 Modelo OMG - Object Core Model 81

4 SISTEMA DE DESCRIÇÃO ARQUIVÍSTICA 83

4.1 INTRODUÇÃO 83
4.2 DESCRIÇÃO ARQUIVÍSTICA 84
4.2.1 Normas Gerais Internacionais de Descrição em Arquivo 85
4.2.2 Análise Conceptual 86
4.3 REQUISITOS E INFORMAÇÃO DA BASE DE DADOS 89
4.4 SISTEMA DESENVOLVIDO 90
4.4.1 Processo de Desenvolvimento 94
4.4.2 Utilização da Aplicação 102
4.4.3 Carregamento da Base de Dados 105
4.5 CONCLUSÕES 107

5 COMPARAÇÃO DE TECNOLOGIAS 108

5.1 EVOLUÇÃO DAS NECESSIDADES DAS APLICAÇÕES DE BASES DE DADOS 108
5.2 EVOLUÇÃO A PARTIR DO MODELO RELACIONAL 111
5.2.1 Relações NF2 111
5.2.2 Ligação entre Linguagens de Programação e SGBDs Relacionais 112
5.2.3 Extensão dos Modelos Relacionais 112
5.2.4 Terceiro Manifesto 113
5.2.5 Evolução de Sistemas Relacionais Comerciais 114
5.3 SQL3 115
5.3.1 Tipos de Dados Abstractos 116
5.3.2 Tipo Linha e Tabelas 120
5.3.3 Procedimentos 122
5.3.4 Recursão 123
5.3.5 Rotinas 124
5.3.6 Ligações a Linguagens de Programação 124
5.3.7 Triggers e Asserções 124

5.3.8 Construtores de Tipos Colecção 125
5.4 LINGUAGENS DE INTERROGAÇÃO 125
5.4.1 Convergência SQL3/OQL 127
5.5 SGBDOO VS SGBDR 128
5.6 CONCLUSÕES 131

6 CONCLUSÕES 132

6.1 RESULTADOS OBTIDOS E DESENVOLVIMENTO FUTURO 132
6.2 TENDÊNCIAS EVOLUTIVAS 134

7 REFERÊNCIAS 135

8 ANEXO A 139

Lista de Figuras

2.1 Diferença de terminologia entre o Modelo Relacional e SQL 20
2.2 Sintaxe de operações de junção em SQL-92 26

3.1 Objecto complexo 43
3.2 Processo de desenvolvimento de uma BDOO e aplicação 57
3.3 Hierarquia de tipos primários do modelo ODMG 62
3.4 Diagrama de relacionamentos 64
3.5 Interfaces da Meta -informação 66
3.6 Mapeamento de ODL para outras linguagens 68
3.7 Diagrama OMT para Detentores 69
3.8 Passos para construção de aplicação C++ para um SGBDOO. 78
3.9 Ligação de ODMG à Linguagem Smalltalk 79

4.1 Exemplos de esquemas de níveis 86
4.2 Divisão esquemática do modelo da aplicação 87
4.3 Subesquema que modeliza a estrutura de níveis 87
4.4 Estrutura de uma aplicação que usa a biblioteca Tcl 91
4.5 Arquitectura de processos do ObjectStore 93
4.6 Processo de desenvolvimento da base de dados 97
4.7 Processo de desenvolvimento da aplicação e base de dados 99
4.8 Janela principal da aplicação de descrição arquivística 103
4.9 Estrutura de um registo Grupo de Arquivos 106
4.10 Esquema da aplicação de conversão 106

A.1 Modelo conceptual da aplicação de descrição arquivística 140

1 Introdução

Durante a década de sessenta, o crescimento do volume de dados a processar por aplicações como sistemas de reserva de bilhetes de avião, sistemas de armazenamento de operações bancárias ou bases de dados empresariais com vários tipos de informação como vendas, funcionários, clientes e fornecedores, levou ao aparecimento de sistemas capazes de lidar com essas quantidades de informação, fornecendo serviços como indexação, transacções, segurança e linguagens de definição e de interrogação dos dados. O objectivo era elevar o nível de abstracção com que os programadores trabalhavam e consequentemente aumentar a sua produtividade. Além disso, conseguia-se aumentar a qualidade dos dados e promovia-se a sua partilha. Foram estes os primeiros Sistemas de Gestão de Bases de Dados (SGBDs), implementados em sistemas centralizados e constituindo um avanço significativo relativamente aos sistemas de ficheiros.

A complexidade crescente, quer dos SGBDs quer das aplicações, conduziu a que nos anos setenta, os SGBDs relacionais se impusessem no mercado pela sua elegância, simplicidade e facilidade de utilização. A clareza e a sólida fundamentação dos seus conceitos essenciais permitiram que todos os intervenientes, fabricantes, analistas e utilizadores, se entendessem em torno de um modelo comum. A divulgação dos sistemas relacionais foi acompanhada de uma progressiva adesão das organizações à construção de sistemas de informação assentes em bases de dados, em particular nas áreas de gestão.

Nos anos oitenta, a generalidade das organizações tinha já automatizado as funções que dependiam de aplicações tradicionais. Estas caracterizavam-se por um reduzido número de tabelas diferentes, de estrutura estável, tipicamente de grande dimensão e lidando com informação textual ou numérica.

Depois desta massificação ocorrida na área de gestão empresarial, nos anos noventa as bases de dados invadem áreas de aplicação que antes estavam dominadas por aplicações específicas ligadas a áreas como automatização de escritório, científicas ou do ramo de engenharia, e originam outras como fornecimento de certos serviços através da Internet. A disponibilização de suportes de armazenamento secundário a custos sistematicamente mais baixos e capacidades de armazenamento cada vez mais elevadas, permite que as bases de dados possam armazenar imagens, audio, vídeo e outros tipos de informação que ocupam um espaço muito considerável quando comparado com tipos de dados como números inteiros ou cadeias de caracteres. Surge assim a necessidade de lidar com tipos complexos de dados.

Entretanto aparece na década de oitenta uma tecnologia de bases de dados descendente do paradigma de programação orientado por objectos, que apresenta uma riqueza semântica e capacidade de abstracção superior à do modelo relacional.

A área de bases de dados vê-se desse modo confrontada com dois modelos que representam duas diferentes abordagens ao conceito de sistema de gestão de base de dados e uma base de aplicações em plena expansão e mutação.

1.1 Motivações

Com a publicação em 1994 das primeiras normas internacionais de descrição arquivística, representando um avanço qualitativo no domínio da catalogação de documentos de arquivo, criou-se um grande interesse no sentido da modernização tecnológica dos respectivos sistemas, neste novo quadro. Desta forma, o desenvolvimento de um sistema de descrição arquivística, que considere as normas citadas, constitui uma aplicação de grande utilidade no âmbito da circulação de informação relativa aos documentos de arquivo.

A natureza intrinsecamente hierárquica do problema a par com o cruzamento entre a organização lógica e o armazenamento físico apontam inequivocamente para o interesse de testar a aplicabilidade das tecnologias orientadas por objectos na análise e implementação de um sistema de descrição arquivística. Contudo, as grandes quantidades de dados envolvidas e o seu baixo dinamismo aconselham a que se considere também SGBDs relacionais para a componente de armazenamento intensivo de dados do sistema.

Coloca-se portanto a questão da avaliação da adequação das duas tecnologias de bases de dados mencionadas, numa perspectiva comparativa, de modo a abarcar aspectos aplicacionais e tecnológicos e a analisar em que medida estes se relacionam com os modelos referidos.

1.2 Objectivos

Os principais objectivos delineados para o trabalho aqui exposto são:

- Estudo do modelo de dados relacional e normas associadas;
- Estudo do modelo orientado por objectos e normas associadas;
- Identificação da adequação das tecnologias de bases de dados orientadas por objectos perante situações concretas;
- Estudo das modalidades de aplicação ao problema de automatização da catalogação arquivística das técnicas orientadas por objectos;
- Implementação de um protótipo com base num modelo do problema de descrição arquivística;
- Avaliação da adequação da tecnologia orientada por objectos aplicada ao problema anterior por comparação com bases de dados relacionais.

O presente trabalho enquadra-se num projecto mais ambicioso que comporta todas as fases de desenvolvimento de uma aplicação de descrição arquivística para utilização em situação real num arquivo distrital.

1.3 Contribuição Específica

O trabalho aqui apresentado "rompe" por duas frentes:

- Processo de descrição arquivística. Como o automatizar tendo em conta as normas internacionais pelo qual se rege, recorrendo à tecnologia das bases de dados orientadas por objectos;
- Comparação de duas tecnologias de bases de dados. Como podem enfrentar a evolução tecnológica que impulsiona o aparecimento das aplicações de bases de dados da próxima geração [50].

Nesta perspectiva, tenta-se aplicar uma tecnologia emergente de bases de dados - bases de dados orientadas por objectos, usada intensamente em determinadas áreas específicas, ao problema de descrição arquivística. É ainda importante realçar o carácter de actualidade das normas aplicadas à prática arquivística, razão acrescida para o interesse do trabalho em causa.

1.4 Estrutura da Tese

A dissertação compreende mais cinco capítulos para além deste primeiro de carácter introdutório.

No segundo capítulo é descrito o modelo de dados relacional. Contém uma descrição dos conceitos nucleares do modelo, da álgebra e cálculo relacionais. A seguir, é descrita a linguagem SQL, norma *de facto* dos sistemas de gestão bases de dados relacionais. Conclui-se com uma abordagem da teoria da normalização.

O terceiro capítulo contempla uma descrição dos principais conceitos e características associadas à tecnologia orientada por objectos aplicada aos sistemas de gestão de bases de dados. É ainda considerada uma descrição da mais recente versão da norma ODMG para sistemas de gestão de bases de dados orientadas por objectos.

O quarto capítulo é dedicado à apresentação da aplicação de descrição arquivística desenvolvida no âmbito deste trabalho. Aspectos relacionados com a aplicação como os seus requisitos, fases de desenvolvimento e respectiva utilização, são expostos neste capítulo. Contém ainda uma breve descrição dos componentes utilizados no desenvolvimento da aplicação. Encerra com o tecer de algumas conclusões sobre o processo de desenvolvimento da aplicação.

No quinto capítulo analisa-se o impacto que as novas aplicações com processamento intensivo de dados causaram nos sistemas relacionais. Este contexto tem determinado a evolução da norma SQL. Apresenta-se o estado actual da norma SQL3 e justifica-se a possível convergência entre este e uma linguagem de interrogação para bases de dados orientadas por objectos - OQL. A análise dos dois modelos de bases de dados numa perspectiva comparativa seguida de algumas conclusões encerra o capítulo.

No último capítulo são retiradas as conclusões do presente trabalho e referidas algumas perspectivas de desenvolvimento futuro. São ainda apresentadas algumas tendências de evolução da área das bases de dados.

2 Modelo Relacional

Neste capítulo procura-se descrever o ubíquo modelo relacional, o qual é actualmente o modelo por excelência usado para representação de informação em bases de dados. A descrição do modelo relacional em si constitui o primeiro passo na sequência lógica do estudo de comparação entre este modelo e o modelo orientado por objectos que constitui o tema deste texto.

2.1 Conceitos sobre o Modelo Relacional

O modelo de dados relacional foi introduzido por Codd [15],[14], conseguindo impor-se a outros modelos dada a sua simplicidade e sólida fundamentação teórica, assente na teoria matemática das relações.

No modelo relacional, a informação é representada na base de dados como uma colecção de relações. Cada relação pode ser vista como uma tabela. As linhas representam colecções de valores relacionados entre si. Na verdade, uma linha pode ser interpretada como representando uma entidade, através das suas características relevantes, ou uma ligação entre entidades. Os valores numa coluna correspondem todos à mesma característica nas várias linhas. Às colunas está associada uma designação.

O nome da tabela e os nomes das colunas exibem habitualmente alguma riqueza semântica, no sentido de auxiliarem na interpretação do respectivo conteúdo. Na terminologia do modelo relacional, cada linha é designada por *n-tuplo*, o nome de uma coluna é designado por *atributo*, a tabela por *relação* e o conjunto de valores que pode aparecer numa dada coluna é designado por *domínio*.

2.1.1 Domínios e Atributos

Um domínio atómico D é um conjunto de valores atómicos do mesmo tipo de dados, sendo entendido por atómico um valor que é indivisível à luz do modelo relacional, ou seja, não decomponível. Nos Sistemas de Gestão de Bases de Dados comuns, os domínios atómicos são maioritariamente conjuntos genéricos como, por exemplo, o conjunto dos números inteiros, dos números reais ou de cadeias de caracteres de comprimento até um dado valor máximo. Estes conjuntos são independentes da sua aplicação, existindo para além do uso numa dada base de dados. É possível definir também domínios específicos para uma aplicação numa base de dados

particular, através da determinação de um conjunto de valores possíveis nessa coluna, habitualmente uma restrição de um tipo de dados.

Um domínio é composto por um determinado nome, um tipo de dados e um formato. Poderá ainda ser fornecida informação adicional para melhor interpretação dos valores desse domínio.

Exemplo 2.1: um domínio que represente a altura de uma dada prateleira, poderá ter como nome *Altura_Prateleira* e o valor pode vir expresso em centímetros, polegadas ou noutra unidade. □

Muitos dos sistemas comercias de bases de dados relacionais não suportam completamente esta noção de domínio, em que é possível a existência de domínios genéricos e de domínios definidos pelo utilizador, permitindo assim uma validação dos valores existentes na base de dados. Cada atributo, é definido sobre um determinado domínio, ou seja, pode assumir valores do conjunto que constitui o seu domínio. Com atributos definidos sobre um mesmo domínio, é possível realizar operações de comparação pois existe compatibilidade entre eles.

Mais geralmente é possível comparar valores de dois atributos A_i e A_j definidos em dois domínios respectivamente D_i e D_j, desde que estes sejam semanticamente compatíveis:

$$D_i \cap D_j \neq \phi,\ a_i \in D_i \cap D_j \text{ e } a_j \in D_i \cap D_j,$$

sendo a_i e a_j os valores dos atributos A_i e A_j respectivamente.

2.1.2 N-Tuplos e Relações

O esquema de uma relação R, denotado por $R(A_1, A_2, ..., A_n)$, é um conjunto de atributos $R = \{ A_1, A_2, ..., A_n \}$. Cada atributo A_i é o nome do papel desempenhado por um domínio D no esquema dessa relação. D é designado como o domínio de A_i e denotado por $dom(A_i)$. O esquema da relação é usado para a descrever, sendo R o nome da relação. O grau da relação é o número de atributos *n* no esquema da relação.

Exemplo 2.2: a estruturação da informação arquivística é, segundo as normas ISAD(G)[1], efectuada em *Unidades de Descrição* (UD). Uma UD constitui "um documento, ou conjunto de documentos, sob qualquer forma física, tratado como uma entidade, e que, como tal, serve de base a uma descrição singular"[23]. Um esquema de relação de grau seis respeitante a uma UD poderia ser:

UDescrição(CodReferência, Título, Notas, Quantidade, QuantidSobCustodia, DataProdução)

Neste esquema de relação, UDescrição é o nome da relação contendo seis atributos. O atributo DataProdução na relação UDescrição poderia ter a sua especificação de domínio como: dom(DataProdução) = Data. □

Uma instância de relação *r* de um esquema de relação $R(A_1, A_2, ..., A_n)$, denotada também por r(R) é um conjunto de *n-tuplos* $r = \{t_1, t_2, ..., t_m\}$. Cada n-tuplo *t* é uma lista ordenada de *n* valores $t = <v_1, v_2, ..., v_n>$, onde cada valor v_i, $1 \leq i \leq n$, é um elemento do domínio $dom(A_i)$ ou um valor nulo especial.

---

[1] *General International Standard Archival Description*

A definição dada para uma relação pode ser reescrita como: uma relação r(R) é um subconjunto do produto cartesiano dos domínios que definem R,

$$r(R) \subseteq (dom(A_1) \times dom(A_2) \times \ldots \times dom(A_n))$$

O produto Cartesiano contempla todas as combinações possíveis de valores dos domínios em questão. Se denotarmos o número de todos os valores possíveis, ou *cardinalidade*, de um domínio por |D|, então, assumindo que os domínios em questão têm todos cardinalidade finita, o número de n-tuplos no produto Cartesiano seria:

$$|dom(A_1)| * |dom(A_2)| * \ldots * |dom(A_n)|$$

Uma instância de uma relação num determinado momento, reflecte um estado do mundo através de um conjunto de n-tuplos válidos para o respectivo esquema. Com a mudança do estado do mundo representado, haverá também uma mudança para outra instância de relação que será o novo valor da variável relação associada a um dado esquema. O esquema de relação R apresenta um muito maior grau de estabilidade que a variável relação, sendo alterado muito raramente com, por exemplo, a adição de mais um atributo ao conjunto de atributos do esquema R.

É também possível num esquema de relação a existência de vários atributos que apresentem o mesmo domínio. Os atributos indicam diferentes papeis para aquele domínio.

Exemplo 2.3: no esquema de relação UDescrição, os atributos Quantidade e QuantidSobCustodia poderiam ter especificado o mesmo domínio: dom(Quantidade) = dom(QuantidSobCustodia) = Quantidades_UD que contemplaria o conjunto de todas as quantidades possíveis para Unidades de Descrição. □

2.1.3 Características das Relações

Fruto das definições dadas acima, identificam-se a seguir características do conceito de relação que permitem mais claramente fazer a distinção entre relação e ficheiro ou entre relação e tabela.

Ordem dos n-tuplos numa relação

Uma relação é definida como um conjunto de n-tuplos. Com base na teoria matemática das relações, depreende-se que ao conjunto de n-tuplos de uma relação não está associada qualquer ordem específica. Numa relação é efectuada a representação dos factos a um nível lógico independentemente da ordem. Existem assim múltiplas hipóteses de ordenamento lógico numa relação, representando todas a mesma relação.

Contudo, a representação da informação presente na relação, em ficheiro ou tabela, impõe uma certa ordem. As linhas da tabela, que representam n-tuplos, têm de ser visualizadas numa determinada ordem. No caso do suporte persistente tem de ser considerada uma ordem física para os seus registos.

Ordem dos valores dentro de um n-tuplo

Um n-tuplo é uma lista ordenada de n valores, logo, a ordem dos valores num n-tuplo e consequentemente dos atributos na definição do esquema da relação é importante. Existe no entanto a possibilidade de ultrapassar essa necessidade a nível lógico, desde que se mantenha a ligação entre atributos e n-tuplos.

É possível então dar uma definição alternativa de relação, em que a ordem dos valores num n-tuplo se torna desnecessária. Nesta definição, uma relação *r* de um esquema R = { A_1, A_2, ..., A_n }, é um conjunto finito de mapeamentos r = {t_1, t_2, ..., t_m}, onde cada n-tuplo t_i é uma aplicação de R em D, e D é a união dos domínios dos atributos, ou seja D = dom(A_1) ∪ dom(A_2) ∪ ... ∪ dom(A_n). Nesta definição, t(A_i) deve estar no domínio dom(A_i) para $1 \leq i \leq n$ e para cada mapeamento t existente em r. Cada mapeamento t_i é denominado n-tuplo.

Segundo esta definição, um n-tuplo pode ser considerado um conjunto de pares (<atributo>, <valor>), em que cada par representa o valor da ligação entre um atributo A_i e um valor v_i do domínio dom(A_i). Deste modo, a ordem dos atributos não é importante, pois o nome do atributo acompanha sempre o seu valor em cada n-tuplo.

A definição apresentada poderá constituir uma alternativa válida a um nível abstracto em que não existe nenhuma preferência em ter um dado atributo a preceder outro. No entanto, no armazenamento em ficheiro, os atributos podem ser ordenados fisicamente dentro de cada registo.

Valores presentes nos n-tuplos

Cada valor num n-tuplo é atómico, ou seja, não é possível dividi-lo em componentes no contexto do modelo relacional. Por esse motivo, não é permitida a existência de atributos compostos ou multi-valor. Uma representação das relações que satisfaça esta condição, diz-se estar na *primeira forma normal*. Atributos multi-valor têm de ser representados numa relação separada, e cada atributo composto tem de ser partido num conjunto de atributos simples.

Os valores em alguns atributos de um determinado n-tuplo poderão ser desconhecidos ou mesmo inexistentes nesse n-tuplo. Nesses casos é usado um valor especial designado por nulo. Em geral, poderá haver vários tipos de valores nulos, como *valor desconhecido* ou *atributo não se aplica a esse n-tuplo* ou n-*tuplo não tem valor para esse atributo*. Alguns SGBDs relacionais implementam diferentes códigos para diferentes tipos de valor nulo. No entanto, tal procedimento é complexo de analisar em termos de modelo relacional.

Interpretação de uma relação

O esquema de uma relação pode ser interpretado como um tipo de *asserção*. O esquema de relação UDescrição, apresentado anteriormente, define que uma UD tem um código de referência de arquivo país ou outro, um título, e toma forma numa determinada quantidade de unidades físicas, da qual parte ou toda está sob custódia do arquivo. Cada n-tuplo numa relação poderá ser interpretado como um *facto* ou uma instância particular de uma asserção.

2.1.4 Notação do Modelo

A notação adoptada neste texto para explanação do modelo relacional é descrita a seguir:

- Um esquema de relação R de grau n é representado por R(A_1, A_2, ..., A_n);
- Um n-tuplo t numa relação r(R) é representado por t = <v_1, v_2, ..., v_n>, onde v_i é o valor correspondente ao atributo A_i. Em cada n-tuplo, t[A_i] refere-se ao valor v_i em t para o atributo A_i, e t[A_u, A_w, ...,A_z], onde A_u, A_w, ...,A_z é uma lista (ou conjunto) de atributos de R, refere-se aos valores dos sub-tuplos <v_u, v_w, ..., v_z> de t correspondendo aos atributos especificados na lista;

- As letras Q, R, S denotam nomes de relações;
- As letras q, r, s denotam instâncias de relações;
- As letras t, u, v denotam n-tuplos;
- O nome de uma relação como por exemplo UDescrição, refere-se ao conjunto de n-tuplos nessa relação i.e., à instância da relação. O nome UDescrição(CodReferência, Título,...) refere-se ao esquema da relação;
- Os nomes dos atributos numa relação, por vezes são qualificados com o nome da relação à qual pertencem como UDescrição.CodReferência.

2.1.5 Atributos Chave de uma Relação

Uma relação é definida como um conjunto de n-tuplos. Por definição, todos os elementos de um conjunto são distintos ou seja, numa relação todos os n-tuplos têm também de ser distintos. Tal significa que não poderá haver nenhum par de n-tuplos numa relação que tenha a mesma combinação de valores para todos os seus atributos.

De um modo geral, num esquema de relação R existem subconjuntos de atributos que respeitam a condição enunciada. Se todos os n-tuplos em cada um desses subconjuntos são distintos, então no conjunto original de atributos da relação a condição é também satisfeita. Suponhamos que do esquema de relação $R(A_1, A_2, ..., A_n)$, consideramos um subconjunto de atributos $SK = \{A_1, A_2, ..., A_i\}$ com $i \leq n$. Então temos que para quaisquer dois n-tuplos t_1 e t_2:

$$t_1[SK] \neq t_2[SK]$$

Qualquer desses subconjuntos de atributos, representa uma *super-chave* do esquema de relação R. Toda a relação tem pelo menos uma super-chave, constituída por todos os atributos da relação.

Uma *chave* K de um esquema de relação R, é uma super-chave de R com a propriedade adicional de que retirando qualquer atributo A do conjunto de atributos que formam K, obtém-se o conjunto K' que já não respeita a condição para ser super-chave de R. Então uma chave K é uma super-chave mínima da qual não é possível retirar nenhum atributo e ela continuar a ser super-chave de R.

Muitas vezes as chaves são constituídas por apenas um atributo.

Exemplo 2.4: numa tabela Pessoa(BI, Nome, Morada, NC), K_1={BI} constitui uma chave, pois não existem duas pessoas com o mesmo bilhete de identidade. O mesmo sucede com K_2={NC}, o número de contribuinte. Já SK_1={BI, Nome} é superchave, mas não é chave porque $K_1 \subset SK_1$. □

O valor dos atributos de uma chave, serve para identificar univocamente cada n-tuplo numa relação R. Um conjunto de atributos ser ou não chave do esquema de relação R, é uma propriedade desse esquema de relação. É pois uma restrição que tem que acontecer em todas as instâncias desse esquema de relação. Uma chave é determinada a partir do conteúdo semântico dos atributos no esquema de relação. Essa propriedade é invariante no tempo, e deve manter-se mesmo quando se inserem novos n-tuplos na relação.

Um esquema de relação pode ter mais do que uma chave. Nesses casos, cada uma das chaves é designada por *chave-candidata*. Uma das chaves candidatas é designada para identificar os n-tuplos na relação. Essa chave passa a ser a *chave primária*. Na representação do esquema de uma base de dados relacional, utiliza-se a convenção de que em cada um dos seus esquemas de relação, os atributos cujo nome está sublinhado, constituem a chave primária desse esquema de relação. Quando um esquema de relação contém diversas chaves candidatas, é vulgar optar-se pela escolha da chave candidata com o menor número de atributos para ser chave primária.

2.1.6 Esquemas de Bases de Dados Relacionais

Um esquema de uma base de dados relacional S é um conjunto de esquemas de relações S = $\{R_1, R_2, ..., R_m\}$ e um conjunto de restrições de integridade. Uma instância DB de uma base de dados relacional S é um conjunto de instâncias de relações DB = $\{r_1, r_2, ..., r_m\}$ tal que cada r_i é uma instância de R_i e que todas as instâncias de relações satisfazem as restrições determinadas para S.

É permitido que atributos que estão associados ao mesmo conceito no mundo real tenham nomes que podem ou não ser idênticos nas diferentes relações onde estão a ser utilizados. Esta noção pode alargar-se ao caso em que um atributo aparece duas ou mais vezes na mesma relação com nomes distintos e no entanto, representando ainda o mesmo conceito. Também é possível ter, em diferentes relações, atributos com o mesmo nome e que no entanto, representam diferentes conceitos no mundo real.

Exemplo 2.5: esquema da base de dados relacional INSTALAÇÃO, relativa à parte de Unidades de Instalação do sistema de descrição arquivística a qual descreve a arrumação física dos documentos. De notar que o atributo *CodReferencia* na relação UDescrição, representa o mesmo conceito no modelo que o atributo *CodUD* na relação Arrumacao_UD. O mesmo acontece com o par de atributos *Referência* e *RefUI* respectivamente das relações Uinstalação e Arrumacao_UD, assim como os dois pares *NumPrateleira* e *Numero* de Uinstalação e Prateleira, e *NumEstante* e *Numero* de Prateleira e Estante.

Udescrição

CodReferência	Título	Notas	Quantidade	QuantidSobCustodia	DataProdução

Uinstalação

Referência	Tipo	ExtensãoLinear	NumPrateleira

Prateleira

Numero	Altura	NumEstante

Estante

Numero	Comprimento

Arrumacao_UD

CodUD	RefUI

□

2.1.7 Restrições de Integridade

As restrições de integridade são especificadas num esquema de base de dados, e deverão ser respeitadas em todas as instâncias desse esquema. Tem-se no modelo relacional três tipos de restrições de integridade:

Restrições de Chave

Especificam quais as chaves candidatas de cada um dos esquemas de relação da base de dados. Os valores das chaves candidatas devem ser únicos para todos os n-tuplos em qualquer instância daquele esquema de relação.

Restrições de Integridade de Entidades

Determinam que nenhum valor de qualquer chave primária deve ser nulo. Isto porque a chave primária é usada para identificar n-tuplos individualmente numa relação. Se por hipótese, dois ou mais n-tuplos numa relação tivessem valores nulos nas suas chaves primárias, já não seria possível fazer a sua distinção dentro da relação.

Restrições de Integridade Referencial

Enquanto os dois tipos de restrições anteriores são impostos ao nível de uma relação, as restrições de integridade referencial são entre duas relações, sendo usadas para manter a consistência entre n-tuplos de duas relações.

A definição formal de integridade referencial recorre ao conceito de chave externa. Um conjunto de atributos FK num esquema de relação R_1 é uma chave externa em R_1 se satisfizer as duas condições abaixo:

- Os atributos em FK têm o mesmo domínio que os atributos da chave primária PK de outra relação R_2. Dos atributos de FK diz-se que referem a relação R_2;
- O valor dos atributos FK de qualquer n-tuplo t_1 de R_1, ou ocorre também como valor de PK para algum n-tuplo t_2 em R_2 ou é nulo. No primeiro caso tem-se $t_1[FK] = t_2[PK]$ e diz-se que o n-tuplo t_1 refere o n-tuplo t_2.

Para a especificação das restrições de integridade referencial é necessário averiguar primeiro o papel de cada conjunto de atributos nos vários esquemas de relação da base de dados. As restrições de integridade referencial derivam das associações entre entidades implícitas nos esquemas das relações.

Uma chave externa pode referir-se à sua própria relação. Acontece, por exemplo, quando aparecem numa relação dois atributos definidos no mesmo domínio, sendo um dos nomes a chave primária da relação e o outro nome a chave externa, que refere a chave primária da mesma relação.

Todas as restrições de integridade devem ser especificadas no esquema da base de dados de modo a manter-se a consistência em todas as suas instâncias. Num sistema de base de dados relacional, a linguagem de definição de dados deve prever a possibilidade de especificar os vários tipos de restrições existentes, para que o SGBD relacional possa lidar de um modo automático com elas.

Restrições Genéricas

Nos tipos de restrições mencionados até agora, não está incluída uma grande classe de restrições genéricas designadas por restrições de validação, que impõem condições aos valores atribuídos em cada n-tuplo aos atributos.

Trata-se muitas vezes de limitar a gama dos valores permitidos para um dado atributo, de forma absoluta ou condicionada pelo valor de outros atributos do mesmo n-tuplo. Os casos mais complexos podem exigir o recurso a procedimentos definidos pelo utilizador , mas cuja execução é desencadeada automaticamente pela ocorrência de operações de alteração de dados. As restrições de integridade são de natureza semântica e portanto o seu reconhecimento e definição são da estrita responsabilidade da modelização. Não se podem derivar das instâncias ou de considerações sintácticas. Quando muito pode verificar-se que uma determinada instância as respeita ou não.

2.1.8 Operações de Actualização em Relações

Existem três operações básicas de actualização de uma relação: inserção de um novo n-tuplo, apagamento de um n-tuplo e modificação de valores de certos atributos num n-tuplo. Quando se efectua uma operação de actualização, é necessário verificar se as restrições de integridade especificadas no esquema da base de dados não foram violadas. Passa-se então a enunciar as infracções às restrições de integridade que podem ocorrer, fruto de cada uma das operações de actualização:

Inserção de um n-tuplo cujo:

- valor da chave primária é idêntico ao valor da chave de um n-tuplo existente na relação;
- valor da chave primária é nulo;
- valor da chave externa não coincide com nenhuma chave primária na relação referida.

Apagamento de um n-tuplo cuja:

- chave primária se encontrava referida por uma chave externa noutra relação.

Modificação num n-tuplo do:

- valor da chave externa de modo a que o novo valor não coincida com o valor de nenhuma chave primária na relação referida;
- valor da chave primária que se encontrava referenciada por uma chave externa noutra relação;
- valor da chave primária para outro valor de chave primária já existente na relação. Muitos sistemas proíbem mesmo alterações do valor da chave primária.

Exemplo 2.6: nas instâncias das relações ESTANTE e PRATELEIRA ilustradas a seguir, da base de dados INSTALAÇÃO, podem identificar-se as chaves primárias das relações constituídas pelos atributos sublinhados bem como uma chave externa na relação PRATELEIRA representada pelo atributo NumEstante que se refere à chave primária da relação ESTANTE.

Prateleira	Numero	Altura	NumEstante
	1	50	1
	2	40	2
	3	30	2

Estante	Numero	Comprimento
	3	200
	2	250
	1	300

□

2.2 Álgebra Relacional

A álgebra relacional representa uma colecção de operações usadas para manipular relações. Estas operações são usadas para seleccionar tuplos de relações e para combinar n-tuplos pertencentes a várias relações, com vista a formular interrogações à base de dados. O resultado de cada operação é uma nova relação que pode ser posteriormente manipulada pelas operações da álgebra relacional.

As operações básicas definidas na álgebra relacional são: reunião, diferença, produto cartesiano, projecção e selecção. A partir destas podemos obter por composição outro tipo de operações, nomeadamente: intersecção e quociente. As operações referidas, podem também ser divididas entre as operações matemáticas de teoria de conjuntos: reunião, intersecção, diferença e produto cartesiano; e um outro grupo de operações desenvolvido especificamente para bases de dados relacionais que inclui, entre outras, operações de: selecção, projecção e junção.

2.2.1 Selecção

A operação de selecção é usada para seleccionar um subconjunto de n-tuplos de uma relação que satisfaçam uma dada condição. Por exemplo, poderá querer saber-se quais os n-tuplos de Estante cujo comprimento é maior do que 170 centímetros:

$$\sigma_{Comprimento > 170}(Estante)$$

Em geral, a operação de selecção é denotada por

$$\sigma_{< condição\ de\ selecção >}(<nome\ da\ relação>)$$

onde o símbolo σ é usado para denotar o operador de selecção e a condição de selecção é uma expressão lógica especificada sobre os atributos da relação.

A relação resultante da operação de selecção tem os mesmos atributos que a relação <nome da relação> que serve como argumento do operador. A expressão lógica <condição de selecção>, pode ser decomposta num conjunto de expressões:

<nome de um atributo> <operador de comparação> <constante>

ou,

<nome de um atributo> <operador de comparação> <nome de um atributo>

em que <nome de um atributo> é o nome de um atributo da relação <nome da relação>, <operador de comparação> é de um modo geral um operador do seguinte conjunto {=, <, >, ≤, ≥, ≠}, e <constante> é um valor constante. As expressões podem ser ligadas entre si por operadores lógicos E, OU, e NÃO de modo a formar uma condição de selecção genérica. Por exemplo na expressão:

$$\sigma_{(QuantidSobCustodia > 70\ E\ Quantidade < 170)\ OU\ (QuantidSobCustodia < 80\ E\ Quantidade > 160)}(UDescrição)$$

está-se a seleccionar todos os n-tuplos da relação UDescrição cuja quantidade sob custódia seja maior do que 70 e a quantidade existente menor do que 170, mais os n-tuplos cujo valor da quantidade sob custódia seja menor do que 80 e a quantidade existente menor do que 160.

De notar que os operadores de comparação referidos atrás, aplicam-se apenas a atributos cujos domínios sejam valores ordinais como números ou datas. Domínios constituídos por cadeias de caracteres do alfabeto podem ser ordenados alfabeticamente e os domínios constituídos por cadeias alfa numéricas podem ser também ordenados usando o código numérico dos caracteres. No entanto, se o domínio for um conjunto de valores não ordenado, então apenas se poderão aplicar os operadores de igualdade {=, ≠} nos seus atributos.

O operador de selecção é unário ou seja aplicado a apenas uma relação. A operação de selecção é aplicada a cada n-tuplo da relação individualmente, logo as condições de selecção não se podem aplicar a mais do que um n-tuplo. O grau da relação resultante é o mesmo da relação original sobre a qual a operação é aplicada.

A operação de selecção é comutativa:

$$\sigma_{<\text{condição de selecção 1}>}(\sigma_{<\text{condição de selecção 2}>}(R)) = \sigma_{<\text{condição de selecção 2}>}(\sigma_{<\text{condição de selecção 1}>}(R))$$

Por essa propriedade se deduz que uma sequência de operações de selecção pode ser aplicada em qualquer ordem ou mesmo em cascata, sem que se altere a relação resultado desse conjunto de operações:

$$\sigma_{<\text{cond 1}>}(\sigma_{<\text{cond 2}>}(...(\sigma_{<\text{cond n}>}(R))...)) = \sigma_{<\text{cond 1}> E <\text{cond 2}> E ... E <\text{c ond n}>}(R)$$

2.2.2 Projecção

Na operação de projecção, procede-se à remoção de alguns atributos e eventualmente reordenação dos restantes atributos da relação R que serve de argumento. Se se pensar numa relação como uma tabela, então a operação de projecção vai seleccionar algumas colunas da tabela correspondentes aos atributos escolhidos. Está-se assim a *projectar* a relação sobre esses atributos. Para se obter uma lista dos títulos e datas de produção de todas as Unidades de Descrição ter-se-ia:

$$\pi_{Titulo,\ DataProdução}(UDescrição)$$

A forma geral da operação de projecção é:

$$\pi_{<\text{lista de atributos}>}(<\text{nome da relação}>)$$

em que o símbolo π é usado para representar a operação de projecção dos atributos <lista de atributos> da relação <nome da relação>. O grau da relação resultante é igual ao número de atributos da lista, e a ordem dos atributos é a mesma da lista.

Para que a relação resultante seja válida (um conjunto de n-tuplos), a operação de projecção implicitamente remove n-tuplos duplicados originados pela projecção. O número de n-tuplos na relação resultante será então menor ou igual ao número de n-tuplos da relação original. Apenas se tem a garantia de que o número de n-tuplos da relação resultante é igual ao da relação original, quando a lista de atributos projectados inclui uma chave da relação.

Se tivermos duas listas de atributos <lista 1> e <lista 2>, em que a primeira lista está contida na segunda: <lista 1> $\subset$ <lista 2>, então

$$\pi_{<lista\ 1>}(\pi_{<lista\ 2>}(R)) = \pi_{<lista\ 1>}(R)$$

caso contrário a proposição é incorrecta.

A operação de projecção, ao contrário da operação de selecção, não é comutativa.

2.2.3 Reunião

A operação reunião de duas relações R e Q, denotada R $\cup$ Q, é o conjunto de n-tuplos que se encontram em R, em Q ou em ambas. Em álgebra relacional, apenas se pode aplicar a operação de reunião a relações com o mesmo grau (aridade) e com os domínios das colunas correspondentes compatíveis. Visto os nomes originais dos atributos serem ignorados durante a operação, novos nomes de atributos têm de ser especificados para a relação resultante.

2.2.4 Produto Cartesiano

O produto cartesiano de duas relações R e Q de grau *n* e *m* respectivamente, retorna uma relação com grau *n* + *m*, sendo os primeiros *n* atributos os presentes na relação R e os últimos *m* atributos os da relação Q, mantendo-se a ordem dos atributos das relações originais. Os n-tuplos presentes na relação obtida, representam o conjunto de todos os n-tuplos possíveis em que os primeiros *n* valores constituem um n-tuplo da relação R e os últimos *m* valores constituem um dos n-tuplos da relação Q. A operação é denotada por R $\times$ Q.

2.2.5 Diferença

A operação diferença entre duas relações R e Q, denotada por R − Q, retorna o conjunto de todos os n-tuplos de R que não fazem parte da relação Q. É condição necessária que ambas as relações tenham o mesmo grau e os domínios das colunas correspondentes compatíveis.

2.2.6 Operações Derivadas

As operações seguintes, são obtidas através das cinco operações mencionadas atrás.

Intersecção

A intersecção entre as relações R e Q com o mesmo grau é o conjunto dos n-tuplos que pertencem simultaneamente a R e a Q. Tal como na operação diferença, também é condição necessária que ambas as relações tenham o mesmo grau e os domínios das colunas correspondentes compatíveis. A operação é denotada por R $\cap$ Q.

A operação de intersecção é obtida usando operações de diferença:

$$R \cap Q = R - (R - Q)$$

Quociente

Dadas duas relações R e Q de grau *n* e *m*, respectivamente, em que *n* > *m*, e Q $\neq \phi$. Então a operação quociente entre R e Q, denotada por R $\div$ Q, representa uma relação S em que cada n-tuplo é constituído por (*n* - *m*) atributos. Para cada n-tuplo de S, $t_S = <v_1, v_2, ..., v_{n-m}>$, qualquer

que seja o n-tuplo de Q, $t_Q = <v_{n-m+1}, v_{n-m+2}, ..., v_n>$, existe sempre um n-tuplo em R, $t_R = <v_1, v_2, ..., v_{n-m}, ..., v_n>$.

É possível expressar esta operação usando operações mais básicas da álgebra relacional:

$$S = \pi_{1, 2, ..., n-m}(R) - \pi_{1, 2, ..., n-m}(((\pi_{1, 2, ..., n-m}(R) \times Q) - R)$$

Junção

Uma relação S, representa a junção θ sobre os componentes *i* e *j* de duas relações R e Q respectivamente, se o conjunto dos seus n-tuplos formar uma restrição do produto cartesiano R × Q, constituída apenas pelos n-tuplos em que é verdadeira a proposição *$i* θ *$j* em que *$i* e *$j* representam respectivamente o i-ésimo valor de cada n-tuplo da relação R e o j-ésimo valor de cada n-tuplo da relação Q. Na expressão anterior, θ representa um operador aritmético de comparação. Quando θ é substituído pelo operador de igualdade (=), a operação em questão é designada por *equi-junção*.

A operação de junção entre R e Q, é denotada por $R \underset{i\theta j}{\bowtie} Q$, podendo ser obtida usando as operações de selecção e produto cartesiano:

$$S = \sigma_{\$i\ \theta\ \$(r+j)}(R \times Q)$$

em que as relações R e Q têm respectivamente *r* e *q* atributos.

A junção natural, denotada por R ⋈S e aplicável quando os componentes dos n-tuplos em R e S forem designados por atributos, tem como operação implícita a igualdade dos atributos com o mesmo nome. Assim, cada par de tais atributos, dá origem a um único atributo, com o mesmo nome, na relação resultante.

A operação de junção natural, pode ser expressa usando as operações de projecção, selecção e produto cartesiano:

$$R \bowtie Q = \pi_{A_1,...,A_m}(\sigma_{t_R[A_1]=t_Q[A_1]\ e\ ...\ e\ t_R[A_k]=t_Q[A_k]}(R \times Q))$$

em que *k* é o número de atributos comuns às relações R (de grau r) e Q (de grau q), sendo *m = r + q - k*.

Se for efectuada a projecção sobre os atributos de R na relação resultante da operação de junção natural R ⋈Q, temos o que se designa por semi-junção entre R e Q, denotada por R ⋉ Q.

$$R \ltimes Q = \pi_{A_1,...,Ar}(R \bowtie Q)$$

em que $A_1,...,A_r$ representam os atributos do esquema de relação R. Outra expressão equivalente para expressar a semi-junção é:

$$R \ltimes Q = R \bowtie (\pi_{(A_1,...,Ar) \cap (A_1,...,Aq)}(Q))$$

onde $(A_1,...,Ar) \cap (A_1,...,A_q)$ representa a intersecção entre os atributos do esquema de relação de R e os de Q.

A junção natural referida atrás é possível ser efectuada usando duas versões da mesma relação, em que apenas variam os nomes dos atributos entre uma e outra através de renomeação.

2.2.7 Extensão das operações: funções de agregação e junção externa.

Funções de agregação

Um tipo de operações de interrogação que não pode ser expresso usando a álgebra relacional é o que implica o uso de funções matemáticas de agregação sobre colecções de valores da base de dados. Entre as funções mais vulgarmente usadas sobre colecções de dados numéricos estão:

soma de valores, média de valores, máximo, mínimo, ou quantidade de valores. Todas as funções anteriores podem ser aplicadas a todos os n-tuplos de uma relação.

Neste tipo de operações, é vulgar proceder-se a uma partição dos n-tuplos da relação com um dado valor, num ou vários atributos, sendo seguidamente aplicadas as funções de agregação independentemente a cada conjunto de n-tuplos obtido.

Junção externa

Nas operações de junção discutidas atrás, apenas os n-tuplos que satisfizessem a condição de junção eram considerados para a relação resultante. Na junção natural, por exemplo, n-tuplos de uma relação que não tivessem o correspondente na outra relação, eram eliminados da relação resultante. Para evitar esta situação e manter todos os n-tuplos de R ou de Q ou de ambos na relação resultante, é definido um conjunto de operações designadas por junções externas. Quando acontecem n-tuplos numa relação que não têm correspondência na outra, então completando esses n-tuplos com nulos de forma a que toda a informação existente apareça na junção.

Podemos ter três tipos de junções externas entre R e Q:

$R =\!\bowtie Q$, junção externa esquerda em que todos os n-tuplos de R são considerados no resultado final.

$R \bowtie\!= Q$, junção externa direita em que todos os n-tuplos de R são considerados no resultado final .

$R =\!\bowtie\!= Q$, junção externa total em que todos os n-tuplos de R são considerados no resultado final.

Esta operação é particularmente útil na definição de vistas intuitivas para o utilizador mas que, por força das regras de normalização (*ver ponto 2.6*), não são tabelas reais.

2.2.8 Leis Algébricas

Apresenta-se abaixo uma lista das principais propriedades de que gozam as operações definidas para a álgebra relacional:

A reunião é associativa e comutativa:

$R \cup (Q \cup S) = (R \cup Q) \cup S$

$R \cup Q = Q \cup R$

O produto cartesiano é associativo mas não é comutativo:

$R \times (Q \times S) = (R \times Q) \times S$

$R \times Q \neq Q \times R$

A junção natural é associativa e comutativa (devido à independência da ordem das colunas dada pela existência de atributos):

$R \bowtie Q = Q \bowtie R$

A θ-junção não é comutativa mas é associativa no caso dos índices serem válidos:

$R \underset{i\theta 1 j}{\bowtie} (Q \underset{k\theta 2 l}{\bowtie} S) = (R \underset{i\theta 1 j}{\bowtie} Q) \underset{(r+k)\theta 2 l}{\bowtie} S$

2.2.9 Álgebra Relacional como Linguagem de Interrogação

A álgebra relacional fornece um conjunto de operadores que poderão ser aplicados às relações de uma base de dados para obtenção de interrogações específicas. Neste ponto, é demonstrado o uso dos operadores relacionais descritos em três interrogações à base de dados INSTALAÇÃO. Na

sequência de operações levadas a cabo em cada interrogação, são expressas as relações temporárias usadas até obter o resultado final. Uma alternativa em termos sintácticos para expressar a mesma interrogação, poderia ser encaixar as expressões usando parênteses.

Interrogação 1: Titulo de todas as Unidades de Descrição que não estão na Unidade de Instalação que tem como referência "ref1".

$UD\_UI\_REF1 \leftarrow \pi_{CodUD}(\sigma_{RefUI = \text{'ref1'}}(Arrumacao\_UD))$
$UDS \leftarrow \pi_{CodReferência}(UDescrição)$
$UDS\_SEM\_UI\_REF1 \leftarrow (UDS - UD\_UI\_REF1)$
$RESULTADO \leftarrow \pi_{Titulo}(UDS\_SEM\_UI\_REF1 \bowtie UDescrição)$

Interrogação 2: Titulo de todas as Unidades de Descrição que estão na Unidade de Instalação cuja referência é "ref1" e na Unidade de Instalação cuja referência é "ref2".

$UD\_UI\_REF1 \leftarrow \pi_{CodUD}(\sigma_{RefUI = \text{'ref1'}}(Arrumacao\_UD))$
$UD\_UI\_REF2 \leftarrow \pi_{CodUD}(\sigma_{RefUI = \text{'ref2'}}(Arrumacao\_UD))$
$UD\_UI\_REF1\_REF2 \leftarrow (UD\_UI\_REF1 \cap UD\_UI\_REF2)$
$RESULTADO \leftarrow \pi_{Titulo}(UDS\_SEM\_UI\_REF1 \bowtie UDescrição)$

Interrogação 3: Datas de produção das Unidades de Descrição que estão em todas as Unidades de Instalação da prateleira 26.

$REF\_UIS \leftarrow \pi_{Referência}(\sigma_{NumPrateleira = 26}(UInstalação))$
$UDS \leftarrow Arrumacao\_UD \div REF\_UIS$
$RESULTADO \leftarrow \pi_{Dataprodução}(UDescrição \bowtie UDS)$

É importante referir que a expressão destas interrogações só recorre a operações ao nível relacional, e não depende de forma alguma da estratégia de implementação adoptada para as tabelas, a qual pode portanto ser alterada (por exemplo adicionando um índice ou agrupando os registos por outro critério) sem que isso obrigue a reformular as perguntas. Esta disciplina de trabalho deriva do principio da independência dos dados, definido como a imunidade das aplicações às alterações na estrutura de memorização e estratégia de acesso. Este principio é habitualmente bastante reforçado pelos SGBDs relacionais.

2.3 Cálculo Relacional: Cálculo sobre n-tuplos, Quantificadores e Cálculo sobre domínios

Outra linguagem formal para bases de dados relacionais com poder expressivo idêntico à álgebra relacional é o cálculo relacional. A principal diferença entre álgebra relacional e cálculo relacional é que, nesta última, uma interrogação à base de dados é uma expressão escrita de um modo declarativo. Em álgebra relacional, tal não acontecia visto ser necessário especificar uma sequência para as operações executadas para obter a mesma interrogação.

Uma linguagem de interrogação L diz-se *completa*, se for possível expressar nessa linguagem qualquer interrogação que se possa expressar em cálculo relacional. A maior parte das

linguagens de interrogação comerciais são completas, contendo de um modo geral, ainda maior poder expressivo do que a álgebra ou o cálculo relacional, visto contemplarem operações adicionais como agregação, agrupamento e ordenamento.

O cálculo relacional é inspirado num ramo da lógica matemática designado por cálculo de predicados. A adaptação para bases de dados é possível de dois modos. Num deles, o cálculo sobre domínios, as variáveis contêm valores dos domínios dos atributos, no outro, o cálculo sobre n-tuplos, as variáveis representam n-tuplos.

Cálculo Relacional sobre domínios

No cálculo sobre domínios, as variáveis têm como tipos os domínios dos componentes da relação aos quais estão ligadas. Para termos como resultado uma relação de grau *n*, necessitamos de ter *n* variáveis de domínio, uma para cada atributo da relação. Uma expressão de cálculo sobre domínios, tem a forma:

$$\{ x_1, x_2, ..., x_n \mid COND(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+m}) \}$$

em que $x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+m}$ são variáveis de domínio e COND é uma condição ou fórmula do cálculo relacional de domínios.

Uma fórmula é composta por átomos ou fórmulas ligados pelos operadores lógicos *or*, *and* e *not*. Cada átomo pode definir uma relação $R(x_1, x_2, ..., x_j)$, em que R é o nome da relação. Para que este átomo tenha um valor lógico verdadeiro, a lista de variáveis $<x_1, x_2, ..., x_n>$ tem de constituir um n-tuplo da relação R. Um átomo pode também ser da forma x_i *operador* x_j ou x_i *operador c* em que *operador* pertence ao conjunto dos operadores relacionais e c é um valor constante.

Cálculo Relacional sobre n-tuplos

Para o cálculo relacional de n-tuplos, é necessário especificar variáveis do tipo n-tuplo, em que cada uma delas tem como gama uma dada relação da base de dados. Assim cada variável pode ter como valor qualquer n-tuplo da relação à qual está ligada.

Uma expressão geral de cálculo relacional sobre n-tuplos, toma a forma:

$$\{ t_1.A_1, t_2.A_2, ..., t_n.A_n \mid COND(t_1, t_2, ..., t_n, t_{n+1}, t_{n+2}, ..., t_{n+m}) \}$$

em que $t_1, t_2, ..., t_n, t_{n+1}, t_{n+2}, ..., t_{n+m}$ são variáveis n-tuplo não necessariamente distintas. Cada A_i é um atributo da relação que constitui a gama de t_i e COND é uma expressão condicional também composta por átomos. Um átomo da forma $R(t_i)$, estabelece que a relação R vai constituir a gama de valores de t_i. À semelhança do cálculo relacional sobre domínios, a outra hipótese para um átomo é ser da forma $t_i.A$ *operador* $t_j.B$ ou $t_i.A$ *operador c*, em que *operador* pertence ao conjunto dos operadores relacionais e c é um valor constante.

Quantificadores

É também possível a utilização nas fórmulas de cálculo relacional de dois símbolos especiais denominados quantificadores: o quantificador universal (∀) e o quantificador existencial (∃). Numa fórmula, uma variável diz-se ligada se estiver quantificada por um dos dois símbolos referidos, aparecendo (∀ t) ou (∃ t) associado a essa fórmula. Caso contrário diz-se que a variável está livre.

A definição de uma fórmula quantificada e do seu valor lógico é:

- Se F for uma fórmula, também o é (∃ t)(F) ou (∀ t)(F), em que t é uma variável;
- A fórmula (∃ t)(F) é verdadeira se a fórmula F for verdadeira para pelo menos um valor de entre a gama de valores possíveis da variável t, caso contrário é falsa;
- A fórmula (∀ t)(F) é verdadeira se a fórmula F for verdadeira para todos os valores do universo de valores possíveis da variável t, caso contrário é falsa.

2.4 Limitações do poder expressivo de linguagens relacionais

A maioria das linguagens de interrogação usadas em sistemas relacionais são inspiradas no cálculo relacional. Isto porque permite expressar a informação de um modo não procedimental. No entanto, a implementação dos operadores definidos nos sistemas é efectuada com base na álgebra relacional. Assim, o papel de um compilador de interrogações é transformar expressões baseadas em cálculo relacional para o correspondente em álgebra relacional. Esta transformação é efectuada pelo mecanismo de optimização de interrogações, que assim tem a tarefa de escolher de entre as muitas interrogações em álgebra relacional equivalentes à expressão de cálculo relacional, aquela que tem menor custo, ou seja, que use o esquema físico da melhor maneira.

O poder das linguagens relacionais, não é contudo, suficiente para expressar todas as manipulações possíveis. Um tipo de operações que em geral não pode ser especificada na álgebra relacional é o fecho transitivo de uma relação binária. Esta operação é aplicada a uma associação recursiva entre n-tuplos da mesma relação.

Exemplo 2.7: uma operação recursiva sobre uma relação VOO(Aeroporto_origem, Aeroporto_destino), que indica a existência de voos de um para outro aeroporto, poderá ser a interrogação à base de dados de quais os aeroportos para onde é possível voar a partir de um dado aeroporto de origem e considerando todos os níveis possíveis, ou seja, qualquer número possível de aeroportos intermédios.

Com a álgebra relacional ficamos limitados a conseguir saber quantos aeroportos são atingíveis até um determinado nível pré-estabelecido a partir do aeroporto original. Para tal, achamos os aeroportos atingíveis para cada nível e depois usamos o operador de união para achar o resultado final. No problema proposto, como o número de níveis era indeterminado, seria necessário o uso de um mecanismo de iteração não disponível no modelo relacional para conseguir saber o pretendido. □

2.5 SQL - Linguagem para Bases de Dados Relacionais

Originalmente denominada SEQUEL[2] [1] e usada no *System R*, uma base de dados relacional experimental desenvolvida nos laboratórios da IBM, SQL[3] constitui a linguagem de interface para

[2] *Structured English QUEry Language*
[3] *Structured Query Language*

sistemas de bases de dados relacionais mais usada actualmente. É a linguagem usada no sistema relacional DB2 da IBM assim como em muitos outros SGBD's relacionais existentes.

Concebida em meados dos anos 70, teve a sua primeira normalização pela ANSI[4] e pela ISO[5] em 1986 [27] sofrendo a norma a sua primeira actualização em 1989 [28]. Em 1992, a norma é objecto de uma revisão profunda, e embora tentando manter o maior nível de compatibilidade com a versão anterior, revela diferenças que introduzem algumas incompatibilidades pontuais.

O standard SQL, é adoptado com alterações de implementação por numerosos produtos comerciais, surgindo assim variações do dialecto normalizado que em certos aspectos vão além da norma, não suportando também frequentemente algumas partes das normas. Este tipo de SGBD's domina o mercado, havendo uma crescente massificação no uso da linguagem quer em sistemas de médio e grande porte para múltiplos utilizadores, quer em plataformas pessoais para um utilizador simples, passando pelas soluções intermédias.

Em SQL, o modelo relacional é encarado de um modo informal, *estando muito longe de constituir uma reconstituição fiel deste modelo* [18]. A terminologia usada na linguagem, difere ela própria da adoptada no modelo relacional. Na figura seguinte é possível comparar diferentes termos usados nos dois casos para conceitos semelhantes.

Formal - Modelo Relacional	**Informal - SQL**
Esquema de Relação	Descrição da Tabela
Relação	Tabela
N-Tuplo	Linha
Atributo	Coluna
Domínio	Tipo de Dados

Figura 2.1: Diferença de terminologia entre o Modelo Relacional e SQL

A linguagem SQL é orientada aos conjuntos [43]. Significa isto que está vocacionada para que as suas instruções manipulem várias linhas de uma tabela simultaneamente, podendo na mesma expressão ser manipulada toda uma tabela. A sua coexistência com linguagens de 3ª geração como COBOL ou C, que manipulam dados instância a instância, faz-se adoptando o uso de cursores que permitem a navegação linha a linha na tabela.

Nos pontos a seguir, aborda-se o standard actual conhecido informalmente como "SQL-92" ou "SQL-2"[29].

2.5.1 Definição de Dados em SQL

O subconjunto de instruções para definição de dados, da linguagem SQL, tem como objectivos a definição do esquema da base de dados, sua alteração ou remoção de partes.

[4] *American National Standards Institute*
[5] *International Standards Organization*

Uma das possibilidades oferecidas por este subconjunto de instruções é a definição de domínios pelo utilizador. Assim, temos as seguintes expressões usadas para definição, modificação e remoção de um domínio por parte de utilizadores:

- `CREATE DOMAIN`
- `ALTER DOMAIN`
- `DROP DOMAIN`

Exemplo 2.8: uma forma de definir o domínio *altura* seria:

```
CREATE DOMAIN altura AS INTEGER
  CONSTRAINT altura_maior_que_zero CHECK (VALUE >= 0)
```

em que é definida uma restrição que estabelece que os valores possíveis para variáveis do tipo altura não podem ser menores do que zero. □

Na realidade, esta funcionalidade traduz mais uma abreviação sintáctica do que a possibilidade de definição de verdadeiros domínios, pois, por exemplo, não é possível a definição de novos domínios usando como base domínios já existentes (um novo domínio tem de ser expresso usando sempre tipos de dados do sistema como base). Não é possível, igualmente, a definição de operadores aplicáveis aos novos domínios criados pelos utilizadores.

Para a criação, alteração e remoção de tabelas da base de dados, temos:

- `CREATE TABLE`
- `ALTER TABLE`
- `DROP TABLE`

Exemplo 2.9: a criação da tabela PRATELEIRA correspondente à relação com o mesmo nome definida no exemplo 4 poderia ser:

```
CREATE TABLE PRATELEIRA (
  Numero          INTEGER
       CONSTRAINT numero_nao_nulo NOT NULL,
  Altura          altura,
  NumEstante      INTEGER,
  CONSTRAINT chave_primaria_prateleira
       PRIMARY KEY (Numero),
  CONSTRAINT chave_externa
       FOREIGN KEY (NumEstante)
       REFERENCES ESTANTE(Numero)
);
```

em que é definida a chave primária da tabela assim como uma chave externa. Nesta última é indicada qual a tabela e qual a chave primária dentro desta a que se refere a chave externa definida na tabela PRATELEIRA. □

Tabelas Base, Derivadas e Vistas

Uma tabela base tem a informação que a compõe armazenada ou em suporte persistente ou em memória. Se a tabela está neste último caso, designa-se por tabela base temporária e o seu conteúdo persiste ao longo de uma sessão; se for global é acessível de todos os módulos; se for local é manipulável apenas dentro desse módulo.

No caso de tabelas derivadas, o seu conteúdo é obtido directa ou indirectamente a partir de uma ou mais tabelas base, através de uma expressão de interrogação à base de dados. Uma vista da base de dados é uma espécie de tabela virtual que tem uma definição expressa através do comando `CREATE VIEW`. Uma tabela derivada pode ou não ser actualizável. Se sim, porque por exemplo a vista assenta numa única tabela base e não contém agregações, podem efectuar-se operações de inserção, remoção e alteração da respectiva tabela; se não a tabela está disponível apenas para leitura.

Catálogos, Esquemas e *INFORMATION_SCHEMA*

Conjuntos de tabelas, domínios e outros objectos, estão agrupados em esquemas. Cada tabela tem de estar dependente hierarquicamente de um dado esquema que a qualifica sintacticamente. Com esta noção é possível um utilizador criar e gerir mais do que um esquema, deixando de haver um mapeamento obrigatório de um para um entre esquema e identificação para autorização (*authID*) que acontecia nas versões anteriores da norma de SQL. Para afastar o perigo da duplicação de nomes de esquemas definidos por dois utilizadores, está ainda definido o conceito de catálogo. Um catálogo engloba um conjunto de esquemas. Assim o caminho completo para se referir a uma tabela da base de dados (ou outro objecto) vem:

<nome do catálogo>.<nome do esquema>.<nome da tabela>

O uso de uma estrutura organizativa deste género, explica-se também devido ao crescente uso e importância das bases de dados distribuídas com os problemas de gestão que isso acarreta [21].

Em cada catálogo existe um esquema denominado `INFORMATION_SCHEMA` que contém meta-informação sobre a base de dados. Consiste num conjunto de tabelas cujo conteúdo reflecte todas as definições existentes nos outros esquemas. É definido como contendo um conjunto de vistas de um esquema de definições.

O resultado de uma interrogação a este dicionário de dados, depende do utilizador que a executa. Assim, por exemplo, uma interrogação para retornar todo o conteúdo da tabela `SCHEMATA` do `INFORMATION_SCHEMA` irá retirar do esquema de definições apenas as linhas que correspondem a esquemas geridos por aquele utilizador. Evita-se deste modo que um utilizador consiga identificar quais os esquemas que existem naquele catálogo.

2.5.2 Manipulação de Dados

As principais instruções para manipulação de dados em SQL são: SELECT, INSERT, UPDATE e DELETE. Estas podem ser divididas entre operações para consulta de informação da base de dados, que se executam com a instrução:

- `SELECT`, que selecciona a informação da base de dados que satisfaz os critérios definidos na expressão de "interrogação" à base de dados.

e operações de actualização da base de dados que são efectuadas usando as instruções:

- `INSERT`, que acrescenta nova informação à base de dados;
- `UPDATE`, que permite alterar a informação existente na base de dados;
- `DELETE`, para remoção de informação.

Interrogações

Uma expressão de selecção de informação pode ser composta por várias cláusulas:

```
SELECT [ ALL | DISTINCT ] lista-de-itens
FROM lista-de-tabelas
WHERE expressão-condicional
GROUP BY lista-de-colunas
HAVING expressão-condicional
```

onde as últimas três cláusulas são de carácter opcional.

Descreve-se resumidamente cada um dos componentes da expressão:

- `SELECT`, indica-se a seguir o nome das colunas das tabelas alvo da interrogação que se pretende apareçam na tabela resultado. Tem semelhanças com a operação de projecção definida na álgebra relacional. Aqui podem também ser incluídas funções de agregação como somas, contagem de valores, valor mínimo, valor máximo ou médias aritméticas;
- `FROM`, seguido por uma lista das tabelas alvo da interrogação. O resultado da avaliação desta cláusula vai ser o produto cartesiano implícito das tabelas que compõem a lista;
- `WHERE`, onde a seguir se indicam as condições de selecção de informação. Da tabela obtida na cláusula FROM, serão eliminadas todas as linhas cuja avaliação, usando as condições de selecção, não é verdadeira. É equivalente à operação de selecção da álgebra relacional;
- `GROUP BY`, seguida por uma lista de colunas. Esta cláusula produz como resultado um conjunto de grupos de linhas a partir da tabela obtida na cláusula WHERE. É criado um número mínimo de grupos de linhas tal que em cada grupo todas as linhas tenham o mesmo valor para a lista de colunas que é passada como argumento de GROUP BY. Apesar de o resultado não ser uma tabela, esse facto é contornado pela obrigatoriedade de que cada coluna referida na cláusula SELECT deverá estar referida também na lista passada em GROUP BY. As excepções verificam-se quando as colunas são usadas na expressão SELECT como argumentos de funções de agregação. Deste modo, garante-se que o resultado da cláusula de agrupamento será sempre uma relação, caso isto não aconteça, o SQL deverá dar erro;
- `HAVING`, a expressão condicional que se segue a HAVING é de estrutura semelhante à da cláusula de WHERE. Esta cláusula funciona como filtro das linhas da tabela resultante da partição definida por GROUP BY, e consequentes valores agregados. A cláusula WHERE filtra linhas da tabela original.

As três primeiras expressões seguintes respeitam às interrogações expressas em álgebra relacional no ponto 2.2.9. Pode-se desta forma comparar a mesma expressão de interrogação em SQL e álgebra relacional:

Interrogação 1: Titulo de todas as Unidades de Descrição que não estão na Unidade de Instalação que tem como referência "ref1".

```
SELECT Titulo
FROM UDescrição
WHERE CodReferência IN
( SELECT CodReferencia
  FROM UDescrição
  EXCEPT
  SELECT CodUD
  FROM Arrumacao_UD
  WHERE RefUI = 'ref1');
```

A operação de diferença é implementada usando a palavra reservada `EXCEPT`. No entanto, embora esteja assim definido no standard SQL/92, poderão ser usadas outras palavras reservadas para a mesma função nas diversas bases de dados relacionais do mercado[6]. Os argumentos de `EXCEPT` são as duas tabelas obtidas com as respectivas expressões de `SELECT`. À tabela obtida da primeira expressão dentro de parênteses vão ser retiradas as linhas que também aparecem na tabela resultante da expressão de `SELECT` que vem a seguir à instrução `EXCEPT`. Da tabela resultante desta operação, vai ser retirada apenas a coluna com o Titulo através da primeira expressão de `SELECT`.

Interrogação 2: Titulo de todas as Unidades de Descrição que estão na Unidade de Instalação cuja referência é "ref1" e na Unidade de Instalação cuja referência é "ref2".

```
SELECT Titulo
FROM UDescrição
WHERE CodReferência IN
( SELECT CodUD
  FROM Arrumacao_UD
  WHERE RefUI = 'ref1'
  INTERSECT
  SELECT CodUD
  FROM Arrumacao_UD
 WHERE RefUI = 'ref2');
```

Neste caso a operação de intersecção da álgebra relacional é implementada usando a palavra reservada `INTERSECT`.

Interrogação 3: Datas de produção das Unidades de Descrição que estão em todas as Unidades de Instalação da prateleira 26.

```
CREATE VIEW tmp AS
 (SELECT DISTINCT CodUD, I.Referencia AS RefUI
```

[6] no SGBD relacional da Oracle a palavra reservada para diferença é `MINUS`.

```
    FROM Arrumaçao_UD, Uinstalação AS I
    WHERE (NumPrateleira = 26)
    EXCEPT
    SELECT CodUD, RefUI
    FROM Arrumaçao_UD);

SELECT Dataprodução
FROM UDescrição
WHERE CodReferência IN
  (SELECT CodUD
   FROM Arrumaçao_UD
   EXCEPT
   SELECT CodUD
   FROM tmp)
```

De notar, na interrogação 3, que o operador de quociente da álgebra relacional é substituído por uma expressão equivalente, usando os operadores de projecção e de diferença.

Interrogação 4: É ainda apresentada uma quarta interrogação com o intuito de demonstrar uma expressão de selecção mais complexa.

```
SELECT UD.Titulo, UD.Notas, UD.QuantidadeSobCustodia,
UD.DataProduçao, COUNT( A.RefUI )
FROM UDescrição AS UD, Arrumaçao_UD AS A, Uinstalação AS I
WHERE (UD.DataProduçao > '1-1-1900' )
  AND (A.CodUD = UD.CodReferencia )
  AND (A.RefUI = I.Referencia )
GROUP BY UD.Titulo, UD.Notas, UD.QuantidadeSobCustodia,
UD.DataProduçao
HAVING SUM( I.ExtensãoLinear ) > 200 ;
```

Na interrogação anterior, o resultado será composto pelas colunas Titulo, Notas, QuantidadeSobCustodia, DataProduçao e o número de Unidades de Instalação (usando a função de agregação COUNT) que albergam essa Unidade de Descrição. São ainda estabelecidas mais duas condições de selecção. Serão seleccionadas as Unidades de Descrição cuja data de produção seja posterior a 1-1-1990 e cujo somatório das extensões lineares das Unidades de Instalação onde estão armazenadas cada uma das U.D.'s seja superior a 200 centímetros.

Nesta interrogação, demonstra-se a aplicação de uma operação de junção entre as tabelas nas duas últimas condições da cláusula WHERE.

Operações entre múltiplas tabelas

Um dos mais importantes grupos de operações entre tabelas é o constituído pelas várias operações de junção possíveis. Além desse, outras operações permitidas em SQL, que envolvem várias tabelas como operandos são a união, intersecção, e diferença (ver ponto: álgebra relacional) correspondendo às instruções SQL, `UNION`, `INTERSECT` e `EXCEPT`.

Relembra-se que, quando numa instrução de interrogação, a cláusula `FROM` é composta por uma lista com mais do que uma tabela, então a avaliação dessa cláusula vai ser o produto cartesiano implícito das tabelas referidas. Essa tabela resultante servirá de base para o resultado obtido da interrogação. É então possível, através de operações de selecção com a cláusula `WHERE` e operações de projecção com a cláusula `SELECT`, sintetizar operações de junção[7] como na interrogação vista no ponto anterior.

Exemplo 2.10: neste exemplo temos uma operação de equi-junção entre as tabelas `Prateleira` e `Estante`, sendo excluída da tabela resultante a coluna `NumEstante` da relação `Prateleira` que é chave externa da coluna `Numero` da relação `Estante`:

```
SELECT Estante.Numero, Estante.Comprimento, Prateleira.Numero,
Prateleira.Altura
FROM Estante, Prateleira
WHERE Estante.Numero = Prateleira.NumEstante
```

SQL/92 expande, relativamente a versões anteriores da norma, o conjunto de recursos para a realização deste tipo de operações, permitindo de um modo claro e automático a realização de operações de junção interna e externa através da introdução de uma variante na cláusula `FROM` das expressões de interrogação.

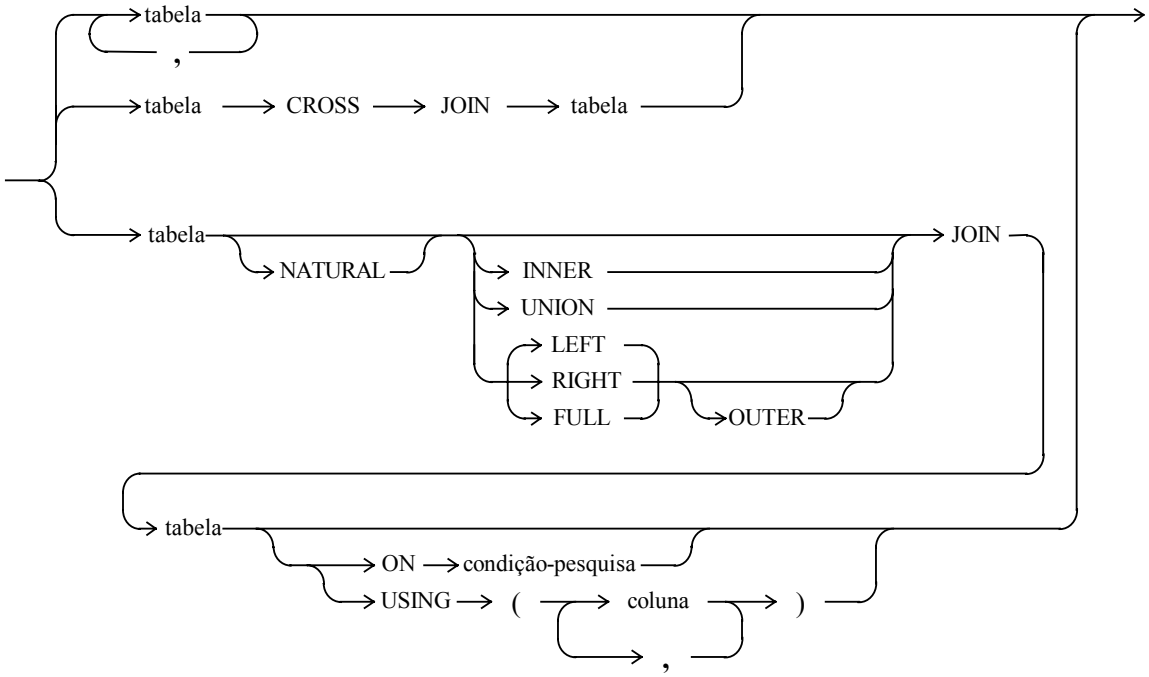


Figura 2.2: Sintaxe de operações de junção em SQL-92

A sintaxe completa para este tipo de operações está expressa na figura acima.

Exemplo 2.11: a interrogação do exemplo anterior, usando este modo de realização de junções ficaria:

```
SELECT *
```

[7] Excepto operações de junção externa que necessitam de uma notação específica.

```
FROM Estante JOIN Prateleira
ON Estante.Numero = Prateleira.NumEstante
```

Actualizações

As operações de actualização de informação da base de dados podem ser de três tipos:

- Inserção de informação usando a instrução INSERT:

exemplo de inserção de uma nova linha na tabela UDescrição:

```
INSERT
INTO UDescrição ( CodReferencia, Titulo, Notas, Quantidade,
QuantidadeSobCustodia, dataProduçao )
VALUES ( 'ud1', 'Unidade de descriçao 1', 'Notas 1', ' 10
filmes', '7 filmes', '1-1-1970');
```

exemplo de inserção de múltiplas linhas na tabela Arrumaçao_UD

```
INSERT
INTO Arrumaçao_UD ( CodUD, RefUI )
  SELECT UDescrição.CodReferência, Uinstalação.Referencia
  FROM UDescrição, Uinstalação
  WHERE ( UDescrição.DataProduçao > '1-1-1900' )
  AND ( UDescrição.DataProduçao < '1-1-1905' )
  AND ( Uinstalação.Referencia = 'ui 1');
```

- Actualização usando a instrução UPDATE:

exemplo de actualização da coluna Altura da tabela Prateleira:

```
UPDATE Prateleira
SET Altura = Altura + 20
WHERE NumEstante = 5;
```

- Eliminação usando a instrução DELETE:

exemplo de remoção de informação da tabela UDescrição:

```
DELETE
FROM UDescrição
WHERE CodReferência =
  ( SELECT Arrumaçao_UD.CodUD
     FROM Arrumaçao_UD
     WHERE Arrumaçao_UD.RefUI = 'ref1' ) ;
```

2.5.3 Uso do SQL em aplicações

Estão contemplados três modos de utilização do SQL/92 SQL. No mais simplificado, as instruções são executadas interactivamente através de invocação directa, sendo o resultado enviado para um dispositivo de saída como um terminal, PC, ficheiro ou impressora.

No entanto, o ambiente de execução típico é composto por uma mistura de SQL e uma linguagem hospedeira de 3ª ou 4ª geração. Assim, a segunda possibilidade é embutir as expressões SQL no código fonte de um programa escrito noutra linguagem de programação que constituirá a linguagem hospedeira (*embedded SQL*).

Outro paradigma possível para uso de SQL e uma linguagem de programação, consiste no uso de módulos SQL (*module language*). A aplicação é composta por programas escritos na linguagem de programação seleccionada e por um ou mais módulos de SQL escritos separadamente que são invocados pelos programas mencionados. As vantagens deste tipo de abordagem relativamente ao SQL embutido são: a clara distinção na aplicação entre os paradigmas de programação utilizados; e a possibilidade de migração de linguagem de programação utilizada mantendo os módulos SQL desenvolvidos.

De notar que se uma expressão SQL puder ser executada interactivamente, então também é possível a sua execução no programa de aplicação, embora o contrário possa não suceder. Exemplo disso são as expressões de interrogação à base de dados. Quando estas são executadas embutidas numa linguagem de programação o seu conjunto resultado necessita ser processado para que possa ser utilizado no respectivo programa.

Desadaptação de impedâncias

Existem diferenças no modo como as linguagens de programação tradicionais e SQL procedem no tratamento da informação. SQL é uma linguagem para tratamento de conjuntos. A maioria das instruções SQL operam num número arbitrário de linhas de uma tabela simultaneamente. Linguagens como C, Fortran, COBOL ou Pascal, não apresentam essa característica, podendo apenas lidar com um número reduzido, tipicamente um registo de itens em cada operação. Logo requerem estruturas cíclicas de controlo quando estão a processar ficheiros. Este facto origina a chamada *desadaptação de impedâncias* entre as linguagens e SQL. O mecanismo adoptado para ultrapassar este facto é a definição de cursores que permitem navegar ao longo de um conjunto, possibilitando o processamento um a um dos elementos do conjunto associado ao cursor em causa.

Outro tipo de desadaptação de impedâncias resulta do facto do universo de tipos de dados suportados por SQL e as linguagens tradicionais não terem um mapeamento biunívoco. O standard SQL/92 disponibiliza a expressão CAST para minorar o problema referido. Este mecanismo permite a conversão explícita de dados de um tipo para outro. É possível, por exemplo, a conversão de uma data em SQL para uma cadeia de caracteres.

SQL embutido

Das soluções apresentadas, a mais usada no desenvolvimento de aplicações para sistemas de bases de dados relacionais SQL é o seu embutimento no código fonte de uma dada linguagem. Neste caso, a técnica geralmente usada é a disponibilização de um pré-processador para a linguagem hospedeira, que separa o código SQL do código da linguagem de programação. Assim, enquanto

o primeiro é passado ao SGBD para ser analisado e executado, o segundo é compilado normalmente.

Apresentam-se a seguir alguns detalhes respeitantes ao uso de SQL embutido:

- Instruções SQL embutidas são delimitadas pelo prefixo `EXEC SQL` e por um símbolo de terminação para fácil distinção das expressões da linguagem hospedeira;
- Todas as variáveis da linguagem hospedeira que são referidas pelo SQL, devem ser definidas dentro de uma expressão declarativa SQL do tipo: `DECLARE SECTION`;
- É permitido que uma instrução executável de SQL possa aparecer em qualquer situação onde seja também possível aparecer uma instrução da linguagem hospedeira. De notar que é possível a distinção entre instruções de carácter declarativo e de carácter executável;
- Instruções SQL podem conter referências para variáveis da linguagem hospedeira. O nome da variável é prefixado com o símbolo ":" para distinção das variáveis SQL;
- Para ser possível a passagem do resultado de uma expressão de `SELECT` em SQL para uma variável na linguagem hospedeira, é usada a cláusula `INTO` entre as sub-expressões `SELECT` e `FROM`. Outro modo de passar a informação da base de dados para a variável hospedeira é através da instrução `FETCH`;
- A variável especial de SQL - `SQLSTATE`, acessível para leitura na aplicação, é usada para devolver um código de erro da última operação SQL executada. Esta variável é representada como um vector de cinco caracteres. Após a execução de cada instrução SQL, deve ser executado um teste ao seu valor. Esse teste pode ser efectuado de um modo implícito usando a expressão declarativa `WHENEVER` juntamente com a condição e a acção a efectuar se o código de retornado em `SQLSTATE` coincidir com essa condição. Uma condição possível para `WHENEVER` é `NOT FOUND` que corresponde ao valor 02000 da variável `SQLSTATE`. Quando não acontece nenhuma situação de erro `SQLSTATE` devolve o valor 00000.

Uma das vantagens da junção de uma linguagem de programação a SQL, é ultrapassar a limitação do poder de processamento do SQL como, por exemplo, no cálculo do fecho transitivo de uma relação binária representando um grafo (ver exemplo 2.7). Outro ponto fraco do SQL prende-se com a limitação no tratamento da interface com o utilizador, podendo este ser ultrapassado com recurso a linguagens de 4ª geração.

Exemplo 2.12: este exemplo (retirado de [20] pp. 36) demonstra como é possível através do uso de SQL embutido o cálculo do fecho transitivo da relação Voo do exemplo 2.7:

```
exec sql begin declare section
int tamanho; /* variável C conhecida do sql */
exec sql begin declare section
int ult_tam;      /* variável C não conhecida do sql */
```

```
exec sql execute immediate
            create table Viagens (
                        origem char(20),
                        destino char(20))
exec sql execute immediate
insert into Viagens
            select *
            from Voo

exec sql prepare uma_iteração from
insert into Viagens
            select Viagens.origem, Voo.Aeroporto_destino
            from Viagens, Voo
            where Viagens.destino = Voo.Aeroporto_origem

exec sql prepare calc_tam from
            select count (distinct *)
            from Viagens

tamanho = 0
ult_tam = 1;
while (tamanho != utl_tam) {
            ult_tam = tamanho;
            exec sql execute uma_iteração;
            exec sql declare cursor1 cursor for calc_tam;
            exec open cursor1;
            /* o resultado contém apenas um n-tuplo */
            exec sql fetch cursor1 into :tamanho;
            exec sql close cursor1;
}
```

No código anterior, é possível observar a declaração de um cursor com o nome `cursor1`, usando uma instrução não executável de SQL (`declare ... cursor ...`), que especifica que a tabela `calc_tam` vai poder ser percorrida elemento a elemento de modo a ser possível aceder ao elemento através de uma operação de `fetch` (embora neste caso específico `calc_tam` seja composta por apenas um elemento...). □

A sequência de operações para utilização de um cursor vem:

- Declaração do cursor, em que é definida a tabela que vai ser associada ao cursor com declare;
- Operação de activação do cursor através da instrução `open`;
- Ciclo no qual vão ser executadas operações de consulta, alteração ou remoção da linha da tabela apontada pelo cursor com as instruções `fetch`, `update` e `delete`. A navegação

na tabela é efectuada sequencialmente ou, opcionalmente, usando mecanismos, para posicionamento do cursor, mais elaborados;

- Desactivação do cursor com `close.`

Algumas operações de SQL embutido não necessitam de cursores para serem realizadas como as operações de:

- Selecção desde que apenas retornem zero ou uma linhas;
- Inserção;
- Actualização e remoção em que não é necessária a linha actualmente apontada pelo cursor.

SQL dinâmico

A filosofia subjacente à existência de SQL dinâmico é a possibilidade de durante a execução da aplicação permitir ao utilizador a formulação de uma instrução SQL. Por exemplo, através de uma interface mais amigável proporcionada pela aplicação, o utilizador poderá formular uma interrogação à base de dados que será processada de modo semelhante ao modo de invocação directa.

Ao contrário do SQL estático, no SQL dinâmico não é possível haver nenhum tipo de processamento como pré-compilação e várias fases de compilação e optimização das instruções digitadas antes da execução da aplicação[8].

2.6 Dependências Funcionais e Normalização

Nesta secção é abordado o conceito de dependência funcional, importante no projecto de um esquema de base de dados. Uma dependência funcional é uma associação muitos-para-um entre atributos de uma relação, que está implícita na semântica do sistema que a base de dados pretende modelizar. A escolha de uma estrutura lógica adequada está directamente relacionada com a correcta identificação e tratamento das dependências funcionais existentes. Um mapeamento deficiente do sistema real para o modelo lógico pode levar à existência de informação redundante, anomalias de actualização, perda de informação e aparecimento de valores nulos. Existem por isso mecanismos de decomposição do esquema lógico de modo a evitar as consequências atrás descritas, tentando manter a riqueza semântica captada no modelo. Essas operações são designadas por normalização da base de dados.

Podem assim ser consideradas quatro métricas informais para a qualidade de um projecto de esquema relacional [21]:

- Semântica dos atributos: deve ser fácil a interpretação da semântica da relação obtida, evitando-se a mistura de atributos de múltiplas entidades e associações na mesma relação;

[8] Em SQL dinâmico, essas operações têm de ser realizadas em tempo de execução, reduzindo vulgarmente a rapidez de execução.

- Redução dos valores redundantes nos n-tuplos: minimizando não só o espaço ocupado pela informação como também evitando as anomalias de actualização. Estas podem ser quer de inserção, de apagamento ou de alteração;
- Redução dos valores nulos nos n-tuplos: numa relação com excesso de atributos, poderá haver atributos que não se aplicam a determinados n-tuplos, originando o aparecimento de valores nulos. Estes, além de ocuparem espaço desnecessariamente, poderão levar a interpretações incorrectas do seu significado e problemas com operações de junção e agregação;
- Não permissão de associações espúrias: más decomposições ou decomposições com perdas de uma relação num conjunto de relações que se desejam equivalentes à primeira, pode levar a perda de informação devido a não ser possível obter a relação original através do uso de operações de junção.

2.6.1 Definições

O conceito de dependência funcional está ligado à possibilidade de numa relação, um seu subconjunto de atributos ter uma relação de dependência com outro subconjunto de atributos da mesma relação. Essa dependência pode ser restrita ao conjunto de n-tuplos que compõe uma dada instância de uma relação ou ser estendida ao conjunto de todos os valores possíveis para os n-tuplos da relação, constituindo uma dependência funcional intemporal.

Assim para o segundo caso referido, pode-se estabelecer o conceito de *dependência funcional* numa relação R, com X e Y como subconjuntos arbitrários do conjunto de atributos de R. Diz-se então que Y é funcionalmente dependente de X, usando a simbologia $X \rightarrow Y$, se e só se para todos os valores possíveis para R, cada valor de X tem associado a si precisamente um valor de Y.

Outra interpretação será, sempre que dois n-tuplos tenham o mesmo valor para o subconjunto de atributos X, então o valor de Y terá de ser o mesmo em ambos os n-tuplos.

Na expressão acima X é *determinante* ou determina funcionalmente Y, e Y é funcionalmente dependente de X ou também designado por termo dependente da expressão.

Esta definição para dependência funcional, pode ser encarada como uma restrição de integridade para a relação em questão. De facto, coloca limites nos valores que os n-tuplos da relação podem assumir.

É também próprio afirmar que se X é uma chave candidata de R, em particular se for chave primária, então todos os atributos Y da relação R são necessariamente funcionalmente dependentes de X (indo assim ao encontro da definição de chave candidata).

Tem-se assim, por exemplo para a relação *UInstalação*:

$$Referência \rightarrow \{ Referência, Tipo, ExtensãoLinear, NumPrateleira \}$$

Na realidade, se uma relação R satisfaz a dependência funcional $A \rightarrow B$ e A não é chave candidata, então R terá algum grau de redundância, isto para os casos em que a dependência funcional não é trivial (ver no ponto seguinte) e A não é super-chave.

Um dos problemas enfrentados reside na tentativa de conseguir transformar o potencialmente elevado número de dependências funcionais de uma relação, para um número mais reduzido que seja mais facilmente tratável. Isto prende-se com o facto de ser mais fácil para o SGBD garantir o

cumprimento das restrições de integridade impostas pelas dependências funcionais, se em vez de lidar com um conjunto S, lidar com um conjunto de dependências funcionais T menor do que S e que garanta que todas as dependências funcionais de S sejam implicadas pelas dependências funcionais de T.

2.6.2 Dependências Triviais e Não Triviais

Um passo para a redução do tamanho do conjunto de dependências funcionais é a eliminação das dependências triviais. Uma dependência funcional é trivial se, numa relação, for impossível não a satisfazer. Para *UInstalação*, tem-se por exemplo a seguinte dependência funcional trivial:

$$\{ Tipo, ExtensãoLinear, NumPrateleira \} \rightarrow Tipo$$

Uma dependência funcional diz-se trivial se e só se o lado direito da sua expressão for um subconjunto do lado esquerdo da expressão.

Para o estudo das restrições de integridade a respeitar, interessa o estudo das dependências funcionais *não triviais*. De um ponto de vista formal, contudo, é necessário a assunção de dependências funcionais para além das não triviais.

2.6.3 Regras de Inferência para Dependências Funcionais

Para um esquema de relação R, é possível definir um conjunto S das dependências funcionais imediatas ou aquelas que são *semanticamente obvias*. Existem no entanto muitas outras dependências funcionais implícitas em S. Ao conjunto de todas as dependências funcionais inferidas de S designa-se por fecho de S, sendo denotado por S^+.

Para se conseguir determinar S^+ a partir de S de um modo sistemático, adoptam-se regras de inferência. Nas regras a seguir mencionadas, A, B e C são subconjuntos arbitrários do conjunto de atributos de uma dada relação R. Usou-se a notação AB para denotar a união dos subconjuntos A e B.

As regras de inferência de Armstrong[9] são:

- Reflexividade: Se B é subconjunto de A, então $A \rightarrow B$;
- Aumento: Se $A \rightarrow B$, então $AC \rightarrow BC$;
- Transitividade: Se $A \rightarrow B$ e $B \rightarrow C$, então $A \rightarrow C$.

outras regras derivadas das regras acima são:

- União: Se $A \rightarrow B$ e $A \rightarrow C$, então $A \rightarrow BC$;
- Decomposição: Se $A \rightarrow BC$, então $A \rightarrow B$ e $A \rightarrow C$;
- Composição: Se $A \rightarrow B$ e $C \rightarrow D$, então $AC \rightarrow BD$;
- Pseudo-transitividade: Se $A \rightarrow B$ e $CB \rightarrow D$, então $AC \rightarrow D$.

[9] Este conjunto de regras é completo, na medida em que partindo de um conjunto de dependências funcionas S, o conjunto de todas as dependências funcionais inferidas usando as regras referidas, constitui sempre S^+ o fecho de S.

2.6.4 Normalização

No processo de normalização, submete-se um esquema de relação a um conjunto de testes com vista a determinar qual a forma normal em que se encontra. Três formas normais foram propostas inicialmente por Codd, tendo sido posteriormente proposta uma definição mais robusta para a 3ª forma normal a que se chamou formal normal de Boyce-Codd. A primeira, segunda e terceira formas normais assim como a de Boyce-Codd, são baseadas nas dependências funcionais entre atributos de uma relação. Mais tarde foram propostas duas outras formas normais, a quarta baseada em dependências multivocas e a quinta em dependências de junção.

Quanto maior for a forma normal em que o esquema se encontra, melhor será a qualidade do mesmo. Se um esquema está numa dada forma normal, então também está nas formas normais mais baixas. Pode-se pois, ver o processo de normalização como um processo de camadas em que a base é a primeira forma normal e o topo a quinta.

Primeira forma normal

Uma relação diz-se normalizada ou na *primeira forma normal*, quando não contém atributos multivalor, atributos compósitos ou combinação dos dois. Os atributos devem então, ser todos atómicos: simples e indivisíveis. Normalizar uma relação é proceder à sua passagem para uma forma tabular.

A normalização pode ser encarada como um método no qual esquemas de relações não satisfatórios são decompostos através da projecção segundo alguns dos seus atributos num conjunto de relações. Essa decomposição terá de ser efectuada sem perda de informação, sendo possível desse modo recuperar a relação original através de operações de junção das relações derivadas. O processo deve pois ser reversível. A decomposição referida, para ser sem perdas, terá que ser efectuada de modo a não haver eliminação de dependências funcionais entre as suas etapas.

Segunda forma normal

A segunda forma normal é baseada no conceito de dependência total ou seja de irredutibilidade do lado esquerdo da expressão da dependência funcional. Este conceito é oposto ao conceito de dependência parcial visto a seguir.

Dada uma dependência funcional $X \rightarrow Y$, esta diz-se *dependência parcial* se existe algum atributo A que possa ser retirado de X tal que a dependência se mantenha.

Uma relação diz-se na segunda forma normal, se estiver na primeira forma normal e todos os atributos não chave forem não parcialmente dependentes da chave primária. Na segunda forma normal, são eliminadas todas as dependências parciais de atributos não chave para com a chave primária. Garante-se assim que não existe nenhum atributo A fora da chave primária tal que exista a dependência funcional $X \rightarrow A$ e X seja subconjunto restrito da chave.

Terceira forma normal

A terceira forma normal é baseada no conceito de dependência transitiva.

Uma dependência funcional $X \rightarrow Y$ numa relação R é *dependência transitiva* se existe um conjunto de atributos Z que não é subconjunto de nenhuma chave de R, tal que existem as dependências funcionais $X \rightarrow Z$ e $Z \rightarrow Y$.

Uma relação está na terceira forma normal, se e só se estiver na segunda forma normal, e todos os atributos não chave forem dependentes da chave primária de um modo não transitivo. São eliminadas as dependências transitivas da chave.

Forma Normal de Boyce-Codd

No caso geral em que uma relação tem mais do que uma chave candidata tal que as chaves candidatas tenham pelo menos um atributo comum, a terceira forma normal não se adequa satisfatoriamente. Assim surge a forma normal de Boyce-Codd que reforça a terceira forma normal, na medida em que contempla o caso geral referido acima.

Uma relação está na forma normal de Boyce-Codd se e só se os únicos determinantes das dependências funcionais forem chaves candidatas.

Restantes formas normais

A *quarta forma normal* baseia-se no conceito de dependência multívoca. Uma dependência multívoca é uma generalização do conceito de dependência funcional, na medida em que uma dependência funcional é sempre multívoca enquanto o contrário não é necessariamente verdade.

Dada uma relação R com X, Y e Z, três subconjuntos arbitrários de atributos de R. Diz-se que B é multidependente de A, se e só se o conjunto de valores B associados a um par (A,C) depende só de A e não de C. A dependência multívoca entre A e B denota-se por: $A \twoheadrightarrow B$.

Uma relação está na quarta forma normal se todas as suas dependências multívocas forem também dependências funcionais.

A *quinta forma normal* assenta no conceito de dependência de junção.

Uma dependência de junção está ligada à decomposição da relação R nas suas projecções A, B, ..., Z. Se através de uma operação de junção das relações A, B, ..., Z for possível recuperar o conteúdo da relação R, então diz-se que R satisfaz a dependência de junção das relações A, B, ...Z, e denota-se por JD * (A, B, ..., Z) sobre o esquema R.

Uma relação está na quinta forma normal (ou forma normal de projecção-junção) se para todas as dependências não triviais JD * (R_1, R_2, ...,R_n) em F^+, todo R_i é superchave de R, em que F^+ é o fecho de F o conjunto de dependências funcionais, multívocas e dependências de junção de R.

3 Modelo de Dados Orientado por Objectos

Neste capitulo pretende-se abordar as bases de dados orientadas por objectos, uma tecnologia alternativa ao modelo relacional para armazenamento de dados. Estas têm origem e ligações profundas ao paradigma de programação orientada por objectos, alvo de estudo desde há muito [16],[24]. No entanto, só mais recentemente surgiu a motivação para a sua aplicação no armazenamento de dados de forma persistente, com o surgimento de novos contextos de aplicação para os sistemas de armazenamento de dados.

O capítulo inicia-se com uma resenha dos conceitos fundamentais dos Sistemas de Gestão de Bases de Dados Orientados por Objectos (SGBDOO). Segue-se uma análise das características que se esperam presentes num SGBDOO para que possa ser classificado como tal. Na restante parte, é analisado o conjunto de normas propostas pelo ODMG - *Object Data Management Group* para SGBDOOs. Estas, têm um elevado grau de abrangência que vai desde a definição do modelo orientado por objectos, linguagens de definição e interrogação de dados, até à ligação a algumas linguagens orientadas por objectos.

3.1 Características de um SGBD Orientado por Objectos

Nesta secção é feita primeiro uma revisão dos conceitos básicos associados à tecnologia orientada por objectos, tendo como base a linguagem Smalltalk. Passa-se a seguir para um aprofundamento desses conceitos e a um estudo das características que um SGBD Orientado por Objectos deverá ter para ser considerado como tal.

3.1.1 Conceitos básicos

O paradigma de programação orientado por objectos (OO) foi o precursor da utilização de sistemas baseados nos modelos OO nas mais variadas áreas de software.

Embora a noção de objecto seja geralmente atribuída à linguagem de programação Simula [16], a programação orientada por objectos afirmou-se como um novo paradigma de programação com o desenvolvimento da linguagem de programação Smalltalk [24].

Não existe consenso sobre as características a que uma linguagem deve obedecer para que seja considerada "orientada por objectos", havendo diferenças significativas nas várias linguagens OO existentes. Contudo a linguagem Smalltalk é considerada como um bom exemplo de uma linguagem OO pura.

Linguagens como C, Pascal ou FORTRAN, usam um modelo de computação operador/operando, que envolve a utilização separada de procedimentos e dados. Neste modelo, o programador define um conjunto de dados (operandos) e desenvolve um conjunto de operadores (funções ou procedimentos) que têm como parâmetros várias combinações dos operandos. O programador é responsável pela determinação de quais as operações a aplicar aos respectivos dados e verificar que os operandos se adequam aos requisitos de tipos especificados para as operações a serem chamadas. Tipicamente, um nome de um operador identifica um procedimento distinto (pedaço de código), e a implementação dos operadores será diferente para diferentes tipos de operandos.

Numa linguagem orientada por objectos, o modelo exposto é substituído por um modelo objecto/mensagem. No modelo objecto/mensagem, toda a informação é representada sob a forma de *objectos*. Um objecto é uma entidade auto-contida que consiste em:

- Uma parte privada ou estado, consistindo normalmente num conjunto de *variáveis de instância* (em Smalltalk) que podem armazenar valores ou referências para outros objectos;
- Um conjunto de operações (designadas por vezes como o seu *protocolo*). Estas operações constituem a *interface* externa do objecto para o resto do sistema.

Este paradigma incorpora o conceito de tipos de dados abstractos, em que é definida uma parte pública e uma privada para a estrutura de dados ou objecto. Tipos de dados abstractos correspondem nas linguagens orientadas por objectos a *classes*.

As interacções com o objecto ocorrem através de requisições (designadas por *mensagens* em Smalltalk) enviadas conceptualmente para o objecto para que este execute uma das operações da sua interface. Uma computação é definida como uma sequência de interacções entre os objectos do sistema. O envio de mensagens é uma forma indirecta de invocação de procedimentos. Em vez de se nomear o procedimento que deve executar a operação num dado objecto, o objecto, designado por *receptor*, é identificado e a mensagem é-lhe enviada. Na mensagem enviada, define-se um *selector* (nome da operação) que especifica qual a operação a ser executada. Os parâmetros a serem passados, são também especificados na mensagem. Em Smalltalk, o receptor é especificado em primeiro lugar seguindo-se o selector e os parâmetros se existirem.

Um objecto que recebe uma mensagem é responsável (conceptualmente, em tempo de execução) por decidir como responder à mensagem, usando os seus procedimentos privados (designados por *métodos*) para executar a operação requerida. Estes métodos têm a exclusiva capacidade de aceder e manipular o estado do objecto (sendo isto referido como *encapsulamento*). Um método é como uma pequena subrotina que pode exercer computação, e pode também enviar mensagens a outros objectos para que estes possam executar outras operações e devolver resultados.

Em sistemas tipicamente orientados por objectos, os objectos são definidos como membros de um ou mais tipos ou classes. As classes definem a implementação de todos os seus objectos assim como o protocolo a eles associado. Em várias linguagens orientadas por objectos, uma classe de objectos é ela própria representada por um objecto do tipo *classe*. Numa linguagem baseada em classes a noção de que um objecto é uma entidade auto-contida, possuindo as suas próprias operações, é conseguida usando um objecto do tipo classe[10], que armazena entre outras

[10] A classe é ela própria um objecto ou meta-objecto.

coisas, o código para os métodos de todos os objectos dessa classe, de modo a que não haja duplicação em cada objecto individual.

Um utilizador pode definir novos objectos classe, que podem ser usados do mesmo modo que os objectos classe fornecidos com a linguagem. Isto permite que a linguagem seja *estendida* com classes específicas para os requisitos da aplicação. Novos objectos classe (*subclasses*) podem também ser definidos como extensões de objectos classe já existentes através de uma técnica designada por *herança*. A classe original é assim designada como a *superclasse* na nova classe. Se uma classe EMPREGADO for especificada como subclasse de PESSOA, significa que objectos da classe EMPREGADO responderão, não só, às mensagens que podem ser enviadas para objectos de PESSOA, mas também às mensagens específicas para os objectos da sua classe. Uma superclasse pode ter várias subclasses. Em algumas linguagens de programação, uma subclasse pode ter várias superclasses herdando mensagens de cada uma das superclasses. Isto é designado por *herança múltipla*. Uma subclasse pode também ser superclasse de outras classes, podendo a estrutura expandir-se por vários níveis. Esta estrutura designa-se por hierarquia de classes.

O uso de herança nas definições de objectos permite a definição de protocolos de objectos de um modo estruturado. Este factor aliado ao uso de mensagens, possibilita uma forma de *polimorfismo*. Polimorfismo é a capacidade de a mesma mensagem ter diferentes (mas semanticamente semelhantes) respostas, dependendo da classe do objecto receptor. Esta facilidade permite que um programa trate objectos de diferentes classes de um modo consistente. Assim, a mesma mensagem pode ser enviada para um conjunto heterogéneo de objectos e cada objecto irá usar o método respectivo que está definido para essa classe.

As diversas linguagens orientadas por objectos, têm associadas , técnicas de implementação específicas, como ligação dinâmica (ou em tempo de execução) entre as mensagens e o código do método ou gestão automática do armazenamento de objectos (*garbage collection*). Por exemplo, em Smalltalk, os métodos não são chamados directamente pelo nome, mas sim indirectamente em tempo de execução através de uma tabela de expedição (*dispatch table*) associada com a classe do objecto. Cada objecto contém um apontador para o seu objecto classe. Quando é enviada uma mensagem a um objecto em particular, é seguido o apontador de classe desse objecto, e é procurado no objecto classe desse objecto o método correspondente à mensagem. Se o método não for encontrado aí, passa-se para o objecto superclasse na hierarquia até que tal se verifique. Se o método não for encontrado até à classe raiz (classe *Object*), é retornado um erro. Outras linguagens de programação como o C++[55], usam ligação estática (ou tempo de compilação) entre nome da operação e método a ser usado, perdendo alguma flexibilidade em prol de questões de desempenho.

Em linguagens orientadas por objectos, cada objecto tem um identificador único (*object identifier - oid*). Relacionamentos entre um objecto e outro, são representados directamente através do armazenamento do *oid* do segundo objecto numa variável de instância do primeiro.

Na linguagem Smalltalk é usada *garbage collection* para garantir a integridade de apontadores devido à natureza dinâmica da criação e destruição de objectos. Nas linguagens que apresentam esta característica, não existe possibilidade de destruir explicitamente um objecto. Em vez disso, os objectos são "recolhidos como lixo" quando já não estão acessíveis (não são referenciados por nenhum outro objecto). No entanto, outras linguagens como o C++, sacrificam este aumento de segurança por melhor desempenho, permitindo a destruição de objectos explicitamente em vez de *garbage collection*.

As características gerais dos objectos que podem ser definidos numa linguagem OO, como se é baseado em classes ou não, tipos de herança existentes, destruição explícita ou não, é que determinam o modelo OO adoptado pela linguagem.

3.1.2 Arquitectura

O reconhecimento das vantagens, para certas aplicações, de aproximar a tecnologia das bases de dados e a das linguagens OO fez com que diversas abordagens ao problema fossem experimentadas, conforme o ponto de partida de cada um. O modo como o SGBD é implementado e integrado com outros componentes do sistema, o seu conjunto de funcionalidades, desde a mais básica à mais avançada, permite-nos distinguir quatro diferentes arquitecturas OO [11]. O autor de [11] avança mesmo com a designação Sistemas de Gestão de Bases de Dados Objecto (SGBDO) como termo que abarca as arquitecturas referidas a seguir:

- *Sistema baseados num modelo relacional estendido*: estes, representam extensões a sistema de bases de dados, disponibilizando linguagens de interrogação que incorporam procedimentos assim como outras funcionalidades presentes em SGBDOs. Existe ainda uma desadaptação entre a linguagem de programação usada nas aplicações e a linguagem de interrogação. No caso de sistemas relacionais, a extensão visa a introdução de procedimentos, identidade de objectos, hierarquia de tipos assim como outras características de sistemas orientados por objectos, na sua linguagem de interrogação. A popularidade destes sistemas pode residir no atractivo da migração de utilizadores de sistemas relacionais convencionais para este tipo de sistemas. Outra designação adoptada para a extensão de sistemas relacionais com capacidades orientadas por objectos é de Sistemas de Gestão de Bases de Dados Relacionais e de Objecto (SGBDRO);
- *Linguagens de programação*: são baseadas na arquitectura de linguagens de programação de bases de dados. As aplicações são escritas usando uma extensão de uma linguagem de programação existente com funcionalidades de bases de dados[11]. A maior parte dos sistemas de bases de dados deste género tende a suportar mais do que uma linguagem de programação para as aplicações. Oferecem um leque alargado de funcionalidades de bases de dados incluindo linguagens de interrogação e controlo de concorrência. Possibilitam um ambiente comum para os dados, e assim os objectos armazenados na base de dados aparecem semi-transparentemente como objectos da linguagem de programação. São designados em [11] por Sistemas de Gestão de Bases de Dados Orientadas por Objecto (SGBDOO);
- *Gestores de objectos persistentes*: oferecem funcionalidades básicas ao nível físico. Permitem a constituição de um repositório de objectos persistentes, embora não ofereçam características de bases de dados mais elaboradas como linguagens de interrogação. Podem oferecer controlo de concorrência;
- *Sistemas geradores de bases de dados*: estes sistemas permitem a construção de um SGBD adequado a necessidades específicas. Os modelo de dados, conceptual e interno,

[11] Como por exemplo *ObjectStore*, integrado com as linguagens de programação orientadas aos objectos C++ e Java.

possibilitam algum grau de configuração para cada implementação. Podem ser usados para obtenção de um SGBDO configurado à medida para determinadas aplicações.

Das arquitecturas focadas, as de maior relevância e utilização em termos comerciais são os SGBDOO e SGBDOR. A designação SGBDO poderá ser usada indiferentemente para cada um destes sistemas.

3.1.3 Características ligadas a SGBDOOs

Não existe um modelo de dados orientado por objectos universalmente aceite como acontece nos sistemas relacionais. Na realidade, a maior parte dos Sistemas de Gestão de Base de Dados Orientados por Objectos (SGBDOO), tentam capturar os mesmos conceitos que a programação orientada por objectos, acrescentando características adicionais para suportar armazenamento de grandes quantidades de objectos persistentes de forma partilhada. Estas características incluem suporte para interrogações, eficiente processamento sobre grandes volumes de informação em memória secundária, controlo de concorrência e recuperação de faltas.

A ideia básica subjacente à utilização de um SGBDOO é representar uma entidade ou objecto do domínio da aplicação com um correspondente objecto na base de dados. Isto inclui modelização do comportamento de cada objecto assim como a estrutura do objecto e relações com outros objectos definidos. Este mapeamento um-para-um reduz o fosso semântico e a desadaptação de impedâncias entre o domínio da aplicação e a modelização da base de dados para esse domínio.

Em sistemas orientados por objectos, todas as entidades são modelizadas como objectos. Um objecto é composto por uma parte estrutural (atributos, dados) e por um comportamento (métodos, código). São conhecidas as vantagens do ponto de vista de modularidade, localidade e expressividade deste paradigma. Daí que seja apelativo para os programadores adicionar persistência às suas aplicações, tornando-as capazes de armazenar e recuperar objectos, tal como eles são usados, sem terem que passar por um processo de conversão para outro formato, por exemplo, relacional. Esta é a primeira função que se espera um SGBD desempenhe. O esforço suplementar de memorizar estruturas mais complexas que os simples dados dos sistemas relacionais é claramente compensado pelas vantagens referidas.

Certos sistemas exigem a designação prévia de quais são as classes persistentes. Todas as instâncias dessas classes e só dessas permanecem armazenadas entre execuções de programas. Outros sistemas (como Pjava ou Java Persistente descrito em [2]) suportam o principio da persistência ortogonal, que significa que qualquer objecto pode ser persistente ou limitar-se a existir em memória, independentemente da sua classe. É claro o ganho em flexibilidade destes sistemas que implementam a persistência ortogonal. A indicação de qual o tratamento a dar, em termos de persistência, a um dado objecto é habitualmente feito por uma de três maneiras:

- Na escrita - através de uma operação de escrita explícita na base de dados de cada objecto a armazenar. Este método é indesejável pela responsabilidade que deixa ao programador;
- Na criação - através de uma modificação do construtor do objecto que permite escolher criar um objecto volátil ou um persistente, como por exemplo o SGBDOO ObjectStore;
- Atingibilidade - partindo de um ou mais objectos raiz, que referem outros objectos, que por sua vez referem terceiros, armazenam-se todos os objectos que seja possível atingir [2].

Comparado com um SGBD relacional convencional, um SGBDOO típico geralmente difere pelo menos nos seguintes aspectos:

- Um SGBDOO permite que o utilizador defina estruturas de dados e operações nelas definidas (classes e métodos), em vez de restringir os utilizadores a instâncias do tipo relação e um conjunto limitado de operações (operadores relacionais);
- Um SGBDOO suporta o conceito de identidade de um objecto. Isto significa que um objecto tem uma identidade independente dos valores dos seus atributos (esta identidade pode ser usada para referir o objecto a partir de outros objectos);
- Um SGBDOO suporta relações directas entre objectos. Os objectos relacionados com um dado objecto X podem ser acedidos por invocação de um método em X que exija os identificadores (oid's) existentes nas variáveis de instância de X. Logo, objectos relacionados podem ser tratados como se fizessem parte da estrutura interna do objecto, em vez de ser sempre necessário proceder-se a uma operação de junção em atributos do objecto (embora também sejam possíveis em SGBDOO operações de junção de atributos baseados em valores).

De um modo geral, para tirar partido do paradigma OO, as linguagens de um SGBDOO devem incorporar conceitos de polimorfismo e herança. Outros conceitos importantes a considerar são: mecanismos avançados de controlo de concorrência; mecanismos para manter múltiplas versões dos objectos; e uma variedade de bibliotecas de classes pré-definidas.

Um SGBDOO geralmente integra as seguintes características:

- Possibilidade de definir classes de objectos ou tipos usando linguagens de definição;
- Uma linguagem de programação incorporada e uma ou mais interfaces para linguagens de programação convencionais. Estas são usadas não só como interfaces para programação de aplicações convencionais, mas também para definição de métodos para objectos (comportamento) que formam parte da definição de classes;
- Uma linguagem de interrogação (uma forma de extensão de SQL para lidar com objectos);
- Um gestor de objectos ou "motor" de base de dados para suporte das operações básicas de base de dados;
- Uma biblioteca de classes pré-definidas para disponibilizarem as facilidades de base de dados para as aplicações, e para facilitar a tarefa de escrita dos métodos para os objectos;
- Um conjunto de ferramentas. Estas incluem capacidades de administração como ferramentas de projecto de esquemas, interfaces gráficas para a base de dados, ferramentas para melhorar desempenho da base de dados, etc.

Em fins da década de 80, foram publicados dois artigos com o intuito de clarificar quais as características que teriam de satisfazer os sistemas de base de dados da próxima geração, mais especificamente bases de dados orientadas por objectos [3] e bases de dados da terceira geração [57]. Este conjunto de características constituiu um marco importante nas bases de dados. Com base nas características focadas nesses artigos assim como em [11], foca-se a seguir as características que constituem requisitos para SGBDOO.

Segundo [3], um SGBD orientado por objectos deve satisfazer essencialmente, dois critérios: ser um sistema de gestão de base de dados; e ser um sistema orientado por objectos, consistente o mais possível com o conjunto de linguagens de programação orientadas por objectos actuais.

O primeiro critério leva a cinco requisitos: persistência; capacidade de gestão de memória secundária; concorrência; recuperação de faltas; e interrogações *ad hoc*.

O segundo critério conduz a oito requisitos: objectos complexos; identidade dos objectos; encapsulamento; tipos ou classes; herança; overriding, sobrecarga, ligação dinâmica (*late binding*); extensibilidade; e ser computacionalmente completa.

Em [57] é acrescentado um terceiro critério de que um SGBDO tem de suportar pelo menos as mesmas capacidades que um sistema relacional.

Objectos Complexos ou Compostos

O sistema deve permitir a construção de objectos com *estado complexo*, ou seja, que compreendam ou refiram outros objectos ou valores de modo a modelizar objectos para aplicações que requerem estruturas internas mais complexas.

Também deve suportar relações complexas entre objectos de modo a suportar correctamente a semântica da aplicação.

Objectos complexos são obtidos a partir de objectos mais simples pela aplicação recursiva de *construtores*. Os objectos mais simples, por vezes designados por tipos atómicos, são tipos como inteiros, caracteres, cadeias de bytes, booleanos e números reais. Existem vários construtores em questão dos quais são exemplo: estruturas, conjuntos, conjuntos com repetição de elementos (*bags*), listas e vectores.

Segundo [3], o sistema deverá suportar:

- Estruturas, pois são uma forma natural de armazenar as propriedades de um objecto;
- Conjuntos, que constituem a forma mais intuitiva para representação de colecções do mundo real;
- Listas e vectores são importantes pois são capazes de capturar a ordem que ocorre no mundo real.

Em [57], para além dos referidos acima, é também identificado o tipo união como desejável para introduzir a capacidade de definição de um valor de um atributo como provindo de um elemento de um conjunto de tipos. É igualmente indicado que seria desejável o suporte de funções como tipos pré-definidos ou primitivos.

Em [3], refere-se que os construtores devem ser ortogonais, ou seja, qualquer construtor deve ser possível aplicar a qualquer objecto. Devem também ser disponibilizados operadores para lidar com esses objectos, qualquer que seja o seu tipo de composição. Assim, operações em objectos complexos, devem propagar-se transitivamente para todos os seus componentes. Como exemplo temos o retorno ou remoção de todo um objecto complexo da base de dados, ou a produção de uma cópia *profunda*[12].

[12] Cópia por valor ou replicação, em que em vez de ser copiada apenas a referência, é criado um novo objecto com diferente identidade mas com os mesmos valores ou seja uma réplica do original.

Exemplo 3.1: exemplo de objectos complexo são os objectos da classe UDescrição usado na estruturação da informação arquivística.

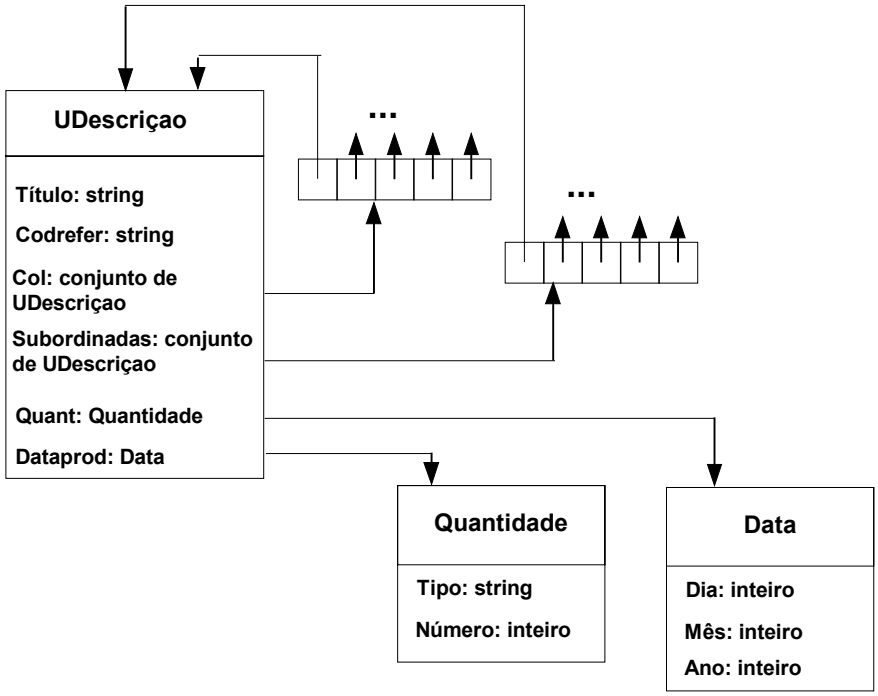


Figura 3.1: Objecto complexo

Uma definição simplificada dos tipos, usando uma sintaxe abstracta, poderá ser:

```
Tipo UDescrição
Propriedades:
  codrefer:  string;
  titulo:    string;
  dataprod:  Data;
  quant:     Quantidade;
  subordinadas: Lista de UDescrição;
  col: Lista de UDescrição;
Operações:
  imprime;
  mudaDataprod: Data {Data};
  insereSubordinada: UDescrição;

Tipo Quantidade
Propriedades:
  numero:    string;
  tipo:      inteiro;
Operações:
  imprime;
```

```
Tipo Data
Propriedades:
  dia: inteiro;
  mês: inteiro;
  ano: inteiro;
Operações:
  imprime;
```

□

Identidade de um objecto

A identidade de um objecto está relacionada com o facto de, no modelo, ele ter uma existência independente do seu valor [5], implementando também o conceito de entidade do mundo real dos modelos semânticos [61]. Um objecto diz-se possuir identidade se existe um modo de o referir independentemente do seu estado corrente [41].

Assim dois objectos podem ser idênticos, ou seja, são o mesmo objecto, ou eles podem ser iguais, tendo o mesmo valor. Objectos idênticos podem distinguir-se de objectos iguais em duas situações: partilha de objectos, em que duas estruturas podem partilhar uma parte comum e actualização de objectos, em que a actualização da parte comum numa das estruturas vai reflectir-se em todas as outras estruturas que partilham essa parte [3].

De um modo geral, a identidade de um objecto é implementada usando um *oid* que é atribuído explicitamente pelo sistema. Em [3], o suporte de identidade de objecto é considerado um requisito do sistema. Em [57], nota-se que deve ser atribuído um identificador único (UID) para o registo, apenas se este não tiver uma chave primária definida pelo utilizador. Esta afirmação, reforça o uso de chaves primárias como identificadores únicos, como acontece no modelo relacional.

Identidade de objecto é também uma poderosa primitiva de manipulação de dados, podendo ser a base de manipulação de conjuntos, n-tuplos e manipulação recursiva de objectos complexos.

O suporte de identidade de objecto, implica a existência de operações como atribuição de objecto, cópia de objecto (referencial e por replicação), e testes para a identidade de objectos e igualdade de objectos (profunda ou superficial).

Modelos baseados em identidade são norma em linguagens de programação imperativas em que cada objecto manipulado num programa tem uma identidade e pode ser actualizado. Esta identidade é obtida quer a partir do nome da variável quer da localização física em memória.

Encapsulamento

Segundo [3], a ideia de *encapsulamento* deriva da necessidade de:

- Uma clara distinção entre a especificação e a implementação de uma operação;
- Modularidade para estruturação de aplicações complexas onde o projecto e implementação está a cargo de uma equipa. É também necessária como uma ferramenta de protecção e autorização.

Numa linguagem de programação orientada por objectos, o conceito de encapsulamento advém da noção de tipos de dados abstractos. Nestes, um objecto tem uma parte de interface ou

um conjunto de operações visíveis para o exterior que definem o seu *comportamento*, uma de implementação que é composta pela parte dos dados e representa o *estado* do objecto, e por procedimentos.

Num sistema de base de dados, a parte estrutural do objecto poderá ou não estar visível através da interface. De qualquer modo, num SGBDOO, dados e procedimentos são ambos armazenados na base de dados, aplicando-se o mesmo modelo aos dois, permitindo assim a ocultação da informação. Nenhuma operação ,sobre objectos daquele tipo, pode ser efectuada para além daquelas definidas na interface. Esta restrição aplica-se quer às operações de actualização quer às de interrogação.

O encapsulamento oferece também uma forma de independência lógica visto ser possível a alteração da parte de implementação de um tipo sem ser necessário alterar os programas que usam esse tipo. O encapsulamento é total quando a interface do tipo é constituída apenas por operações, estando a parte dos dados não acessível para o exterior.

Enquanto [3] impõe o encapsulamento como característica de um SGBDOO, [57] considera-o "uma boa ideia".

Tipos e classes

Uma distinção possível entre os sistemas orientados por objectos, é no suporte da noção de classe ou tipo. Um tipo, numa linguagem orientada por objectos, sumaria as características comuns de um conjunto de objectos. Corresponde à noção de tipo de dados abstracto. É composto pela parte de interface e pela de implementação ou implementações. Enquanto a parte de interface é visível aos utilizadores do tipo, a implementação apenas está acessível ao projectista do tipo.

Nas linguagens de programação, os tipos são usados para aumentarem a produtividade através da certificação da correcção do programa. Forçando os utilizadores a declarar os tipos das variáveis e expressões utilizadas, o sistema deduz acerca da correcção do programa baseado na informação dos tipos. A verificação dos tipos apresenta vantagens em ser efectuada em tempo de compilação, embora por vezes tenha de ser deferida para tempo de execução. Em geral, tipos não são cidadãos de primeira classe ou seja, não podem ser modificados em tempo de execução.

A especificação de uma classe é a mesma que um tipo, embora represente mais uma noção em tempo de execução. A classe é composta por dois aspectos: a criação dos objectos ou *fábrica* dos objectos e o *armazém* ou seja o aspecto que permite a associação da classe à sua *extensão* (conjunto dos objectos que são instâncias dessa classe).

O sistema deve oferecer a possibilidade de construção de estruturas de dados quer sejam elas classes ou tipos[3]. A noção clássica de esquema de base de dados será substituída pela de conjunto de classes ou conjunto de tipos.

A manutenção da extensão do tipo poderá ser automática ou efectuada pelo utilizador que assim terá um maior controlo sobre o ou os conjuntos de objectos do mesmo tipo existentes na base de dados [3].

Completude computacional

A completude computacional significa a possibilidade de expressar qualquer função computável usando a linguagem de manipulação de dados do sistema de base de dados. É um requisito, pois deve ser possível a definição de aspectos comportamentais (operações ou métodos) dos objectos do sistema quando definidos pelo utilizador. A completude computacional pode ser conseguida

através da ligação a uma linguagem de programação, como acontece com a maioria dos sistemas de bases de dados de orientadas por objectos.

Enquanto [3] requer esta capacidade, [57] indica que procedimentos e métodos na base de dados são "uma boa ideia".

Extensibilidade

O sistema de base de dados vem com um conjunto de tipos pré-definidos ou primários. Esse conjunto de tipos deve ser extensível, ou seja, deve haver um modo de definir novos tipos e não deve haver distinção no uso de tipos disponibilizados pelo sistema e tipos definidos pelo utilizador [3]. O modo como o sistema suporta estas duas possibilidades de tipos deve ser invisível para a aplicação e para o programador.

A definição de um tipo inclui a definição das operações para esse tipo. A noção de encapsulamento implica a existência de um mecanismo para definição de novos tipos, a de extensibilidade reforça essa ideia ao requerer que novos tipos criados tenham o mesmo estatuto que os já existentes. Não é necessário no entanto que tipos colecções sejam extensíveis.

Em [57], afirma-se que é desejável o suporte de tipos de dados abstractos para construção de novos tipos base.

Hierarquias de Tipos ou Classes

A especificação de herança envolve a formação de hierarquias de tipos ou classes que organizam especificações das interfaces e implementações dos objectos, tornado-os mais concisos ao factorizarem as partes partilhadas.

O mecanismo de herança tem duas vantagens: é uma poderosa ferramenta pois dá uma descrição precisa e concisa do mundo real; e ajuda a agrupar especificações e implementações partilhadas nas aplicações.

Como permite que um objecto passe o seu estado ou comportamento para outro objecto, [3] requer o seu uso.

Exemplo 3.2: dado o tipo *Data* definido atrás, poderia ser definido um subtipo *Feriado* que iria partilhar todas as propriedades de *Data* e iria acrescentar por exemplo um atributo *motivo* onde estaria a descrição do motivo do feriado. □

Herança ajuda também a reutilização do código, pois todos os programas estão ao nível hierárquico que permita o maior número possível de objectos partilhá-los.

Relativamente a tipos de heranças, [3] identifica quatro: por substituição, inclusão, restrição e especialização.

- Em *herança por substituição*, dizemos que um tipo t herda de um tipo t', se conseguimos efectuar mais operações no objecto t que em t'. Esta forma de herança é baseada em comportamento e não em valor;
- *Herança por inclusão* corresponde à noção de classificação. Traduz que um tipo t é subtipo de um tipo t' se todos os objectos do tipo t são também objectos do tipo t'. A base desta forma de herança é a estrutura e não as operações;

- *Herança com restrições* é um caso da herança por inclusão. Um tipo t é um subtipo de um tipo t' se todos os objectos do tipo t são não só objectos do tipo t' como também satisfazem alguma restrição;
- Se um tipo t é subtipo de t' e se os objectos do tipo t são objectos do tipo de t' mas com acrescentados com informação mais específica, então temos o caso de *herança por especialização*.

Em [57], indica-se que herança é "uma boa ideia", chamando particular atenção para a herança com restrições.

Herança múltipla é uma generalização de herança que permite a um objecto herdar propriedades ou operações de mais do que um caminho na hierarquia (por contraponto com *herança simples*). [57] considera herança múltipla essencial enquanto [3] a considera opcional.

Herança selectiva, como referido em [56] traz mais flexibilidade ao modelo, na medida em que permite que um subtipo não herde determinadas propriedades do seu super-tipo. Este tipo de mecanismo, raramente é suportado pelos modelos orientados por objectos.

Polimorfismo, sobrecarga e ligação dinâmica

Em [3] é requerido que os SGBDOOs suportem alguma forma de sobrecarga do nome da operação e polimorfismo da operação. *Sobrecarga* é a capacidade de utilização do mesmo nome para diferentes operações. Um único nome pode então denotar vários programas.

Para diferentes tipos ou classes, a implementação de uma operação pode ser redefinida de modo que o mesmo nome denote diferentes programas. A esta capacidade de permitir que invocações do mesmo nome de operação resultem na execução de diferentes codificações para a operação, é dado o nome de *polimorfismo*.

A implementação dos tipos é desse modo separada da implementação da aplicação. Existem também vantagens a nível de simplicidade e manutenção. A tarefa de escolha de qual a operação a aplicar a um dado tipo, poderá ser executada automaticamente em tempo de execução através da ligação dinâmica aplicada a métodos. A associação entre o nome e o código das operações em tempo de execução em vez de ser em tempo de compilação, tem a desvantagem de tornar a verificação de tipos mais difícil.

Persistência

Persistência é a qualidade do programador fazer com que os dados do programa sobrevivam para além do tempo de execução do processo de modo a poderem ser reutilizados por outros processos. Este requisito é obvio do ponto de vista de bases de dados.

Cada objecto, independentemente do seu tipo, deverá poder ser tornado persistente sem uma transferência explícita, ou seja de um modo ortogonal. O utilizador não deverá ter de mover ou copiar informação explicitamente, para a tornar persistente[3].

Gestão de memória secundária

A gestão de memória secundária é uma capacidade tradicionalmente presente nos SGBDs. É usualmente suportada através de mecanismos como gestão de índices, agrupamento de dados, *buffering* da informação, selecção dos caminhos de acesso e optimização das interrogações.

Embora estes mecanismos devam manter-se invisíveis para o programador por uma questão de independência e separação entre o nível lógico e o nível físico dos sistema, eles devem estar presentes, pois é critica a sua ausência para a performance do sistema [3].

Em [57] refere-se que esta questão quase nada tem a ver com os modelos de dados, logo não deve ser aí mencionada. Refere também que a questão de explicitar agrupamentos específicos em memória secundária e outros tipos de optimizações, relacionados com performance devem ser especificados usando ferramentas de mais baixo nível.

Concorrência e transacções

Para vários utilizadores interagirem concorrentemente com o sistema, [3] afirma que deve ser oferecido um nível de serviço semelhante ao que existe nos sistemas de bases de dados actuais, permitindo que haja uma coexistência harmoniosa entre utilizadores que estejam a trabalhar simultaneamente. Devem assim ser oferecidos mecanismos de controlo de concorrência e processamento de transacções.

Em certos tipos de aplicações, nas quais são usadas SGBDOOs, o modelo de transacção tradicional não é satisfatório, pois confronta-se com o facto de lidar com transacções muito longas, envolvendo cooperação de aplicações. Assim, fica comprometido o uso de transacções como mecanismo de controlo de concorrência. Uma solução possível para este problema é o uso de versões (ver a seguir).

Recuperação e transacções

O sistema deve fornecer o mesmo nível de serviços que as bases de dados existentes para a recuperação de faltas de software e de hardware. Deverá ser possível repor um estado coerente dos dados após a falta [3].

O mecanismo de transacção, importante para efeitos de recuperação e para permitir o agrupamento de uma sequência de operações como se de uma operação só se tratasse, sofre também com o facto de lidar com transacções muito longas. Surgem assim as transacções encaixadas dentro de transacções mais longas. Transacções encaixadas, são transacções que permitem confirmar ou abortar um grupo de actualizações numa dada transacção sem obrigar à confirmação ou aborto de outros grupos de operações pertencentes a transacções adjacentes [11].

Interrogações

Segundo [3], um SGBDOO deve oferecer funcionalidades correspondentes às das linguagens de interrogação presentes nas bases de dados relacionais. No contexto das SGBDRO, [57] aponta requisitos idênticos.

O serviço consiste em permitir aos utilizadores executar interrogações à base de dados de um modo simples, e pode ser fornecido sob a forma de uma interface gráfica ou como parte de uma linguagem de definição de dados, não sendo absolutamente necessário que seja sob a forma de uma linguagem de interrogação autónoma [3].

A facilidade de interrogação, deve satisfazer seguintes critérios [3],[57]:

- ser de alto nível de modo a ser possível expressar interrogações não triviais de um modo conciso. Isto implica um grau de declarativiade, enfatizando o *que* e não o *como*;

- ser eficiente: a formulação de interrogações deve integrar uma forma de optimização [3];
- ser independente da aplicação, seguindo uma norma para que possa trabalhar em qualquer base de dados [3];
- deve haver pelo menos dois modos de especificar colecções, uma usando enumeração de membros e outra usando uma linguagem de interrogação para especificar condições de pertença dos membros ao conjunto [57];
- "para o melhor ou para o pior, SQL é a linguagem de bases de dados intergaláctica" [57];
- interrogações e o seu resultado, devem constituir o nível mais baixo de comunicação entre o cliente e o servidor [57].

Uma linguagem de acesso à base de dados pode ser dividida em três sublinguagens, uma Linguagem de Definição dos Dados - LDD, uma Linguagem para Manipulação dos Dados LMD e finalmente uma de interrogação.

Em [11] é feita a abordagem às linguagens de interrogação e linguagens de programação, confrontando-as em relação a sete aspectos:

Ambiente: as linguagens de programação para bases de dados (SGBDOO) são extensões das linguagens de programação existentes. Partilham o mesmo ambiente e a mesma estrutura de tipos, assim são executadas no mesmo processo.

Os sistemas de bases de dados relacionais estendidos (SGBDRO), contudo, têm uma linguagem de base de dados separada da linguagem de programação da aplicação, e embora a linguagem de base de dados possa ter capacidades de programação, as duas linguagens trabalham em diferentes ambientes. Esta situação traduz-se na necessidade de embutir a linguagem de base de dados na aplicação, o que pode levar ao já referido problema da desadaptação de impedâncias ou, pelo menos, da conversão de tipos.

Embora a linguagem de base de dados num SGBDRO, tipicamente seja computacionalmente completa, sofre frequentemente de problemas de acesso a recursos como por exemplo a programação da interface gráfica ou a entrada de dados pelo utilizador. Daí a necessidade da utilização de uma linguagem para as aplicações.

Resultados das interrogações: os SGBDOs diferem no tipo de resultados que podem ser obtidos através das interrogações não procedimentais, ou seja, declarativas como em SQL.

Os SGBDO, oferecem de um modo geral uma das seguintes soluções quanto à linguagem de interrogação:

Não dispor de capacidades de interrogação - alguns sistemas não oferecem acesso associativo aos objectos, mas apenas possibilidade de "navegar" através de caminhos de acesso pelos objectos da base de dados. Existem algumas aplicações específicas onde a capacidade de interrogação é um factor de baixa prioridade. As interrogações nestes casos, são escritas sob a forma de programas.

Resultados sob a forma de colecções - neste caso, os sistemas permitem a facilidade de acesso associativo apenas em objectos que satisfazem uma dada restrição ou sobre uma dada colecção de objectos, como por exemplo a extensão de um tipo, e não sobre toda a base de dados.

Resultados sob a forma de relações - os sistemas relacionais estendidos, apresentam facilidades de interrogação pelo menos idênticas às do SQL, excedendo até, de um modo geral, o poder expressivo desta linguagem.

Todo o tipo de resultado - o standard ODMG exibe uma linguagem de interrogação cujo resultado pode ser de qualquer tipo, um valor literal, um objecto, uma lista ou conjunto de objectos ou um vector de dados.

Encapsulamento: alguns SGBDOs aderem ao conceito estrito de encapsulamento, em que só métodos definidos na interface (na zona pública do objecto) estão visíveis aos utilizadores da linguagem de base de dados. Isto oferece um mecanismo de independência de dados. Outros sistemas violam o encapsulamento com o propósito de o utilizador da linguagem de interrogação poder ver todos os atributos de um objecto. Alguns sistemas não têm nenhum mecanismo de encapsulamento.

O encapsulamento coloca restrições na utilização da linguagem de interrogação. As linguagens de interrogação tradicionais foram projectadas com a assunção de que oferecer uma estrutura simples, uniforme e visível dos objectos operados. Uma aproximação orientada por objectos mais estrita, dita que o acesso aos atributos e relações dos objectos, seja feita através dos métodos da sua interface.

Dados virtuais: alguns SGBDOs permitem que a informação devolvida pela linguagem de base de dados seja definida usando atributos derivados (ver a seguir vistas e dados derivados) e relacionamentos. Se o facto da informação ser derivada ou estar explicitamente armazenada na base de dados for transparente para o utilizador, diz-se que a informação é virtual. Informação virtual fornece outro mecanismo para a independência dos dados, permitindo a mudança do esquema conceptual de dados sem a modificação da aplicação.

Modelo de dados: as linguagens de bases de dados que os SGBDOs apresentam, diferem também no modelo de dados: relacional, funcional ou orientado por objectos. O modelo de dados afecta a sintaxe e semântica das linguagens de bases de dados. Como a maioria dos SGBDOs apresentam uma linguagem de base de dados computacionalmente completa, a questão do modelo de dados coloca-se mais ao nível sintáctico e na discussão de qual será a abordagem mais natural.

Extensão com operações: os SGBDOs de um modo geral, estendem as linguagens convencionais e declarativas para interrogação como o SQL, com novas operações como o fecho transitivo, recursividade e regras. Outra extensão útil, centra-se na capacidade de pesquisa e tratamento de texto normal.

Normalização: um aspecto importante no uso de um SGBDO em utilizações correntes é a compatibilidade com os standards existentes para linguagens de interrogação como o SQL, e compatibilidade com outros SGBDOs tanto na linguagem de programação como na de interrogação. Estes permitem a escrita de programas que possam trabalhar com mais do que um SGBDO.

3.1.4 Características Optativas

Além das características referidas até aqui, outras existem que, embora não consideradas essenciais para a classificação de um sistema como sendo SGBDOO, enriquecem claramente o sistema com a sua presença.

Estas, podem ser dividas entre as que têm natureza orientada por objectos, como a herança múltipla, e as mais ligadas a bases de dados, como as transacções. Do segundo grupo, a maioria tem como objectivo o serviço de novas aplicações que têm *rebocado* um pouco os SGBDOs existentes do ponto de vista de conquista de mercado.

Verificação e inferência de tipos

Os SGBDOOs diferem na firmeza de verificação de tipos ou na possibilidade de determinar automaticamente o tipo dos dados derivados - inferência de tipos. Isto depende essencialmente da noção de sistema de tipos implementada. Uma das motivações no desenvolvimento de linguagens de programação para bases de dados tem a ver com o interesse em preservar uma verificação rígida de tipos quando se trata da interface entre os ambientes da linguagem de programação e da linguagem de base de dados. A questão coloca-se também quando se está a desenvolver linguagens de programação optimizáveis e fechadas.

Estas considerações são mais importantes quando se trata de SGBDOO do que no caso dos sistemas relacionais, pois tem-se aqui um sistema de tipos mais rico, com estruturas mais complexas produzidas pelas operações da base de dados.

A capacidade de verificação e inferência de tipos não é prescrita como característica obrigatória em [3]. No entanto enfatiza-se que a situação óptima se atinge quando é possível garantir que um programa não provoca erros de execução pelo facto de não ter acusado nenhum erro de compilação.

Distribuição

O suporte para distribuição é considerado opcional em [3], porque o suporte para a distribuição do sistema é independente de este ser classificado como SGBDOO. Considerações semelhantes são efectuadas em [57].

No entanto, para o tipo genérico de ambiente em que as aplicações mais avançadas são desenvolvidas, é essencial permitir o acesso a objectos residindo em componentes distribuídos. Isto envolve a capacidade das linguagens de bases de dados suportarem a especificação das características dos objectos distribuídos pela rede de computadores.

A maioria dos SGBDOs é implementada usando a filosofia cliente/servidor, sendo alguns implementados com características totalmente distribuídas.

A integração de SGBDOOs com a arquitectura de distribuição CORBA[13], definida pela OMG[14], é já explicitamente considerada na norma da ODMG.

Versões e Configurações

Muitas das aplicações para as quais os SGBDOOs estão direccionados, envolvem várias formas de actividade de projecto, requerendo portanto a capacidade de suportar várias *versões* de todo o projecto ou suas componentes que podem ser desenvolvidas ou estar em desenvolvimento ao mesmo tempo.

O uso de múltiplas versões dos objectos permite coordenação entre vários utilizadores, que trabalham com diferentes cópias dos objectos, sendo possível, ao nível mais básico, a criação ou remoção de versões de objectos.

O SGBDOO pode permitir que as versões sejam também aplicadas ao próprio esquema da base de dados, simplificando o problema de evolução do esquema focado a seguir [11].

---

[13] *Common Object Request Broker Architecture*

[14] *Object Management Group*

Um dos problemas levantados pela manutenção de versões, relaciona-se com a actualização de referências para uma dada versão do objecto quando esta é criada ou destruída. A questão de um utilizador saber quais as versões de que objectos estão interligadas de uma forma consistente é resolvida com o conceito de *configuração*.

Uma configuração é assim uma colecção de versões de objectos numa base de dados que estão mutuamente consistentes.

Existem duas aproximações possíveis para a implementação de configurações [11]:

- O SGBDOO pode dispor de um mecanismo mas não de uma política de gestão de configurações, deixando o problema de gestão das configurações ao cargo do utilizador. Este mecanismo pode ser na forma de operações, associadas aos atributos referência com membro inverso, que são desencadeadas quando o objecto referenciado sofre uma actualização. Exemplos de operações são: mover a referência para a nova versão, abandonar a referência na nova versão ou copiar a referência para a nova versão. Outra abordagem é a definição de referências estáticas (a referência mantém-se apenas para a versão antiga) ou dinâmicas (a referência aponta sempre para a versão mais recente do objecto);
- O SGBDOO pode implementar as configurações de um modo automático com algumas opções especificadas pelo utilizador. Em ObjectStore é implementado usando objectos especiais para representarem configurações. Existe a noção de configuração corrente, e quando é feita uma actualização, esta é efectuada no contexto da configuração corrente. O mesmo acontece em operações de leitura. O utilizador pode abrir uma configuração existente ou criar uma nova configuração como mudança da configuração actual.

No entanto, [3] não considera o suporte de versões como sendo nuclear para que o sistema possa ser classificado como SGBDOO.

3.1.5 Outras Características

Neste ponto, refere-se um conjunto de características para as quais os autores de [3] não chegaram a um consenso quanto ao carácter obrigatório ou opcional.

Vistas e dados derivados

Num SGBD relacional, a capacidade de criar *vistas* sobre a base de dados significa a definição de relações que não estão explicitamente em memória persistente, mas sim que são computadas a partir de relações base. Uma extensão deste conceito é a permissão de quaisquer dados arbitrários poderem ser derivados de informação persistente.

Para um SGBDOO, seria de grande utilidade o suporte de características semelhantes a esta, ou seja, derivar novos objectos a partir de objectos já existentes. Por exemplo, um mecanismo de criação de vistas permitiria revelar de um modo selectivo diferentes aspectos do seu comportamento, ou providenciar visibilidade de diferentes colecções de objectos, dependendo da vista.

Em [57] é referido que a existência de vistas com possibilidade de actualização, constitui condição essencial.

A construção de vistas e dados derivados põe determinados requisitos na linguagem de base de dados. Em particular, a linguagem deve ser capaz de construir dinamicamente novos tipos e

suas instâncias. Isto requer que a linguagem tenha certas propriedades de fecho na operação de criação de tipos dinâmicos. Adicionalmente, para uma verdadeira completude, deve ser possível para os dados derivados e vistas, a construção de operações sobre objectos, e não apenas composição a nível estrutural. Isto requer que a linguagem permita definir uma composição de operações e tratar o resultado como uma operação.

Utilitários de administração de bases de dados

É de alguma importância que um SGBDOO providencie um conjunto completo de utilitários de administração. No conjunto de utilitários possíveis incluem-se ferramentas para reorganização da base de dados, monitorização de estatísticas, ferramentas que permitam manter listagens geradas automaticamente reportando a evolução do estado da base de dados, e ferramentas para proceder ao arquivo da base de dados.

O modelo OO não especifica tais utilitários de uma forma explícita. No entanto, os utilitários de administração seriam necessariamente influenciados pelo modelo OO.

Restrições de integridade

As restrições de integridade constituem conjuntos de regras que deverão ser sempre respeitadas pelo estado da base de dados e podem ser usadas pelo SGBD para assegurar a correcção e consistência da base de dados.

Alguns SGBDs relacionais oferecem um mecanismo designado por *trigger* para suportar e fazer respeitar as restrições impostas. *Triggers* são acções que são desencadeadas quando se dá o acesso a determinados itens, podendo ser usados para verificar o cumprimento das restrições ou para executar outro tipo de operações adicionais com vista a colocar a base de dados num estado consistente.

Num SGBDOO, podem ser incluídas algumas formas de restrições na altura da definição das operações dos objectos. No entanto, noutros casos será mais apropriado permitir a definição de restrições de um modo independente das operações individuais de objectos, como no caso dos *triggers*.

Em [57] é explicitado que as regras (*triggers* ou restrições) se irão tornar uma das características mais importantes dos sistemas futuros. Não devem por isso estar associadas com funções ou colecções específicas.

Protecção e segurança

Quer em [3] quer em [57] não é mencionado explicitamente nada com relação às capacidades de segurança que um SGBDOO deve possuir. No entanto, tal item pode ser incluído quer nos utilitários de administração quer na definição de restrições de integridade. Em qualquer dos casos, a definição de vistas e capacidade de definição de vistas pode também constituir a base para capacidades de segurança de SGBDOOs.

Evolução de esquemas

A evolução do esquema num SGBDOO refere-se ao problema da modificação da definição de tipos ou a hierarquia de tipos da base de dados, preservando ao mesmo tempo a consistência entre as definições alteradas e os objectos instanciados na base de dados [22].

Idealmente, deveria ser possível proceder à modificação do esquema da base de dados de um modo arbitrário, sem obrigar à paragem do sistema. No entanto, na prática isto pode ser bastante difícil para determinados tipos de modificações de esquemas.

Sistema de tipos

A única propriedade explicitamente requerida por [3] para formação de tipos é a disponibilização de um construtor de objectos para permitir a definição de novos objectos. Outros criadores de tipos podem também ser incluídos num SGBDOO como geradores de tipos (por exemplo *set(T)* para criar um tipo conjunto de elementos do tipo arbitrário *T*), restrições, ou união de tipos. Construtores de tipos são de particular importância para os requisitos de uma linguagem de base de dados (ver ponto de objectos complexos).

Uniformidade

A questão aqui focaliza-se nas seguintes interrogações:

- É um tipo um objecto?
- É um método um objecto?
- Devem as três noções referidas, ser tratadas separadamente?

O problema põe-se a três níveis: implementação, linguagem de programação e interface de programação.

No nível de implementação, a decisão prende-se com o tratamento da informação sobre tipos como objectos ou de um modo *ad hoc*. A decisão deve ser baseada em critérios de desempenho e facilidade de implementação.

No nível de linguagem de programação, a questão é se os tipos são cidadãos de primeira classe na semântica da linguagem. Há provavelmente estilos diferentes de uniformidade - sintáctica ou semântica. Uniformidade total a este nível é também inconsistente com a verificação de tipos estática.

Por último, ao nível da interface de programação, a decisão centra-se na disponibilização para o utilizador de uma visão uniforme sobre tipos, objectos e métodos mesmo se a semântica da linguagem impõe distinções. O converso também é possível.

3.2 Standard ODMG93 v2.0

A ODMG foi fundada em 1991 impulsionada por um grupo de empresas dedicadas à comercialização de SGBDOOs. Até então, a falta de standards na área demonstrava ser uma limitação para que o uso de SGBDOOs fosse mais alargado. A metodologia do trabalho de normalização foi a de recolher e tentar agrupar as melhores ideias e características de implementações de SGBDOOs existentes.

Objectivos

O sucesso da filosofia de orientação por objectos no campo de bases de dados passa também pelo nível de normalização oferecido. A aceitação de um standard, possibilita um alto grau de portabilidade e interoperabilidade entre sistemas, simplificando a aprendizagem de novos SGBDOOs e constituindo uma alavanca para a confirmação do uso desta tecnologia.

Com base nessas premissas, foram ditados objectivos para o processo de elaboração das normas ODMG:

- disponibilização de um conjunto de normas com vista a permitir que utilizadores de SGBDOO escrevam aplicações que possam ser executadas em mais do que um produto SGBDOO. Esta portabilidade ao nível de código fonte, deverá abranger o esquema de dados, ligação às linguagens de programação, e linguagem de manipulação assim como linguagem de interrogação;
- as propostas de normalização, deverão ter papel decisivo também na interoperabilidade entre SGBDOOs, nomeadamente com a abertura da possibilidade de comunicação de bases de dados distribuídas heterogéneas através da ligação ao *Object Request Broker* da OMG;
- a integração harmoniosa das linguagens de programação com os sistemas de bases de dados é também um ponto fulcral, pois permite um maior avanço na industria de software orientado por objectos como um todo;
- outro objectivo importante é o relacionamento com outros grupos de elaboração de standards como a OMG na qual a ODMG é filiada desde 1994. O standard ODMG, está adoptado como serviço persistente por parte da OMG, e a linguagem de interrogação da ODMG - OQL definida como serviço de interrogação para pesquisa de objectos OMG. Estão também estabelecidas ligações com os comités ANSI X3H2 (SQL), XJ16 (linguagem C++) e X3J20 (lingugagem Smalltalk), estando a ser estudada a convergência entre OQL e o novo standard SQL - SQL3.

Definição

O contexto do desenvolvimento do standard ODMG, difere significativamente dos sistemas relacionais. Um SGBDOO integra de um modo transparente as capacidades de bases de dados com a linguagem de programação das aplicações, abolindo assim a desadaptação de impedâncias existente. Esta transparência evita a necessidade de cópia e tradução explícita entre a informação na base de dados e a sua representação na linguagem de programação. Os SGBDOOs incluem

capacidades de interrogação através de uma linguagem que incorpora diversos tipos de colecções, como listas e vectores, permitindo que os resultados de uma interrogação sejam de qualquer tipo.

Um SGBDOO pode ser visto como uma extensão, com capacidades de armazenamento transparente e persistente de objectos, da linguagem de programação associada. A informação residente na base de dados fica armazenada sob a forma de objectos. Apresenta também outras características inerentes a uma base de dados como controlo de concorrência, recuperação, interrogações associativas entre outras funcionalidades.

Arquitectura

As principais componentes da norma ODMG 2.0, a serem descritas nesta secção, são:

- Modelo Objecto: o modelo de dados que deverá ser comum a todos os SGBDOOs. Tem como base o modelo objecto da OMG. Segundo a arquitectura OMG, o standard ODMG constitui um perfil para sistemas de bases de dados objecto, adicionando componentes ao núcleo do modelo OMG de modo a suportar tal;
- Linguagem de Especificação: as linguagens de especificação previstas são duas. Uma de definição de objectos - ODL por confronto com as tradicionais linguagens de definição de dados e usando a linguagem de definição de interfaces - IDL da OMG como base. A outra linguagem definida, permite a troca de objectos entre bases de dados, sendo designada por OIF (*object interchange format*);
- Linguagem de Interrogação - OQL: definição de uma linguagem de interrogação declarativa usando como base o SQL;
- Definição das ligações às linguagens de programação C++, Smalltalk e Java: define versões da ODL para utilização com estas linguagens. Prevê uma linguagem de manipulação de objectos - OML associada a cada linguagem. Especifica mecanismos para invocação da OQL e procedimentos para operações na base de dados e transacções.

Um dos objectivos é tornar possível a leitura e escrita da mesma base de dados usando uma das três linguagens de programação indiferentemente. Isto é possível se for usado o subconjunto dos tipos suportados por todas as três linguagens referidas.

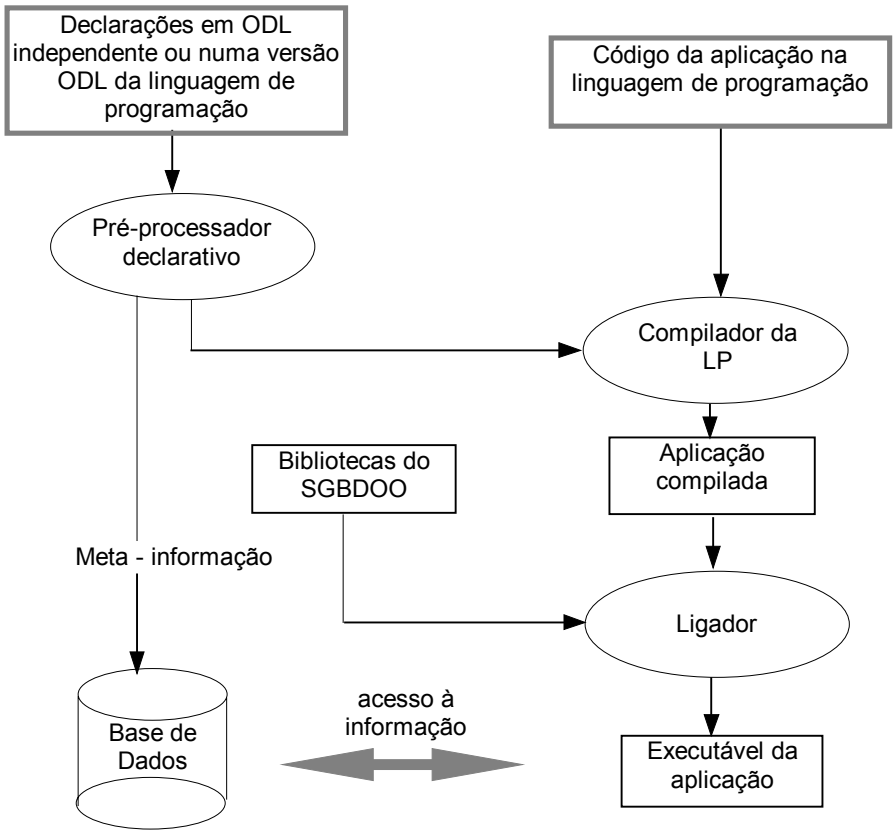


Figura 3.2: Processo de desenvolvimento de uma BDOO e aplicação

Na figura acima é descrito sumariamente o processo de elaboração de uma aplicação típica que se procura normalizar com ODMG. O programador elabora a declaração do esquema aplicacional da base de dados, incluindo a parte estrutural e a comportamental, e usando a ODL. Escreve também o programa na linguagem de programação escolhida, incluindo a interface de programação para uso das funcionalidades do SGBDOO disponibilizadas pela biblioteca de classes que disponibiliza toda a parte de OML e OQL, transacções e outras.

As declarações e o programa fonte são depois compilados e ligados de modo a produzir uma nova base de dados ou uma actualização de uma já existente, e um executável para a aplicação que vai interagir com a base de dados criada.

3.2.1 Modelo Objecto ODMG

O modelo objecto da ODMG (ODMG/OM) especifica os tipos de semântica que podem ser definidos explicitamente para um SGBDOO. Entre outras coisas, a semântica do modelo define as características dos objectos, como os objectos se relacionam entre si e como os objectos podem ser nomeados e identificados.

O modelo, especifica as construções que são suportadas por um SGBDOO:

- as primitivas básicas são o *objecto* e o *literal*. Cada objecto tem um identificador único. Um literal não possui identificador;
- Objectos e literais podem ser categorizados pelos seus *tipos*. Todos os elementos de um determinado tipo têm uma gama comum de estados (partilham o mesmo conjunto de

propriedades) e um comportamento comum (partilham o mesmo conjunto de operações definidas). Um objecto pode ser referido como uma *instância* de um tipo;

- O estado de um objecto é definido pelos valores das suas *propriedades*. Estas podem ser *atributos* do objecto ou *relacionamentos* entre o objecto e um ou mais objectos. De um modo geral, o valor de uma propriedade de um objecto pode variar ao longo da vida do objecto;
- O comportamento de um objecto é definido pelo conjunto de operações que podem ser executadas sobre ou pelo objecto. As operações podem ter uma lista de parâmetros quer de entrada quer de saída, e cada um com um determinado tipo. Cada operação pode também devolver um resultado que possui um determinado tipo;
- Uma base de dados armazena objectos partilhando-os por vários utilizadores e aplicações. Um base de dados é baseada num *esquema* que é definido usando uma linguagem de definição de objectos - ODL, e contém instâncias de tipos definidos pelo seu esquema.

O modelo ODMG/OM inclui uma semântica mais rica do que o modelo relacional, ao declarar relacionamentos e operações explicitamente [13].

Especificação e Implementação de Tipos

Um tipo tem uma *especificação* externa e uma ou mais *implementações*. A especificação define as características externas do tipo. Estas são a parte visível para o utilizador do tipo: as operações passíveis de serem invocadas nas suas instâncias; as *propriedades*, ou estado das variáveis, cujos valores podem ser acedidos; e quaisquer *excepções* eventualmente provocadas pelas suas operações. A implementação de um tipo define os detalhes internos dos objectos desse tipo como implementação das operações e outros.

A distinção entre especificação e implementação é o modo como o modelo reflecte o encapsulamento.

Existem três tipos de especificação. Uma definição de *interface* é uma especificação que contempla apenas o comportamento de um tipo objecto. Uma definição de *classe* é uma especificação que define o comportamento abstracto e o estado abstracto de um tipo objecto. Uma definição de *literal* determina apenas o estado abstracto de um tipo literal.

Uma implementação de um tipo objecto consiste numa *representação* e num conjunto de *métodos*. A representação é a estrutura de dados derivada do estado abstracto do tipo através de uma *ligação a uma linguagem* específica: para cada propriedade contida no estado abstracto, há definida uma variável de instância de um dado tipo. Os métodos são corpos de procedimentos derivados do comportamento abstracto do tipo através da ligação a uma linguagem específica: para cada operação definida no comportamento abstracto do tipo, é definido um método. Este método implementa um comportamento visível para o tipo objecto.

Cada ligação a linguagens específicas também define um mapeamento para a implementação dos tipos literais, dependendo das construções existentes na linguagem.

A implementação de um tipo está dependente da linguagem assim como de pormenores de arquitectura do sistema, pelo que preserva a intenção semântica do tipo manter separadas as partes de especificação e implementação. As classes, na finalidade de construções existentes em linguagens como C++, Smalltalk e Java, são classes de implementação e não *classes abstractas* definidas no modelo orientado por objectos.

Subtipagem e herança comportamental:

O modelo ODMG inclui herança baseada em relacionamentos tipo/subtipo. Estes relacionamentos são vulgarmente representados por grafos, em que cada nó é um tipo e um arco liga um supertipo a um subtipo. O relacionamento tipo/subtipo é normalmente designado por *relacionamento is_a* ou *ISA*. Outra designação possível é de relacionamento *generalização - especialização*. O supertipo é o mais genérico, o subtipo o mais específico. O *tipo mais específico* de um objecto é o tipo da hierarquia de tipos que descreve todo o comportamento e propriedades da instância. A interface de um subtipo pode definir características adicionais às definidas para os seus supertipos.

O modelo ODMG/OM suporta herança múltipla do comportamento de objectos. No entanto não permite a sobrecarga de nomes durante a herança.

As classes são tipos directamente instanciáveis, pois instâncias desses tipos podem ser criadas pelo programador. No entanto, o mesmo já não acontece com as interfaces que não são directamente instanciáveis. A subtipagem concerne apenas a herança de comportamento. Devido à existência de ambiguidades na herança múltipla de estado não é possível a heranças de tipos em que o supertipo é classe.

Herança de Estado:

Além dos relacionamentos ISA para definição de heranças comportamentais, o modelo ODMG/OM define também relacionamentos de extensão (EXTENDS) para herança de estado. O relacionamento EXTENDS aplica-se apenas a tipos objecto, logo apenas classes, e não literais, podem herdar estado. EXTENDS, é um relacionamento de herança simples entre duas classes, onde a classe subordinada herda todas as propriedades e todo o comportamento da classe que estende. O relacionamento EXTENDS é transitivo.

A única excepção legal à proibição de sobrecarga de nomes ocorre quando a mesma declaração de propriedade ocorre numa classe e numa das suas interfaces herdadas.

Extensão:

A extensão de um tipo é o conjunto de todas as instâncias do tipo dentro de uma base de dados particular. Se um objecto é instância do tipo A, então pertence à extensão de A. Se tipo A é subtipo de B, então a extensão de A é subconjunto da extensão de B.

A manutenção da extensão de uma classe em SGBDOs pode ou não ser automática.

Chaves:

Nalguns casos, instâncias individuais de um tipo podem ser identificadas univocamente pelo valor que possuem numa - ou num conjunto de - propriedades. Estas propriedades identificadoras são designadas por *chaves* (*chaves candidatas* em sistemas relacionais). Uma *chave simples* tem apenas uma propriedade, uma *chave composta* consiste num conjunto de propriedades. O contexto de unicidade é a extensão do tipo, logo um tipo tem de possuir uma extensão para ter uma chave.

Objectos

Os objectos são criados através da invocação de operações de criação em *interfaces construtoras* disponíveis em objectos construtores fornecidos ao programador pela implementação da ligação à linguagem de programação. Neste exemplo, a operação *new()* retorna um novo objecto do tipo *Object*:

```
interface ObjectFactory { Object new(); };
```

Identificadores de objectos:

No modelo ODMG/OM todos os objectos têm identificadores, os *identificadores de objecto*, que são únicos na sua base de dados ou *domínio de armazenamento*. Os identificadores de objecto são gerados pelo SGBDOO, mantêm-se imutáveis durante toda a vida do objecto e são vulgarmente usados como meio de um objecto referir outro.

Os literais não têm o seu próprio identificador e não podem existir isoladamente como os objectos. São embutidos em objectos não podendo ser referenciados individualmente.

Nomes de objectos:

Um objecto pode ter um ou mais nomes. O SGBDOO oferece uma função para efectuar o mapeamento entre os nomes do objecto e o objecto. A aplicação pode referir-se a um objecto por nome. Os nomes podem ser usados pela aplicação para se referir a *objectos raiz* ou *pontos de entrada* na base de dados. O contexto de unicidade de nomes é uma base de dados, não havendo hierarquia de espaços de nomes dentro da base de dados.

Tempo de vida de um objecto:

O tempo de vida de um objecto é definido quando este é criado, é independente do tipo do objecto, e pode ser dividido em duas categorias:

- Temporário: o tempo de vida do objecto é no máximo igual à duração do processo a ser executado, estando o objecto alocado em memória estática ou na *heap*;
- Persistente: este tipo de objecto é armazenado pelo SGBDO para além do processo, continuando a sua existência. Objectos persistentes são também designados por *objectos da base de dados*.

O mesmo conjunto de operações pode ser aplicado a ambas as categorias.

Tipos de objectos:

- Atómicos: são definidos pelo utilizador, que especifica as suas propriedades e o seu comportamento;
- Colecções: instâncias de colecções são composições de elementos distintos. Esses elementos podem ser de qualquer tipo, mas têm de ser todos do mesmo tipo. Uma *Colecção*, é um tipo abstracto derivado do tipo *Objecto*. Colecções são criadas usando a interface *CollectionFactory*, sendo possível especificar o seu tamanho na altura da criação. Os geradores de subtipos de Colecção são parametrizados pelo tipo dos elementos da colecção, referido entre parênteses rectos. Estes são:
    - Conjuntos - Set<t>: colecção não ordenada sem repetição de elementos;
    - Multi-conjuntos - Bag<t>: colecção não ordenada que permite a repetição de elementos;
    - Listas - List<t>: colecção ordenada de elementos com tamanho indefinido;
    - Vectores - Array<t>: colecção ordenada de elementos com tamanho definido dinamicamente;
    - Dicionários - Dictionary<t,v>: sequência não ordenada de pares chave-valor, sem duplicação de chaves, as quais são usadas para obter o respectivo valor.

  Cada um dos tipos de colecção referidos fornece um conjunto de operações para manipulação das colecções de acordo com a sua semântica. Por exemplo, objectos *Set<t>*

têm definidas operações matemáticas convencionais para lidar com conjuntos. Para percorrer os elementos de uma colecção é possível recorrer a *iteradores* ou a *iteradores bidireccionais* para colecções ordenadas. Mecanismos Iteradores são criados usando as operações *create_iterator()* e *create_bidirectional_iterator()* presentes na interface Colecção, que retornam os respectivos objectos;

- Estruturados: tipos de objectos estruturados definidos pelo modelo, derivam directamente do tipo abstracto *Objecto*. Os tipos estruturado são:
    - Data (*Date*): permite armazenar uma data;
    - Intervalo de tempo (*Interval*): permite armazenar um intervalo de tempo. Objectos Intervalo de tempo são criados como resultado de operações de diferença aplicadas a objectos Tempo;
    - Tempo (*Time*): denotam um tempo especifico (GMT) tendo em conta a zona temporal;
    - Marca temporal (*Timestamp*): agrupa um tempo e uma data. Pode retornar um objecto Tempo ou um objecto Data.

Literais

Literais não têm identificadores de objectos. Os tipos literais atómicos definidos são também suportados pela linguagem de definição de interface da OMG. O objectivo é permitir uma fácil ligação entre o modelo e linguagens de programação específicas. Se a linguagem não tiver definidos tipos análogos aos do modelo orientado por objectos, deverá ser usada uma biblioteca para expansão dos tipos dessa linguagem.

O modelo ODMG, inclui também um tipo Tabela de modo a expressar tabelas SQL. Este tipo é semanticamente idêntico a uma colecção de estruturas.

Para todos os tipos literais, existe outro tipo literal suportando valores nulos. Este tipo nulo, é o mesmo que o tipo literal mas estendido com o valor nulo *nil*.

Hierarquia completa de Tipos Primários do Modelo

O modelo é fortemente tipado, obrigando a que todos os objecto e literais tenham um tipo e todas as operações requeiram operandos tipados. Dois objectos ou literais têm o mesmo tipo só se foram declarados como instâncias de tipos com o mesmo nome. Nos relacionamentos de subtipagem, se TS é um subtipo de T, então um objecto do tipo TS pode ser atribuído a uma variável do tipo T, mas o contrário já não é possível.

Na figura retirada de [13], os tipos abstractos estão em itálico e os restantes, os tipos concretos, representam os tipos directamente instanciáveis. Os geradores de tipos têm parênteses rectos a seguir.

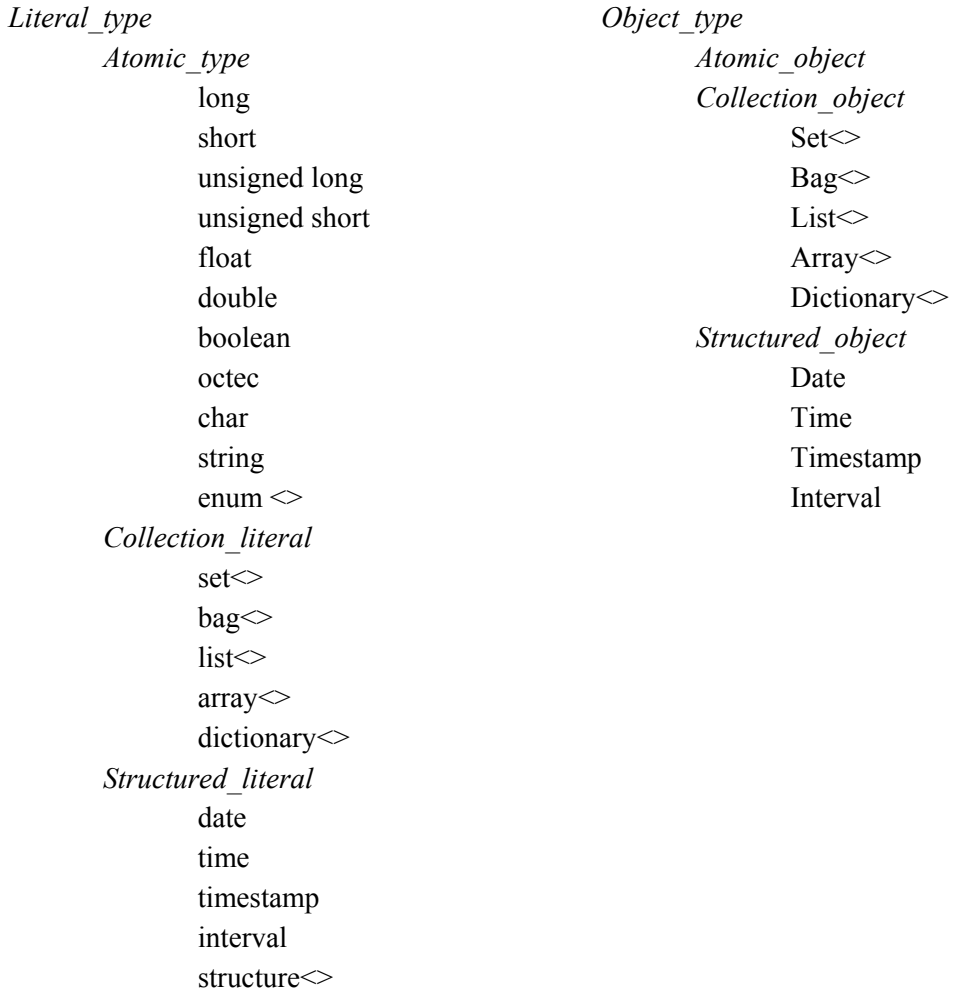


Figura 3.3: Hierarquia de tipos primários do modelo ODMG

Modelização do Estado

São definidos dois tipos de propriedades no modelo ODMG/OM:

- Atributos: numa interface, a declaração do atributo define o estado abstracto do tipo. O valor de um atributo é sempre ou um literal ou um identificador de objecto. Os atributos podem ser representados por estruturas de dados ou, em determinados casos, por métodos. Os atributos não são "cidadãos de primeira classe" visto que o atributo não é um objecto, nem é possível definir atributos, relacionamentos e operações específicas para o próprio atributo;
- Relacionamentos: os relacionamentos são definidos entre tipos, sendo suportados em ODMG/OM apenas relacionamentos binários. Estes podem ser um-para-um, um-para-muitos e muitos-para-muitos. Tal como os atributos, também os relacionamentos não são "cidadãos de primeira classe". Um relacionamento é definido implicitamente através da declaração de conexões lógicas entre os dois objectos que nele participam as quais funcionam como caminhos de travessia de um objecto para outro. Estas conexões são

declaradas em pares, em cada objecto, para suportar ambas as direcções e ligadas sintacticamente na sua declaração usando a palavra inverso - *inverse*. O SGBDOO é responsável por manter a integridade referencial dos relacionamentos. Se um objecto for removido, todos os caminhos de travessia de outros objectos para esse objecto têm de ser igualmente removidos. Um caminho de travessia pode ter, em cada extremidade, cardinalidade "um", indicando apenas um objecto, ou "muitos", indicando uma colecção de objectos. Referências unidireccionais de um objecto para outro, através de atributos objecto não são considerados verdadeiros relacionamentos, à luz do modelo ODMG/OM.

Exemplo 3.3: utilização de atributos e relacionamentos. Declaração e representação gráfica.

```
Interface Prateleira {
relationship set<Uinstalação> contém_as
  inverse Uinstalação::localizada_em;
relationship Estante parte_da
  inverse Estante::composta_pelas;
attribute short Numero;
attribute short Altura; }

Interface Estante {
relationship set<Prateleira> composta_pelas
  inverse Prateleira:: parte_da;
attribute short Numero;
attribute short Comprimento; }

Interface Udescrição {
relationship set<Uinstalação> estão_em
  inverse Uinstalação::possui_as;
attribute string CodReferência;
attribute string Titulo;
attribute string Notas;
attribute Tipo_Quantidade Quantidade;
attribute Tipo_Quantidade QuantidadeSobCustodia;
attribute Date DataProdução; }

Interface Uinstalação{
relationship set<Udescrição> possui_as
  inverse Uinstalação:: estão_em;
relationship Prateleira localizada_em
  inverse Prateleira:: contém_as;
attribute string Referência;
attribute string Tipo;
attribute short ExtensãoLinear; }
```

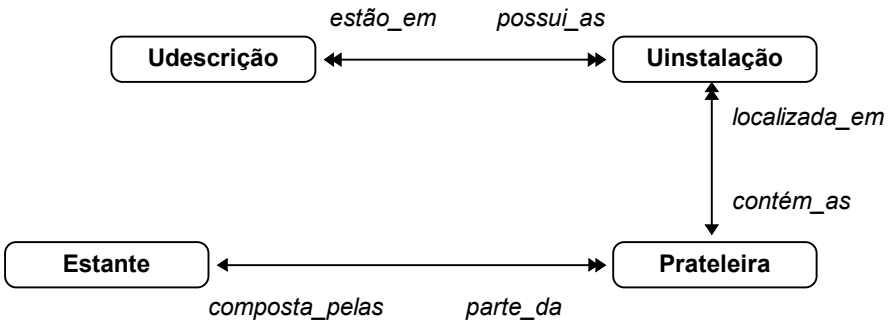


Figura 3.4: Diagrama de relacionamentos

Modelização do Comportamento

Outra característica importante de um tipo é o seu comportamento, especificado por um conjunto de *assinaturas de operações*. Cada assinatura define o nome da operação, o nome e tipo dos seus argumentos, o tipo dos valores retornados e os nomes das *excepções*, ou condições de erro, que a aplicação possa provocar. Uma operação é sempre definida em apenas um único tipo, sendo o seu nome único dentro desse tipo. Diferentes tipos, contudo, podem ter operações com o mesmo nome, situação designada por *sobrecarga*. O mecanismo de selecção da operação a ser executada, quando na presença de sobrecarga, é denominado *resolução de nomes da operação*, e privilegia o tipo mais específico.

As excepções são consideradas objectos do ODMG/OM, possuindo uma interface e hierarquia de excepções. É assim possível a definição de tipos de excepções a partir de um tipo raiz *Excepção*.

Bloqueios e Controlo de concorrência

O ODMG/OM define um mecanismo para reforçar o acesso partilhado ou exclusivo a objectos. O SGBDOO, garante a seriabilidade das transacções, monitorizando requisições para bloqueios e autorizando-os caso não exista nenhum conflito. Consegue-se assim a coordenação entre múltiplas transacções e a manutenção de uma vista consistente para cada transacção.

O ODMG/OM suporta uma política pessimista de controlo de concorrência por defeito.

São suportados bloqueios para leitura (partilhada), escrita (exclusiva) e actualização. Este último destina-se a evitar situações de *deadlock* quando dois processos que obtiveram bloqueios para leitura no mesmo objecto, estão ambos a tentar obter bloqueios para escrita. Assim, se um processo quiser efectuar operações de leitura seguido de operações de escrita em objectos, primeiro requer um bloqueio de actualização enquanto efectua leituras, sendo este partilhado apenas com bloqueios de leitura, e depois requer bloqueio de escrita.

Os bloqueios podem ser implícitos, quando se efectuam operações de leitura ou manipulação, ou explicitamente pedidos. Os bloqueios de actualização são sempre explícitos.

Os bloqueios persistem enquanto persistir a transacção, ou seja, até que esta seja confirmada ou abortada.

Meta-informação

Meta-informação é a informação descritiva acerca dos objectos da base de dados que definem o esquema da base de dados. A meta-informação é guardada num *repositório de esquema ODL*, sendo equivalente ao *repositório de Interface IDL* existente em ambientes OMG CORBA. É usada pelo SGBDOO para definir a estrutura da base de dados e para fornecer informação sobre a mesma. Está acessível a ferramentas e aplicações que necessitem de conhecer a estrutura da base de dados em tempo de execução.

A definição da estrutura interna de um repositório de esquema ODL recorre a um conjunto de interfaces definidas também em ODL. Estas usam relacionamentos para definir o grafo de interligações entre meta objectos, produzidos, por exemplo, durante a fase de compilação do código fonte ODL. Esses relacionamentos garantem a integridade referencial do grafo de meta objectos. Contudo, não conseguem garantir a integridade semântica.

As definições de meta objectos estão agrupadas num módulo, que assim define um contexto de nomes para os elementos do modelo. Este designa-se por *ODLMetaObjects*.

As interfaces consideradas, incluem operações que permitem a definição de operações para construção de esquemas válidos. Entre estas temos operações de criação, adição e remoção, permitindo a ligação e desvinculação dos relacionamentos requeridos. São também definidos procedimentos de recuperação de erros no caso de acontecerem erros de natureza semântica.

A interface de contexto:

```
interface Scope { ... };
```

Define uma hierarquia de nomes para os meta-objectos no repositório, suportando operações de ligação - *bind()* para adição de objectos, de *resolve()* para retornar meta-objectos por nome, e *un_bind()* para remover ligações de meta-objectos à hierarquia.

Todos os objectos no repositório são subclasses das interfaces: *MetaObject*, *Specifier* e *Operand*. Todos os meta-objectos (subtipos de *MetaObject*) têm um atributo nome e um atributo comentário, e participam num relacionamento simples juntamente com outros meta-objectos com a interface *DefiningScope* (subtipo de *Scope*). Esta interface representa a definição de um contexto que contém meta-objectos e operações para criar, adicionar e remover meta-objectos neles próprios.

A interface MetaObject é definida do seguinte modo:

```
interface MetaObject {
attribute    string name;
attribute    string comment;
relationship DefiningScope definedIn
                  inverse DefiningScope::defines;
};
```

A hierarquia de interfaces do repositório encontra-se recriada na figura seguinte:

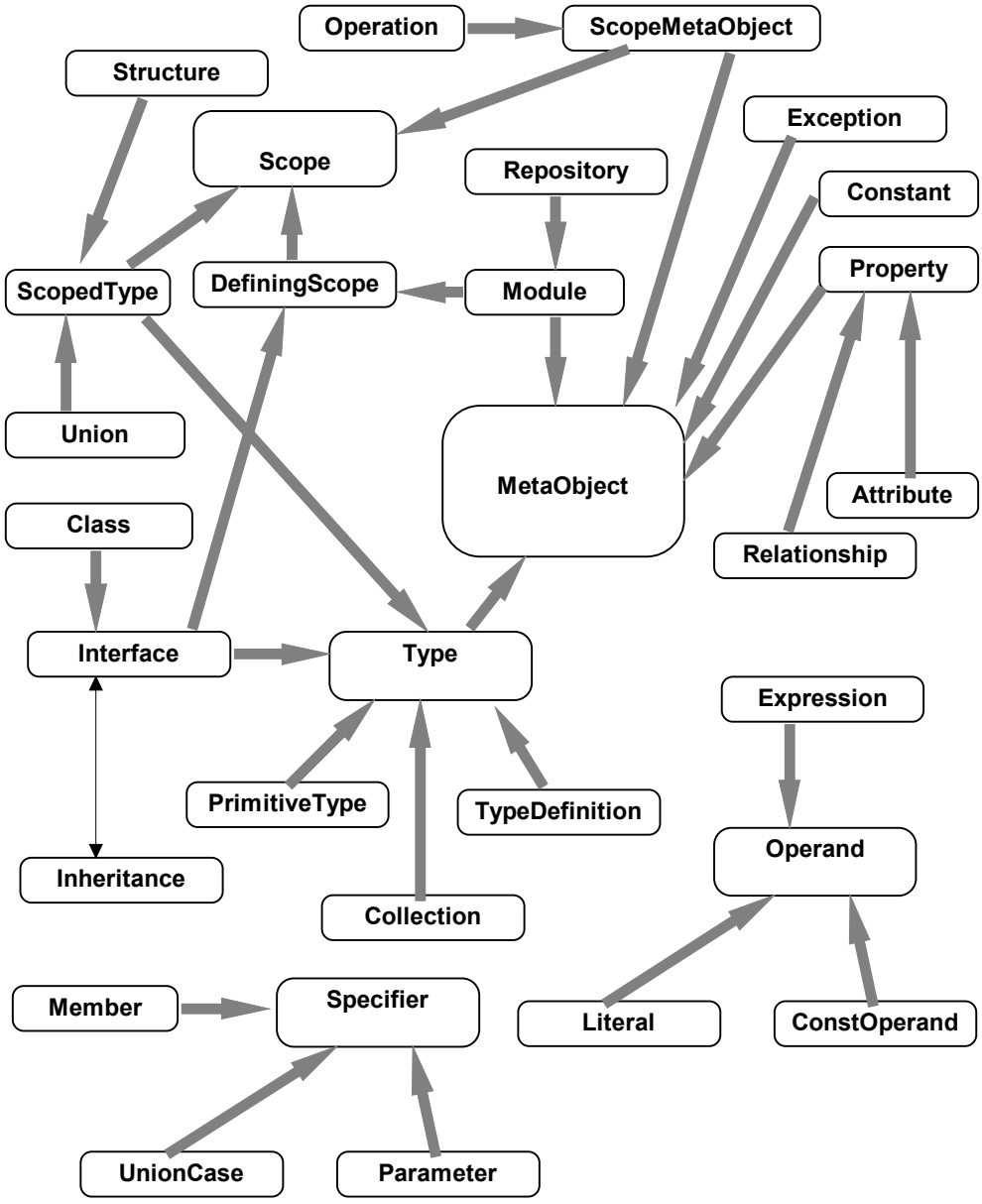


Figura 3.5: Interfaces da Meta-informação

Modelo de Transacções

O uso de objectos persistentes está associado a transacções. A gestão de transacções é uma funcionalidade importante para assegurar integridade, partilha e recuperação da base de dados. Qualquer acesso, criação, modificação e remoção de objectos persistentes deve ser feito dentro de uma transacção. Esta funciona como uma unidade lógica para a qual o SGBDOO garante as propriedades ACID: atomicidade, consistência, isolamento e durabilidade.

Os objectos temporários não estão sujeitos à semântica de transacção. Logo quando uma transacção é abortada, os objectos temporários não voltam ao estado anterior à transacção como acontece com os objectos persistentes.

Transacções Distribuídas: são transacções que se espalham por múltiplos processos ou múltiplas bases de dados. O modelo ODMG/OM exige que estas transacções sejam em conformidade com o *Object Transaction Service* da OMG e ISO XA.

Transacções e Processos: quando um processo possui vários fios de execução - *threads*, existe exactamente uma transacção corrente por cada fio de execução sendo essa transacção implícita às operações executadas por esse *thread* sobre a base de dados. É garantido o isolamento de transacções entre *threads*.

Uma transacção aplica-se a uma base de dados lógica, que possivelmente estará distribuída fisicamente pela rede.

Operações em Transacções: a criação de um objecto *Transacção* faz-se usando a operação *new()* da interface *TransactionFactory* e a operação *current()* averigua se existe alguma transacção associada ao processo ou *thread*.

Um objecto Transacção permite efectuar as operações de abertura da transacção - *begin()*, confirmação - *commit()*, aborto da transacção - *abort()*, e confirmação a meio da transacção com sua continuação - *checkpoint()*. É possível a existência de mais do que um objecto transacção por *thread*, usando-se as operações de junção - *join()* e abandono - *leave()* para o *thread* alternar entre cada transacção. É ainda possível que vários *threads* que partilham o mesmo espaço de endereçamento partilhem também o mesmo objecto transacção através de operações de junção múltiplas. Neste caso, não existe nenhuma operação de bloqueio entre esses *threads* e o controlo de concorrência fica a cargo do programador.

Operações na Base de Dados

Cada SGBDOO pode gerir uma ou várias bases de dados lógicas, instâncias do tipo Base de Dados - *Database*. A criação de um objecto *Database* faz-se usando a operação *new()* da interface *DatabaseFactory*. Para ser possível o acesso à base de dados, esta tem de ser aberta com a respectiva operação - *open()*, recorrendo-se à operação de fecho - *close()* quando o acesso deixa de ser necessário.

A associação de nomes com objectos da base de dados, tendo esta como contexto, é possível usando a operação de ligação - *bind()*. Para quebrar essa ligação é usada a operação *unbind()*. Estes nomes servem geralmente como pontos de entrada, sendo possível aceder ao identificador de objecto pelo seu nome através de uma operação de *lookup()*. Estes nomes frequentemente representam extensões de tipos, tornando mais fácil o acesso à colecção de objectos desse tipo.

3.2.2 Linguagem de Definição de Objectos - ODL

A linguagem de definição de objectos (*Object Definition Language - ODL*) é usada para definir a especificação dos tipos dos objectos que estão de acordo com o modelo ODMG/OM. A ODL é utilizada para suporte de portabilidade de esquemas de bases de dados em SGBDOOs, representando um avanço na interoperabilidade entre múltiplos SGBDOOs.

Os principais aspectos considerados no desenvolvimento desta linguagem foram:

- garantir o suporte de todas as construções semânticas do modelo;
- não dever ser uma linguagem de programação, mas antes uma linguagem de definição para especificação de objectos;

- ser independente de qualquer linguagem de programação de modo a que possa ser usada como uma linguagem de especificação de esquemas para serem implementados em várias linguagens como C++, Java ou Smalltalk;
- ser compatível com a linguagem de definição de interfaces (*Interface Definition Language - IDL*) da OMG. Visto esta ser influenciada pela linguagem C++, essa influência transitou também para a ODL;
- ser extensível, não apenas para suporte de funcionalidades futuras, mas também para optimizações físicas;
- ser prática, de forma a atrair para o seu uso quem está a desenvolver, e a ser suportada pelos fabricantes de SGBDOOs em tempo útil;
- fornecer um contexto de integração de esquemas oriundos de múltiplas fontes e aplicações, usando diferentes modelos orientado por objectos desenvolvidos. Entre esses temos os casos dos standards STEP/PDES (EXPRESS), ANSI X3H2 (SQL), ANSI X3H7 (Object Information Management), CFI (CAD Framework Initiative) e outros (ver figura abaixo).

Os SGBDs tradicionalmente suportam uma linguagem de definição e uma linguagem de manipulação de dados. Enquanto a primeira permite que os utilizadores definam os seus tipos de dados e interfaces, a segunda é usada para criação, leitura, alteração, remoção e outras operações ao nível de instâncias desses tipos de dados. A ODL é uma linguagem de definição de dados para tipos objecto, permitindo definir as características dos tipos incluindo as suas propriedades e operações.

Um esquema definido em ODL é suportado por qualquer SGBDOO em conformidade com o modelo ODMG e por um conjunto de linguagens de programação. Este grau de compatibilidade é necessário para que uma aplicação possa ser executada com o mínimo de modificações em diversos SGBDOOs. Outro grau de compatibilidade requerido é que uma base de dados criada numa linguagem possa ser acedida por uma aplicação escrita noutra linguagem. ODL oferece assim um grau de isolamento contra as variações existentes quer em linguagens de programação quer em diferentes SGBDOs.

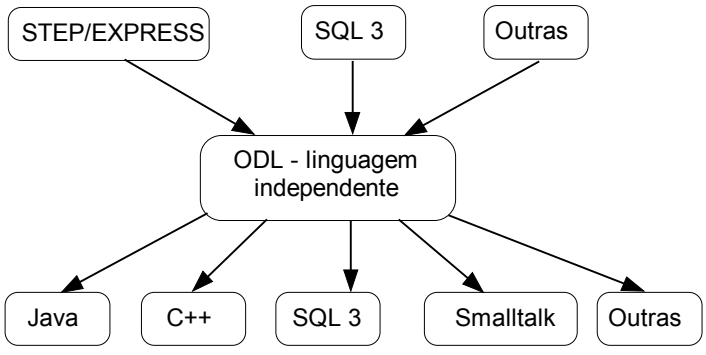


Figura 3.6: Mapeamento de ODL para outras linguagens

Exemplo 3.4: Especificação de interfaces de Detentores em ODL e respectivo diagrama OMT.

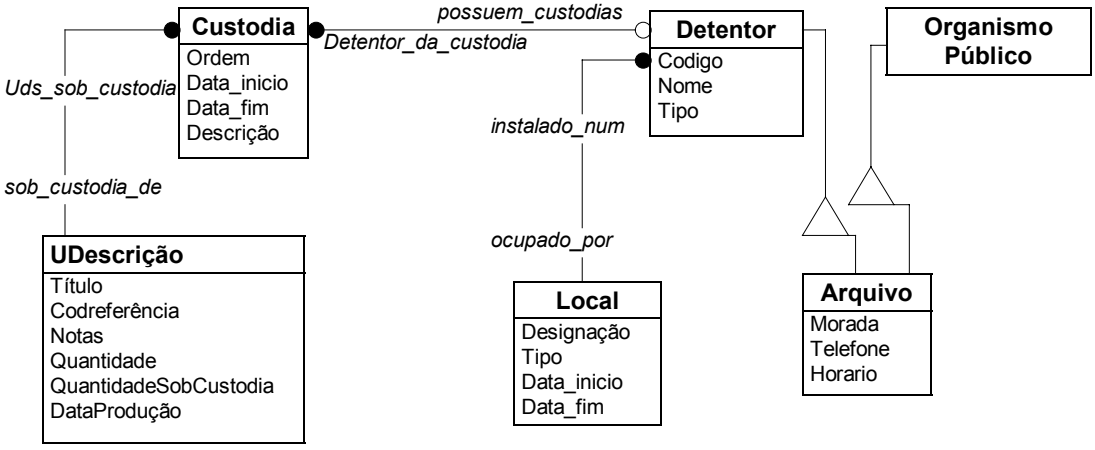


Figura 3.7: Diagrama OMT para Detentores

A definição em ODL da especificação do modelo OMT vem:

```
class Udescrição
( extent Udescrições )  {
struct S_Quantidade {string descrição, unsigned short numero};
relationship set<Custodia> sob_custodia_de
  inverse Custodia::Ud_sob_custodia;
attribute string CodReferência;
attribute string Titulo;
attribute string Notas;
attribute S_Quantidade Quantidade;
attribute S_Quantidade QuantidadeSobCustodia;
attribute Date DataProdução; };

interface Organismo_Público  {
attribute string Tutela;
string Consulta_Tutela();
string Altera_Tutela(in string nova_tutela); };

class Custodia
( extent Custodias )    {
relationship UDescrição Ud_sob_custodia
  inverse UDescrição:: sob_custodia_de;
relationship set<Detentor> detentores_da_custodia
  inverse Detentor::possui_custodias_de;
attribute string Ordem;
```

```
    attribute string Descrição;
    attribute Date Data_inicio;
    attribute Date Data_fim;     };

    class Detentor
    ( extent Detentores )   {
    relationship set<Custodia> possui_custodias_de
      inverse Custodia::detentores_da_custodia;
    relationship Local instalado_em
      inverse Local::ocupado_por;
    attribute string Codigo;
    attribute string Nome;
    attribute string Tipo;  };

    class Arquivo extends Detentor : Organismo_Público
    ( extent Arquivos )     {
    struct  S_Horario  {unsigned  short  ent_manha,  unsigned  short
sai_manha, unsigned short ent_tarde, unsigned short sai_tarde};
    attribute string Tutela;
    attribute string Morada;
    attribute string Telefone;
    attribute S_Horario Horario; };

    class Local
    ( extent Locais ) {
    relationship set<Detentor> ocupado_por
      inverse Detentor::instalado_em;
    attribute string Ordem;
    attribute string Tipo;
    attribute Date Data_inicio;
    attribute Date Data_fim;     };
```

□

3.2.3 Linguagem de Especificação OIF - *Object Interchange Format*

O objectivo da linguagem para *Formatação da Troca de Objectos* é a especificação de um formato para troca de informação entre um ficheiro e uma base de dados objecto, em ambos os sentidos. Pode também ser usada para efectuar a troca de objectos entre bases de dados, permitir gerar documentação sobre a base de dados, e efectuar conjuntos de testes a bases de dados.

Os princípios subjacentes à elaboração da linguagem OIF foram:

- OIF deve suportar todos os estados possíveis para as bases de dados em conformidade com o modelo ODMG e com as definições de esquema permitidas em ODL;

- não deve constituir uma linguagem de programação, mas sim uma linguagem de especificação para objectos persistentes e seus estados;
- deve ser projectada de acordo com as normas STEP e ANSI sempre que possível;
- não necessita de outras palavras chave para além dos identificadores tipo, atributo e relacionamento existentes nas definições de esquema de bases de dados elaborados em ODL.

A caracterização do estado de todos os objectos contidos na base de dados contempla identificadores de objectos, ligações a tipos, valores dos atributos e ligações com outros objectos, estando cada um destes itens especificado em OIF.

Um ficheiro OIF contém definições de objectos. Cada uma delas especifica o tipo, valor dos atributos e relacionamentos desses objectos para outros objectos. Os identificadores de objectos são especificados com nomes de objectos únicos para o ficheiro ou ficheiros OIF. Não são assim necessárias mais declarações do objecto, sendo este identificador visível em todo o conjunto de ficheiros OIF.

Um exemplo simples de definição de objecto é:

```
ADPorto Arquivo{}
```

em que é criada uma instância da classe Arquivo. Os valores dos atributos deste objecto não estão inicializados, sendo o seu nome `ADPorto` usado para referir o objecto no conjunto de ficheiros OIF.

Se se pretendesse criar uma nova instância persistente - `FundoCMPorto2` da classe *UDescrição* que ficasse localizada fisicamente perto[15] da instância anterior - `FundoCMPorto1` poderíamos defini-la:

```
FundoCMPorto2 (FundoCMPorto1) UDescrição{}
```

Inicialização de Atributos com Valores

Para criação de um novo objecto de uma dada classe com valores iniciais, fornece-se dentro de chavetas uma lista de pares <nome do atributo> <valor do atributo>.

Considerando que os atributos Quantidade e QuantidadeSobCustodia são atributos do tipo estruturado Tquantidade, e DataProdução do tipo estruturado Tdata definidos como:

```
struct Tquantidade {
        string          descrição;
        unsigned short  numero;
};

struct Tdata {
        unsigned short  dia;
        unsigned short  mês;
        unsigned short  ano;
};
```

poderia ter-se uma nova instância de UDescrição definida como:

[15] conceito dependente da implementação

```
    FundoCNPorto01 UDescrição{Titulo "escritura",
Codreferência "CNPT01", Notas "escrituras diversas",
Quantidade{descrição "microfilmes", numero 10},
QuantidadeSobCustodia{ descrição "microfilmes", numero 9},
DataProdução{ asValue{ month 1, day 1,year 1850}}}
```

Pode-se aqui constatar que estruturas dentro de objectos são referidas colocando o nome do atributo estruturado, caso do atributo Quantidade, e dentro, entre chavetas, a lista de pares <nome do atributo> <valor do atributo>.

Para inicialização um objecto por cópia de outro objecto temr-se:

```
FundoCNPorto02 UDescrição{FundoCNPorto01}
```

que cria um novo objecto com o nome `FundoCNPorto02` e cujos valores são copiados do objecto `FundoCNPorto01`.

Tipos lógicos podem ter os literais FALSE e TRUE. Atributos que representam números reais são inicializados com um literal composto por: <sinal negativo opcional> <parte inteira> **.** <parte fraccionária> **e** ou **E** <expoente negativo opcional>

Exemplos de números reais representados com este formato são: 26.0, -12.05, 5E-3, .04, 12.001e-2.

Na representação de vectores, por exemplo o vector `attribute unsigned short Idade[10];` dentro do objecto `Pessoas`, tem-se:

```
TurmaA Pessoas{ Idade{[0] 18, [3] 16}}
```

os índices ficam entre parênteses rectos e o valor vem a seguir. Na ausência de índices, os valores são associados aos primeiros índices. As colecções têm um tratamento idêntico. Se for uma colecção ordenada, como um vector de objectos, tem-se uma lista de pares <índice do objecto> <valor do objecto>.

Definição de Relacionamentos

Relacionamentos entre objectos são indicados recorrendo ao nome do relacionamento seguido do nome do objecto relacionado. Um relacionamento da classe *Custodia* com a classe *UDescrição*, poderia ser especificado em OIF do seguinte modo:

```
Custodia01 Custodia{ Ud_sob_Custódia FundoCNPorto01}
```

o exemplo acima descreve um relacionamento de cardinalidade um entre `Custodia01` e `FundoCNPorto01`. Para relacionamentos de cardinalidade muitos tem-se:

```
FundoCNPorto01 UDescrição{ sob_custodia_de{ Custodia01,
Custodia02, Custodia03 }
```

Comandos para OIF

A linguagem define ainda comandos para criar um ficheiro OIF a partir de uma base de dados:

```
odbdump <nome da base de dados>
```

e para criar uma base de dados a partir de um ou mais ficheiros OIF:

```
odbload <nome base de dados> <ficheiro 1> ... <ficheiro n>
```

3.2.4 Linguagem de Interrogação de Objectos

A linguagem de interrogação de objectos - OQL (*Object Query Language*), suporta o modelo ODMG/OM, lidando com objectos complexos.

Os princípios que balizaram a sua elaboração, e as suas principais características são:

- a OQL tem como base o modelo ODMG/OM;
- a OQL tem semelhanças com SQL92. As extensões a esta última linguagem têm a ver com conceitos de orientação por objectos como sejam objectos complexos, identidade dos objectos, expressões de navegação, polimorfismo, invocação de operações e ligação dinâmica;
- permite primitivas de alto nível para lidar com conjuntos; contudo, não se restringe a construções de conjuntos; tem também primitivas para tratamento de estruturas, listas e vectores com eficiência semelhante a conjuntos;
- a OQL é uma linguagem funcional onde os operadores podem ser compostos livremente, desde que os operandos respeitem o sistema de tipos. Isto advém do facto de o resultado de qualquer interrogação ter um tipo que pertence ao modelo de tipos da ODMG, podendo portanto ser aplicado noutra interrogação;
- a OQL não é uma linguagem computacionalmente completa. É uma linguagem de interrogação de uso simples que permite o acesso ao SGBDOO;
- a OQL pode ser invocada dentro de uma linguagem de programação para a qual ODMG/OM tenha definida ligação ao modelo, pois é baseada no mesmo sistema de tipos. O contrário também pode acontecer, ou seja, OQL invocar operações nessas linguagens de programação;
- a OQL não permite operações explícitas de actualização mas antes invoca operações dos objectos com essa finalidade. Deste modo não viola a semântica de um SGBDOO que por definição exige que os métodos de acesso aos atributos sejam definidos nos próprios objectos;
- a linguagem permite acesso declarativo aos objectos. É assim possível a realização de optimizações na implementação das interrogações;
- a semântica formal de OQL pode ser facilmente definida.

Argumentos e Resultados de uma Interrogação

Usada isoladamente, OQL permite a interrogação com base nos nomes dos objectos que servem assim como pontos de entrada na base de dados. Estes nomes podem denotar qualquer tipo de objectos.

Como linguagem embutida, permite a interrogação de objectos denotáveis que são suportados pela linguagem nativa através de expressões contendo átomos, estruturas, colecções e literais. Uma interrogação OQL é uma função que tem como resultado um objecto cujo tipo pode ser inferido do operador que contribui para a expressão de interrogação.

Nos exemplos que se seguem, é utilizado o esquema definido anteriormente em ODL. Neste esquema *Udescrições* é a extensão da classe *UDescrição*.

Exemplo 3.5: retornar os títulos e as notas de todas as unidades de descrição que têm como valor para o atributo notas ”escrituras diversas”. O resultado da interrogação vai ser um literal que representa um conjunto de estruturas do tipo `set<struct>`.

```
select distinct struct(tit: u.titulo, not: u.notas)
from UDescrições u
where u.notas = "escrituras diversas"
```

□

É possível definir funções mais complexas, podendo usar expressões *select-from-where* na cláusula *select*. Para navegação em estruturas complexas, ou seguimento de relacionamentos, a OQL usa o “.” ou “→“ indiferentemente para separação dos campos.

Exemplo 3.6: obter para cada unidade de descrição, o título e o conjunto de custódias cuja data inicial de custódia seja posterior a 1800 e a data final anterior a 1900. O resultado da interrogação vai ser um literal que representa um conjunto de estruturas do tipo `set<struct(tit: string, custs: bag<Custodias>)>`.

```
select distinct struct(tit: u.titulo, custs: (select c
                  from u.sob_custodia_de as c
                  where (c.data_inicio.asValue.year > 1800
                  and
                  c.data_fim.asValue.year < 1900))
from UDescrições u
```

□

É também possível usar expressões *select-from-where* na cláusula *from*.

Exemplo 3.7: retornar para cada unidade de descrição, com data de produção posterior a 1850 e notas “escrituras diversas”, o título e código de referência.

```
select distinct struct(tit: u.titulo, codref: u.codreferência)
from    (select v from Udescrições v where
        v.dataprodução.asValue.year > 1850) as u
where u.notas = "escrituras diversas"
```

□

Como em SQL, a expressão *select-from-where* permite interrogar mais do que uma colecção, podendo estas aparecerem na cláusula *from*. É também possível, tal como em SQL, aplicar funções de agregação.

Exemplo 3.8: listar a descrição de todas as custódias de cada unidade de descrição para as unidades de descrição que estão sob mais do que uma custódia.

```
select c.descrição
from   Udescrições u, u.sob_custodia_de as c
```

```
where count(u.sob_custodia_de) >= 2
```

Operações de junção entre classes não directamente relacionadas também são possíveis de efectuar em OQL.

Exemplo 3.9: retornar o título de todas as unidades de descrição cujo ano de produção seja igual ao ano de início de instalação de um Detentor num Local, sabendo que Locais é extensão da classe Local.

```
select u.titulo
from   Udescrições u, Locais l
where u.dataprodução.asValue.year = l.data_inicio.asValue.year
```

Identidade de Objectos

A OQL fornece suporte para retornar quer objectos quer literais, dependendo do modo como os objectos são construídos ou seleccionados.

Para criar um objecto com identidade, é necessário usar o nome de um construtor de tipo. Para criar, por exemplo um objecto da classe Detentor, poderia ser:

`Detentor(Codigo: "ADPRT", Nome: "Arquivo Distrital do Porto", Tipo: "Arquivo")`

em que os parâmetros dentro de parênteses permitem efectuar a inicialização de certas propriedades do objecto.

As expressões de extracção podem devolver:

- colecções de objectos como por exemplo: `select u from Udescrições u;`
- um objecto com identidade, por exemplo: `select u from Udescrições u where codreferência = "CNPT01";`
- uma colecção de literais como nos exemplos já vistos;
- um literal como por exemplo: `select u.titulo from Udescrições u where codreferência = "CNPT01".`

Valores Nulos

O resultado de aceder a uma propriedade de um objecto nulo - *nil* - é indefinido. As regras OQL, para lidar com um resultado indefinido são:

- as operações de acesso a propriedades "." ou "" aplicadas a um operando indefinido do lado esquerdo produzem um resultado indefinido;
- operações de comparação (=, !=, <, >, <=, >=) com um ou os dois operandos indefinidos, produzem resultado de valor lógico falso;
- a função *is_undefined*(valor indefinido) retorna verdadeiro. A função *is_defined*(valor indefinido) retorna falso;

- qualquer outra operação com valores indefinidos como operandos resulta num erro de execução.

Invocação de Métodos

OQL permite a invocação de métodos com ou sem parâmetros em qualquer lugar onde o tipo do resultado do método esteja de acordo com o tipo esperado na interrogação. A notação para invocação do método é a mesma que para aceder a um atributo ou a um relacionamento, no caso deste não ter parâmetros. Caso tenha parâmetros, devem ser passados entre parênteses. Esta flexibilidade torna transparente para o utilizador se a interrogação está a aceder a uma propriedade ou a um método que retorna um valor.

Polimorfismo

Quando uma colecção polimórfica (contendo instâncias de uma classe e instâncias de suas subclasses) é filtrada numa interrogação OQL, os seus elementos são estaticamente reconhecidos como sendo da superclasse. Significa isto que uma propriedade de uma subclasse não pode ser aplicada a esses elementos, excepto em dois casos: ligação dinâmica a um método ou indicação explícita da classe.

Composição de operadores

OQL é uma linguagem puramente funcional. Todos os operadores podem ser compostos livremente desde que o sistema de tipos seja respeitado. Esta filosofia de ortogonalidade é diferente de SQL onde as regras de composição não são ortogonais.

Além dos operadores OQL já focados, existem também operadores de reunião, intersecção e excepção (*union*, *intersect*, *except*), quantificadores universais (*for all*) e existenciais (*exists*), operadores de ordenação (*order by*), de definição de partições (*group by*) e de agregação (*count*, *sum*, *min*, *max* e *avg*).

Exemplo 3.10: retornar o ano de produção de todas as unidades de descrição com atributo notas contendo "escrituras diversas" e cuja média do número de unidades sob custódia (atributo quantidade.numero) seja a mais baixa.

Pode-se proceder à interrogação em várias etapas, guardando o resultado das interrogações intermédias usando *define*:

```
Define UD_not_escdiv() as
  select u from Udescrições u
  where u.notas = "escrituras diversas"

Define Por_anos() as
  select ano, quant_media: avg(select q.quantidade.numero
                                  from partition q)
  from UD_not_escdiv() u
  group by ano: u.dataprodução.asValue.year
```

```
Define Ord_por_anos() as
  select p from Por_anos() p order by p.quant_media
```

ou usar apenas uma interrogação:

```
first(
select ano, quant_media: avg(select q.quantidade.numero from
partition q)
from ( select p from Udescrições p where p.notas = "escrituras
diversas") as u
group by ano: u.dataprodução.asValue.year
order by quant_media).ano
```

□

3.2.5 Ligação às Linguagens de Programação

A ligação do modelo ODMG/OM às linguagens de programação orientadas por objectos C++, Smalltalk e Java, é efectuada definindo um mapeamento da linguagem de definição ODL para a linguagem de programação respectiva. É também definida uma linguagem de manipulação OML (*object manipulation language*), com sintaxe e semântica da linguagem de programação, que permite o acesso aos objectos da base de dados e sua modificação.

O conjunto linguagem de definição e linguagem de manipulação definem apenas características lógicas dos objectos e as operações usadas para os manipular. Não discutem o armazenamento físico dos objectos, nem os aglomerados (*clusters*) de objectos na memória secundária, ou questões de gestão de memória associadas com a representação física dos objectos ou estruturas de acesso como índices para acelerar o acesso a objectos da base de dados.

Princípios que acompanharam o projecto de ligação do modelo ODMG/OM às linguagens de programação:

- a coexistência das duas linguagens deve ser harmoniosa, e deve ser o mais transparente possível para o programador;
- existe apenas um sistema de tipos entre a linguagem de programação e a base de dados. Instâncias individuais desses tipos comuns podem ser ou persistentes ou temporárias;
- no caso de Smalltalk, os conceitos ODL devem ser representados usando convenções definidas para esta linguagem, devendo a ligação respeitar o facto de Smalltalk ter tipagem dinâmica;
- O facto de Smalltalk e Java terem gestão automática de memória, faz com que os objectos fiquem persistentes quando são referenciados por outros objectos persistentes na base de dados, e que sejam removidos quando já não sejam atingíveis desta maneira.

Ligações às Linguagens

Linguagem C++:

A ligação de C++ ao modelo ODMG/OM é efectuada introduzindo um conjunto de classes que pode ter instâncias persistentes ou temporárias. Estas classes são definidas como subclasses

da classe *d_Object* que fornece sobrecarga do operador *new()* e *delete()* para criar e remover instâncias persistentes. A manipulação de objectos persistentes é efectuada usando referências para os objectos. Para cada subclasse *T* de *d_Object*, é definida uma classe auxiliar *d_Ref<T>*. Instâncias da classe *T* são referenciadas usando referências parametrizadas:

```
d_Ref<T>      ref_para_T;
```

Os passos envolvidos na geração de uma aplicação em C++, estão ilustrados na figura abaixo.

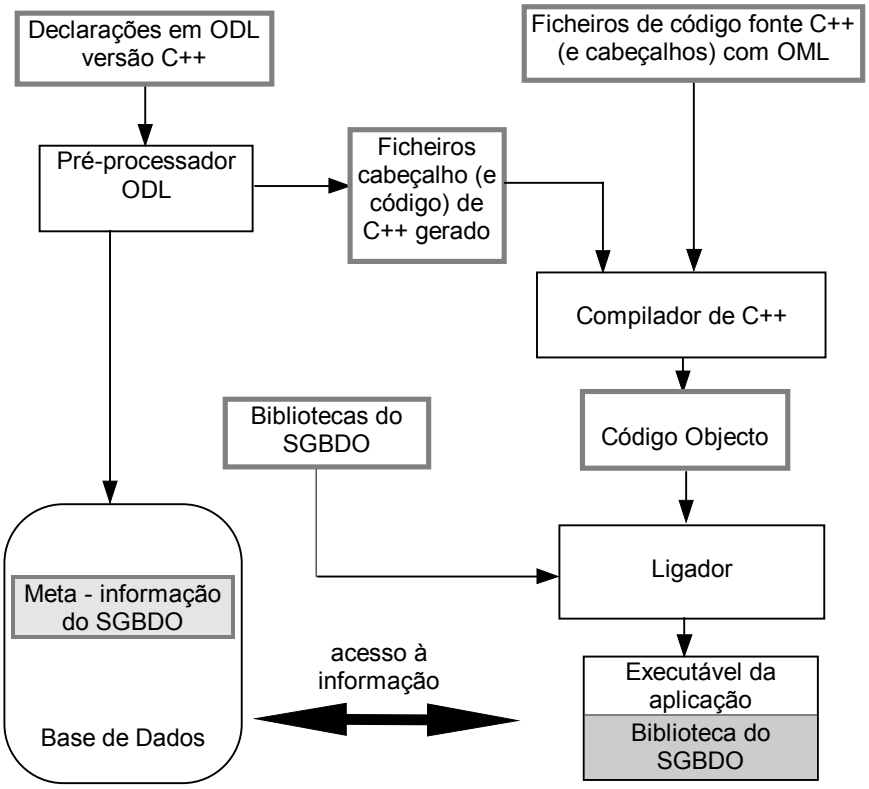


Figura 3.8: Passos para construção de uma aplicação C++ para um SGBDOO.

Linguagem Smalltalk:

A ligação entre ODMG/OM e Smalltalk é baseada na ligação entre Smalltalk e IDL, definida pela OMG. Esta pode ser automatizada por um compilador de ODL que processa as declarações ODL e gera um grafo de meta-objectos, que modelizam o esquema da base de dados. Estes meta-objectos, disponibilizam a informação de tipos que permitem à ligação do Smalltalk suportar a semântica requerida pelo sistema de tipos ODL. O conjunto completo dos meta-objectos define o esquema da base de dados e comporta-se como um repositório de esquema ODL. Nesse repositório, os meta-objectos podem ser acedidos e modificados pelas aplicações em Smalltalk através da sua interface standard. Uma aplicação designada por gerador de ligações (*binding generator*) soluciona as escolhas de mapeamento tipo/classe inerentes à ligação de ODMG/OM a Smalltalk. A informação nos meta-objectos é suficiente para regenerar as declarações de ODL para as porções de esquema que representam. As relações entre esses componentes estão ilustradas na figura abaixo.

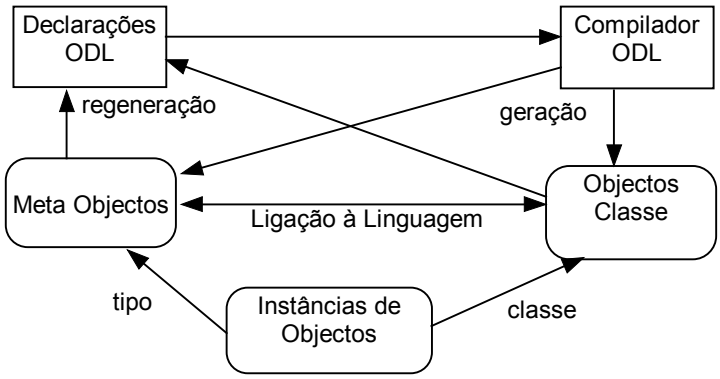


Figura 3.9: Ligação de ODMG à Linguagem Smalltalk

Linguagem Java:
Java oferece dois modos de declarar classes capazes de se tornarem persistentes:

- classes já existentes podem ser transformadas em capazes de ser persistentes;
- declarações de classes Java podem automaticamente ser geradas (assim como o esquema da base de dados) por um preprocessador para ODMG/ODL.

Implementações possíveis seriam:

- um pós-processador que teria como entrada o ficheiro de Java `.class` (em *bytecode*) produzido pelo compilador de Java e produzia um novo ficheiro *bytecode* que suportasse persistência;
- um pré-processador que modificaria o código fonte Java antes de este ser compilado pelo compilador de Java;
- um interpretador de Java modificado.

A ligação ODMG para Java deve suportar todas estas implementações possíveis.

Mapeamento do modelo ODMG/OM para as linguagens

Na tabela seguinte faz-se uma súmula de como os conceitos do modelo ODMG/OM se traduzem para as linguagens concretas.

ODMG/OM	**C++**	**Smalltalk**	**Java**
Objectos	Classe em C++.	Classe em Smalltalk	Tipo Objecto Java
Literais	Classe embutida dentro de outra classe	Classe em Smalltalk	Tipo primitivo Java
Estruturas	Estrutura em C++ ou classe embutida dentro de outra classe	Classe em Smalltalk	Classe Java
Implementação	Classe C++ tem 2 partes: 1. pública ou interface. privada ou implementação (só possível uma implementação)	Tudo implementado como objecto. Variáveis de instância dos objectos são privadas à implementação dos seus métodos.	Java suporta a noção de interface independente da implementação. Interfaces não são instanciáveis.
Tipos geradores de Colecções	Classes padrão (*template classes*)	-	-
Tipos de Colecções	Classes colecção	Smalltalk tem variedade de classes na hierarquia colecção como conjuntos, listas, multi-conjuntos, vectores e dicionários.	Java possui interfaces para conjuntos, listas, multi-conjuntos e vectores.
Instâncias de Colecções	instâncias de Classes colecção.	Instâncias dessas classes colecção.	Instâncias dessas classes colecção.
Vectores	Vectores C++, ou a classe *d_Varray* para vectores com limite superior variável	instâncias da Classe vector (*Array*).	Suporte sintáctico para sequência indexável de objectos. Classe *Vector* ou a classe ODMG *VArray*
Relacionamentos	Referência para objecto (um-para-um) ou colecção (um-para-muitos) embutida no objecto.	Implementada usando métodos que suportam o protocolo. Referência para objecto (um-para-um) ou colecção (um-para-muitos) embutida no objecto	Relacionamentos como definido em ODMG ainda não suportados.
Extensões	Classe *d_Extent<T>* constitui extensão da classe *T*.	Não suportadas.	Não suportadas.
Chaves	não suportado em C++.	Não suportadas.	Não suportadas.
Nomes	múltiplos nomes por objecto (invocação de *set_object_name()*)	Objectos têm identificador único.	Objectos podem ser nomeados usando métodos da classe *Database*

A tabela seguinte refere-se ao uso das características das linguagens:

	C++	**Smalltalk**	**Java**
Prefixos	nomes globais ODMG têm prefixo *d_* para evitar "colisões" com outros nomes.		
Espaços de Nomes	prevista a utilização de um espaço de nomes *odmg*		A API ODMG para Java será definida num pacote especifico de um produto.
Excepções	Classe *d_Error* derivada da classe *exception* do C++ standard.		Excepções standard lidadas pelas classes ODMGException e ODMGRuntimeException

3.2.6 Modelo OMG - *Object Core Model*

O modelo orientado por objectos OMG/OM da OMG [47],[48] constitui um subconjunto do modelo ODMG/OM da ODMG, ajustando-se este à estrutura definida pela OMG.

O modelo objecto da OMG é baseado num pequeno número de conceitos básicos: objectos, operações, tipos e sub-tipagem. Um objecto pode modelizar qualquer tipo de entidade como por exemplo: uma pessoa, um automóvel, um documento, um departamento, um n-tuplo, um ficheiro ou um gestor de janelas gráficas. Em OMG/OM um objecto possui uma identidade: distinta; imutável ao longo da sua existência; e independente das suas propriedades e comportamento.

O modelo objecto da OMG tem como propósito suportar portabilidade aplicacional, definindo-a em três níveis: projecto; código fonte; e código objecto.

Enquanto o modelo OMG/OM centra-se no primeiro nível, o modelo ODMG/OM agrupa os dois primeiros. O modelo OMG/OM, refere também outras duas dimensões para portabilidade: entre domínios de tecnologia e dentro de um dado domínio, a portabilidade entre os produtos existentes. O standard ODMG/OM enquadra-se no segundo, ou seja, dentro do domínio tecnológico dos SGBDOOs, procurando portabilidade entre os diversos produtos existentes.

O facto de que o modelo ODMG/OM é mais detalhado do que o modelo OMG/OM, fica patente na hierarquia de tipos de ambos. O suporte para propriedades e a apresentação de um desenvolvimento mais detalhado da hierarquia de subtipos debaixo dos tipos *Objecto* e *Literal*, constituem extensões da hierarquia de tipos do modelo OMG/OM em ODMG/OM.

A hierarquia de tipos da ODMG/OM é:

Tipo_Literal
- Literal_Atómico
- Literal_Colecção
- Literal_Estruturado

Tipo_Objecto
- Objecto_Atómico
- Colecção

As diferenças entre ambos residem na nomeação. O tipo que em ODMG/OM é designado por *Literal*, em OMG/OM tem a designação de *Não-Objecto*. Embora OMG/OM não introduza

formalmente um super-tipo dos tipo Objecto e Não-Objecto, em [48] é referido que instâncias destes dois tipos são *Dvals - "denotable values".* Em ODMG/OM, é definido um supertipo para *Objectos* e *Literais*. Instâncias do tipo Objecto são mutáveis. É-lhes atribuído um OID em ODMG/OM, e embora o valor do objecto possa ser alterado, o seu OID é invariante. O OID pode então ser usado para denotar o objecto. Tipos Literais são imutáveis, e instâncias de um tipo literal, distinguem-se entre si pelos seus valores. Esses valores são usados directamente para denotar as instâncias, sem haver necessidade de recorrer a OIDs.

4 Sistema de Descrição Arquivística

No presente capítulo, procede-se à descrição de um protótipo que visa automatizar o tratamento de dados sobre documentos de arquivo histórico. É efectuada uma abordagem ao problema da descrição arquivística, segundo as normas gerais internacionais de descrição em arquivo. Passa-se a seguir à descrição da plataforma adoptada para a implementação da aplicação, procedendo-se à análise de cada um dos seus componentes separadamente. Descreve-se o sistema desenvolvido, nomeadamente a arquitectura usada na sua implementação, e refere-se como se procedeu ao carregamento da base de dados com informação real de um arquivo distrital. Tecem-se no final algumas conclusões.

4.1 Introdução

Tendo como finalidade a automatização de um sistema de descrição arquivística, foi desenvolvida uma aplicação cuja utilização preferencial será em arquivos históricos. O software desenvolvido destina-se a ser executado numa plataforma UNIX, tendo sido utilizadas as linguagens Tcl/Tk [62] para elaboração da interface e C++ [9] para realização da extensão específica criada para Tcl/Tk. A interface com o utilizador apresenta as características comuns de uma aplicação executada num ambiente gráfico, baseando-se em janelas gráficas, menus, botões e caixas de diálogo. A extensão de Tcl/Tk escrita em C++ compreende um conjunto de comandos que interagem com o sistema de gestão de bases de dados. O sistema de gestão de bases de dados seleccionado para ser utilizado na aplicação foi o ObjectStore [46],[45],[44]. Este SGBDOO oferece uma interface de programação em C++ para criar e utilizar bases de dados ObjectStore.

A aplicação desenvolvida inclui a capacidade de introdução de informação na base de dados, permitindo relacioná-la com a informação já existente. Permite também efectuar a consulta de informação quer por navegação, sendo mantido um contexto durante esse processo, quer por interrogações à base de dados. É também suportada a possibilidade de efectuar operações de actualização da informação na base de dados. Por último, é possível a remoção de informação.

4.2 Descrição Arquivística

A prática arquivística envolve várias tarefas específicas relacionadas com o processo arquivístico. Apresentando algumas semelhanças com a prática bibliotecária, denota no entanto especificidades suficientemente importantes para um estudo separado. De entre as diversas tarefas que envolve, uma que se revela de grande importância é o tratamento técnico de fundos documentais das mais variadas instituições e pessoas, através da sua catalogação e inventariação.

Na descrição da informação da qual um arquivo é depositário, interessa o conhecimento de várias vertentes a ela ligadas. Algumas são a seguir referidas:

- Entidades responsáveis pela produção dessa mesma informação;
- Datas de produção, de acumulação e de entrada no Arquivo actual;
- Agrupar em unidades descritivas a informação relacionada, ficando esta com descrições comuns. Esse agrupamento é de natureza hierárquica;
- Relativamente à perspectiva segundo a qual a informação se pode agrupar em unidades, em que cada unidade é constituída por informação relacionada, interessa saber se a totalidade da informação respeitante a uma unidade se encontra na posse do arquivo ou se este só possui parte;
- Ainda no que concerne à unidade, interessa saber se o arquivo possui a informação no formato original, ou se a informação é uma cópia;
- Se a informação estiver na posse de outras entidades, interessa conhecer esses detentores de informação com especial relevo para outros arquivos;
- Que materiais associados à informação existem;
- Qual a localização física de uma unidade de informação no arquivo, e se ela está subdividida em unidades mais pequenas;
- Qual o formato físico da informação.

Como se pode observar existem múltiplos factores relacionados com o processo de descrição arquivística. Com a finalidade de regulamentar e, em especial de normalizar a troca de informação entre arquivos, o Conselho Internacional de Arquivos lançou um conjunto de normas para a descrição arquivística. Estas foram traduzidas para Português pela Associação Portuguesa de Bibliotecários Arquivistas e Documentalistas, conhecida em Portugal por BAD, e criada em 1973, por profissionais portugueses de documentação e informação.

Entre outras finalidades, são objectivos da BAD fomentar a investigação nas áreas relativas aos sectores profissionais por si abrangidos e a promoção do aperfeiçoamento científico, técnico e cultural dos seus associados. A publicação da tradução das normas acima referidas, como documento de referência e não de carácter oficial, foi efectuada nos Cadernos Bad editados pela mesma associação em 1995 [23].

4.2.1 Normas Gerais Internacionais de Descrição em Arquivo

Este conjunto de regras gerais para a descrição em arquivos, publicadas provisoriamente em 1992, revistas em 1993, e cuja versão final foi aprovada em 1994, faz parte de um processo que visa, segundo o Conselho Internacional de Arquivos:

- Assegurar a criação de descrições consistentes, apropriadas e auto-explicativas;
- Facilitar a recuperação e a troca de informação sobre materiais de arquivo;
- Possibilitar a partilha de dados de autoridade;
- Tornar possível a integração de descrições de diferentes arquivos num sistema unificado de informação.

A divisão estrutural adoptada para uma determinada descrição de cada entidade arquivística, agrupa elementos de descrição dessa entidade em seis zonas de informação:

- Identificação, constituindo a informação essencial para proceder à identificação da unidade de descrição;
- Contexto, que congrega a origem e custódia da unidade de descrição;
- Conteúdo e estrutura, contempla informação sobre o assunto e organização da unidade de descrição;
- Condições de acesso e de utilização, informação sobre a disponibilidade da unidade de descrição;
- Materiais associados, agrega informação sobre materiais com uma relação importante com a unidade de descrição;
- Notas, informação especializada ou qualquer outra informação que não possa ser incluída em nenhuma das outras zonas.

O conceito central, tanto das normas como da aplicação, é o de unidade de descrição, já descrito no segundo capítulo deste texto. As unidades de descrição podem estar relacionadas através de uma hierarquia de níveis. Nesta, cada nível tem associadas determinadas unidades de descrição. Assim, por exemplo, um fundo poderá estar descrito como um todo devendo ser representado por apenas uma unidade de descrição de fundo. No caso de a descrição das suas partes se justificar, estas poderão ser descritas em separado, utilizando-se então unidades de descrição de níveis mais baixos. Este processo poderá repetir-se até se atingir o nível de detalhe na descrição desejado. A soma de todas as unidades de descrição obtidas representa o fundo e as respectivas partes, para as quais foram elaboradas descrições.

Situações típicas de combinações de níveis de descrição podem ser visualizadas na figura seguinte:

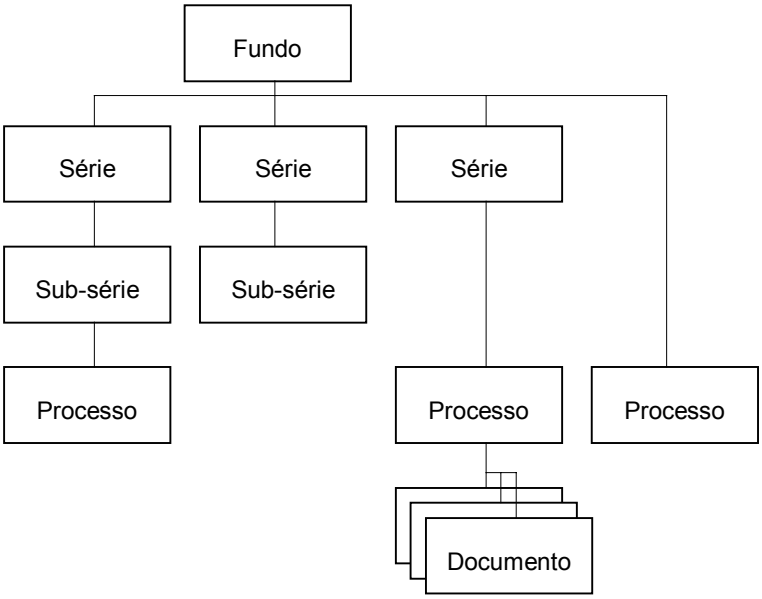


Figura 4.1: Exemplos de esquemas de níveis possíveis

As normas referem também que, quando é adoptada a descrição em vários níveis se deve ter em atenção:

- No nível de fundo, dar informação dele como um todo, nos níveis seguintes, dar informação para as partes descritas;
- Deve-se fornecer informação apropriada para o nível em causa;
- Deve haver relacionamentos que possibilitem a identificação da unidade de descrição imediatamente superior e de qual o nível da descrição;
- Não deve haver redundância de informação, e a informação comum deve ficar sempre no nível mais alto.

4.2.2 Análise Conceptual

O essencial deste modelo foi elaborado pela equipa do projecto JNICT - Archivum em curso no INESC-Porto. Na análise conceptual da aplicação desenvolvida, foram contemplados todos os elementos de descrição das normas ISAD. Estes surgem como atributos de objectos no modelo obtido.

Pode-se efectuar a divisão do esquema conceptual em partes correspondentes a subconjuntos de entidades fortemente ligadas. Como referido no ponto anterior, a entidade central da aplicação é a unidade de descrição. Cada zona de informação das normas ISAD, com os atributos correspondentes, reflecte-se numa ou mais partes do esquema conceptual. As partes que constituem o esquema conceptual podem ser visualizadas na figura abaixo.

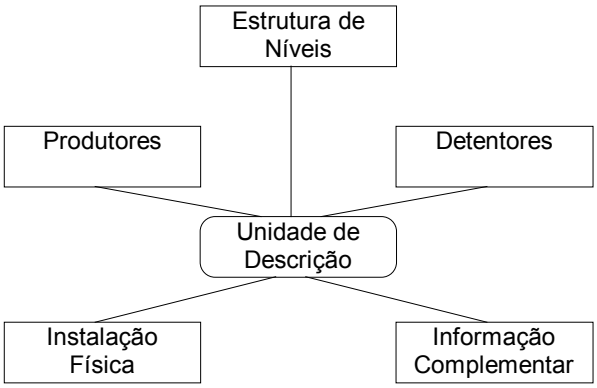


Figura 4.2: Divisão esquemática do modelo da aplicação

O coração do esquema da figura acima, contempla a entidade Unidade de Descrição, que pode corresponder à descrição da informação respeitante a qualquer um dos níveis definidos no esquema de níveis para o fundo em causa. Cada entidade Unidade de Descrição está associada obrigatoriamente com um dado nível, e é considerada parte integrante de uma Unidade de Descrição de um nível acima, excepto no caso dos fundos, que constituem o topo da hierarquia.

Existe um conceito de "esquema de níveis", a definir pelo arquivista para cada fundo, cujo objectivo é disciplinar as ligações possíveis entre unidades de descrição a níveis hierárquicos diferentes.

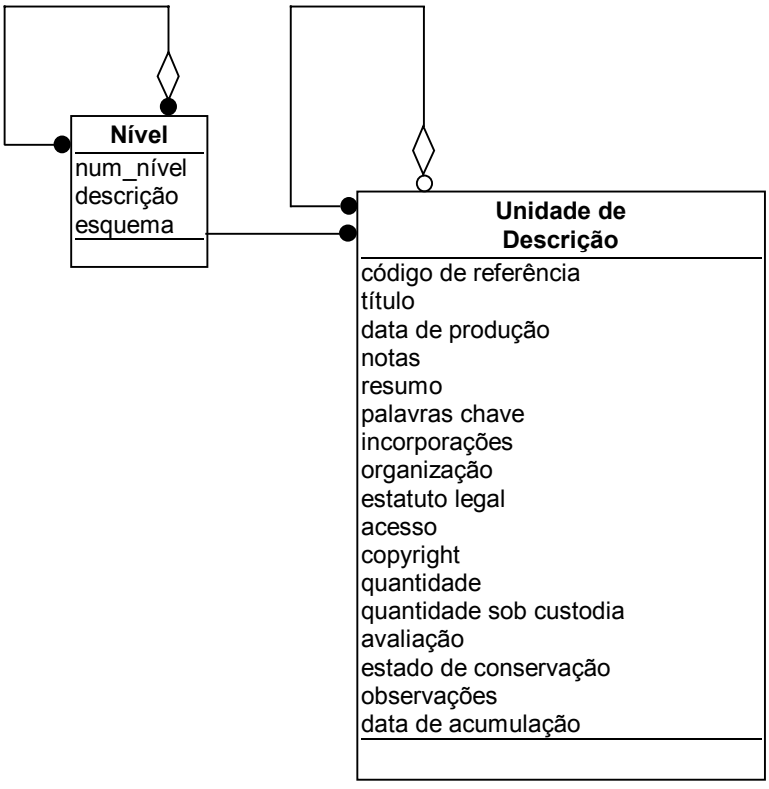


Figura 4.3: Subesquema que modeliza a estrutura de níveis

Os esquemas de níveis são guardados através de associações entre entidades do tipo Nível que estão associadas entre si e com entidades Unidade de Descrição. Um nível do esquema é representado por uma entidade Nível. Um esquema de níveis é um conjunto de entidades Nível. A parte do modelo Estrutura de Níveis refere-se a essa informação. A modelização dessa parte está ilustrada na figura anterior.

A classe Nível contém meta-informação, armazenando informação sobre as estruturas de nível consideradas legais. Cada esquema de nível tem uma denominação - atributo Nível.esquema, e é composto por um conjunto de entidades Nível que estão numeradas - atributo Nível.num_nível e têm uma descrição - atributo Nível.descrição como por exemplo Fundo, Série, etc.

Deste modo consegue-se flexibilidade para representar qualquer esquema de níveis possível, de acordo com a natureza dos documentos a arquivar, e ao mesmo tempo garante-se que nas hierarquias de unidades de descrição existentes seja cumprido esse esquema.

Os restantes subesquemas com indicação da sua função no modelo são enumerados a seguir. O modelo OMT completo pode ser consultado no anexo A.

Produtores

Modeliza a informação respeitante ao produtor da Unidade de Descrição. Na classe *Produtor* são identificadas individualmente designações juntamente com a época para a qual a designação terá sido válida. Existem duas associações entre Produtor e Unidade de Descrição. Uma corresponde ao nome de produtor das normas ISAD, outra ao conceito de produtor original, identificando, caso se justifique, um produtor original da informação quando o produtor registado tenha tido uma função de acumulação.

A classe *Descrição do Produtor* tem elementos da história administrativa ou biográfica de um produtor, independentemente da sua eventual mudança de designação ao longo do tempo. Esta classe está associada com a classe Produtor para dar conta das mudanças de designação. Visto poder haver uma designação principal de um produtor, este facto é consagrado no modelo através de uma associação paralela à última descrita para o produtor em questão. Instâncias de Descrição de Produtor podem estar ligadas via uma relação de agregação com outra Descrição de Produtor. Serve esta estrutura para captar a organização hierárquica de produtores caso esta exista como, por exemplo, o caso de um ministério que produziu um fundo, no qual as séries e subséries são produzidas pelos departamentos e secções respectivos.

Instalação Física

Suporta a localização e arrumação física da informação. Instâncias da classe *Unidade de Instalação* podem representar diversos tipos de unidades físicas (caixa, livro, maço, rolo, etc.) sendo esta numerada com uma Referência e colocada em prateleiras.

As classes *Prateleira*, *Estante*, *Corpo* e *Depósito* têm entre si relações de agregação. A ordem de uma unidade física na prateleira é armazenada em Ordem da classe *Localização* que associa Unidades de Instalação a Prateleiras.

Detentores

É neste subesquema que são consideradas entidades que podem estar na posse de informação. A classe *Detentor* denota um detentor de informação. Para abarcar os casos em que o detentor é um arquivo, foi considerada a classe Arquivo que herda da classe Detentor.

Um detentor pode possuir exemplares da unidade descrição que podem ser cópias ou ser o original, sendo utilizadas as classes *Cópia* e *Original* para esse efeito. A classe *Custódia* identifica um período no qual o detentor teve a custódia da unidade de descrição. A informação sobre a documentação relacionada com uma unidade de descrição e que está na posse de um dado detentor é armazenada na classe *Materiais Associados*. As últimas quatro classes reflectem relacionamentos entre instâncias de Unidades de Descrição e Detentores.

Informação Complementar

Refere-se a informação acerca das unidades de descrição que não se enquadra em nenhum dos subesquemas anteriores.

As classes *Publicação* e *Notas de Publicação* identificam publicações que tenham sido baseadas na utilização, estudo ou análise de uma ou mais unidade de descrição.

A classe *Idioma* tem o objectivo de identificar o(s) idioma(s), tipos de letras e sistemas de símbolos utilizados numa unidade de descrição.

A classe *Matéria Scriptoria* juntamente com o atributo *estado de conservação* da classe Unidade de Descrição, fornecem informação sobre quaisquer características físicas importantes na utilização da unidade de descrição.

Como auxiliares de pesquisa de unidades de descrição são outras unidades de descrição, a classe *Auxiliares de Pesquisa* associa instâncias de Unidade de Descrição.

4.3 Requisitos e Informação da Base de Dados

O estudo de requisitos de uma aplicação de base de dados constitui um processo fundamental para a decisão de qual a melhor solução a ser utilizada na aplicação. A enumeração de requisitos efectuada nesta secção direcciona-se mais para o estabelecimento de critérios de selecção do modelo de dados a considerar.

O modelo da aplicação de descrição arquivística apresenta uma grande riqueza em termos de relacionamentos entre as entidades que o compõem. Dada a sua importância, esse factor deve ser tido em conta na análise efectuada.

Procura-se a seguir detalhar alguns dos requisitos da aplicação em termos de base de dados:

- Elevado número de relacionamentos entre entidades;
- Hierarquia de níveis do esquema necessita de mecanismo de definição poderoso;
- Evolução do esquema de base de dados através da inclusão de novas entidades e modificação da constituição de entidades existentes. Embora este requisito seja relaxado no tempo, pode haver a necessidade de adequação do esquema da base de dados a novas especificações ditadas pela ISAD ou de incorporação de informação noutros formatos (ver ponto seguinte);

- Suporte para informação em múltiplos formatos como, por exemplo, imagem ou audio;
- Suporte para ambiente de multi-utilização de aplicações clientes com diversas permissões de acesso à base de dados. Com a concretização da disponibilização de acesso para alguns tipos de consultas através da Internet é alargado massivamente o universo de utilizadores, devendo por isso a base de dados suportar uma grande quantidade de utilizadores a efectuarem consultas. A arquitectura do SGBD deve poder adaptar-se a um ambiente distribuido como a Internet;
- Suporte para diversas versões dos objectos correspondentes às várias arrumações físicas que as Unidades de Instalação poderão ter ao longo do tempo;
- O SGBD deve incluir mecanismos de segurança como definição de perfis de utilização com permissões de acesso configuráveis;
- Volume de dados elevado com crescimento contínuo. O SGBD deve apresentar uma boa escalabilidade dadas as perspectivas de crescimento traçadas.

A lista de requisitos da aplicação, traduz-se numa mescla de requisitos facilmente suportados pelo modelo relacional e outros mais adaptados para a utilização de um modelo OO.

4.4 Sistema Desenvolvido

O sistema de descrição arquivística foi desenvolvido sobre o sistema operativo UNIX da Digital versão 3.2. A aplicação aproveita as potencialidades gráficas do sistema gráfico de gestão de janelas instalado nesta máquina. Assim, a aplicação está concebida para ser executada num terminal gráfico, ou num computador que suporte o sistema de gestão de janelas por emulação. Outros factores relevantes para a escolha desta plataforma de desenvolvimento foram:

- Disponibilidade do software do sistema de gestão de base de dados para essa plataforma, encontrando-se instalado numa estação de trabalho Alpha Station da Digital provida do sistema operativo referido;
- Disponibilidade das linguagens de programação requeridas para o desenvolvimento da aplicação;
- Possibilidade de facilmente converter a aplicação para ser executada noutras plataformas UNIX, tornando a aplicação facilmente transportável para outros sistemas a nível de código fonte.

O facto de constituir um sistema operativo multi-tarefa que facilmente suporta vários utilizadores em simultâneo, oferece a possibilidade de várias aplicações estarem em simultâneo a aceder à base de dados, constituindo este um requisito importante.

Linguagem C++

A linguagem no qual foram implementadas as rotinas de acesso e manipulação da base de dados foi C++. Foi utilizado o compilador de C++ da Digital - *cxx* versão 5.5 para o sistema operativo Digital Unix. Esta ferramenta de desenvolvimento teve um papel chave em todo o processo de implementação. A ligação à base de dados foi efectuada recorrendo a C++, tendo sido *cxx* o

compilador utilizado na geração do ficheiro objecto do esquema da base de dados. A codificação da extensão de comandos Tcl/Tk foi também efectuada em C++.

Esta linguagem constitui uma versão orientada por objectos da popular linguagem C [38]. C++ estende a linguagem C com o conceito de classes, objectos, mecanismos de herança simples e múltipla, polimorfismo, referências e outros. Algumas das vantagens apontadas a esta linguagem são o modo natural como se projectam e desenvolvem as aplicações, possibilitando também mais valias em termos de reutilização, manutenção e eficiência do software desenvolvido.

Linguagem Tcl/Tk

Tcl é uma linguagem interpretada, de domínio público e de utilização relativamente simples, permitindo a utilização de variáveis, estruturas de controlo de fluxo e procedimentos. Tcl e a sua extensão para suporte de sistemas de gestão de janelas gráficas - Tk, foram desenvolvidas em ambiente universitário, encontrando-se disponíveis para plataformas como UNIX, Windows e Macintosh.

A possibilidade de adicionar facilmente um interpretador Tcl a uma aplicação, constitui um dos grandes atractivos desta linguagem. Substituindo o papel de uma linguagem de comandos específica para a aplicação, Tcl permite que um conjunto de operações primitivas da aplicação seja associado a novos comandos que estendem desse modo a linguagem, podendo estes ser usados num programa Tcl.

Os comandos mencionados podem ser escritos usando a linguagem C, dado Tcl oferecer uma interface de utilização simples para esta linguagem. Para tal é necessária a ligação da biblioteca de comandos Tcl com os novos comandos criados pela aplicação. Deste modo o ficheiro executável obtido será um interpretador que entenderá não só os comandos Tcl como também o conjunto de novos comandos criados quer em C quer em C++, em que cada comando está associado a um procedimento escrito numa dessas linguagens. O programa Tcl obtido, como ilustrado na figura abaixo, pode utilizar comandos da biblioteca ou novos comandos definidos pela aplicação.

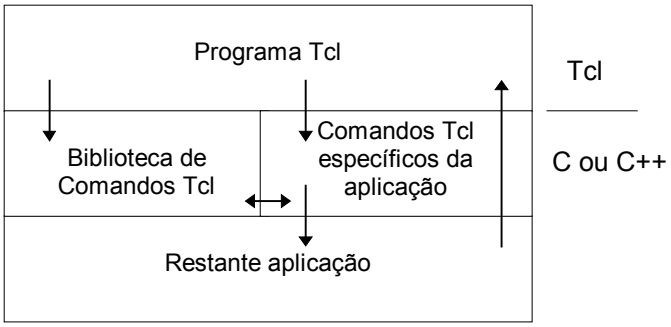


Figura 4.4: Estrutura de uma aplicação que usa a biblioteca Tcl

Esse carácter extensível permite a ligação num só executável de várias bibliotecas de comandos. Uma das mais atraentes é Tk. Esta extensão de Tcl possibilita a utilização do potencial oferecido pela interface de programação para os sistemas de janelas gráficas mais conhecidos, apoiando a elaboração de aplicações com evoluídas interfaces gráficas para o utilizador.

As versões usadas no desenvolvimento foram: 7.6 para Tcl e 4.2 para Tk.

Base de Dados ObjectStore

O SGBDOO ObjectStore, utilizado na construção do sistema de descrição arquivística, permite o desenvolvimento de aplicações em C e C++ que requeiram serviços de bases de dados. Suporta o uso de vários compiladores C++ disponibilizando um grupo de bibliotecas ObjectStore para C++ que serão ligadas com o código objecto da aplicação. É através dessas bibliotecas de classes que o ObjectStore permite o acesso da aplicação às funcionalidades de bases de dados. Essas bibliotecas incluem também funções globais como a sobrecarga de operador *new()* que permite a alocação dinâmica de memória persistente para qualquer tipo de objectos.

A informação persistente em ObjectStore é armazenada numa *base de dados* que pode tomar a forma de um ficheiro do sistema operativo. Uma base de dados é constituída por *segmentos*, que por sua vez são compostos por *páginas*.

O ObjectStore necessita de armazenar em cada base de dados informação acerca das classes de objectos armazenados. Essa informação sobre o esquema é armazenada sob a forma de objectos C++. A informação acerca do esquema de uma aplicação, criada pelo *gerador de esquemas* do ObjectStore, é armazenada em dois locais: um *ficheiro fonte do esquema da aplicação* e uma *base de dados do esquema da aplicação*. O gerador de esquemas, aceita como ficheiros de entrada o conjunto de *ficheiros fonte de esquemas* e possivelmente *esquemas de bibliotecas*. Os primeiros são construídos pelo utilizador, e contêm uma lista das classes cujas instâncias poderão tornar-se persistentes ou cujas instâncias possam funcionar como *pontos de entrada* na base de dados. Os esquemas de bibliotecas são bases de dados que contêm informação de esquema para bibliotecas que armazenam ou acedem informação persistente. As bibliotecas podem ou não ter *esquemas de bibliotecas* associados. O ficheiro fonte do esquema da aplicação tem de ser compilado e ligado com a aplicação que vai utilizar os serviços persistentes do ObjectStore (ver figura 4.6).

As aplicações ObjectStore necessitam de dois processos auxiliares para poderem ser executadas: um *servidor ObjectStore* e um *gestor de memória cache*. O primeiro gere o acesso às bases de dados ObjectStore. Cada máquina onde está localizada uma base de dados ObjecStore tem obrigatoriamente um servidor ObjectStore, que tem de estar activo para ser possível aceder a essa base de dados. Uma aplicação pode usar várias bases de dados, podendo essas bases de dados estar localizadas em diferentes sistemas de ficheiros e ser geridas por diferentes servidores ObjecStore. Um gestor de cache é iniciado automaticamente sempre que uma aplicação ObjectStore é executada. O gestor de cache corre na mesma máquina que a aplicação e participa na gestão da cache de cliente da aplicação. O mesmo gestor de cache executa essa tarefa para todas as aplicações clientes de uma máquina, havendo no máximo apenas um gestor de cache por cada máquina com aplicações clientes. Esta arquitectura distribuída de cache de cliente também permite acesso transparente a objectos através da rede. A figura de abaixo ilustra esta arquitectura.

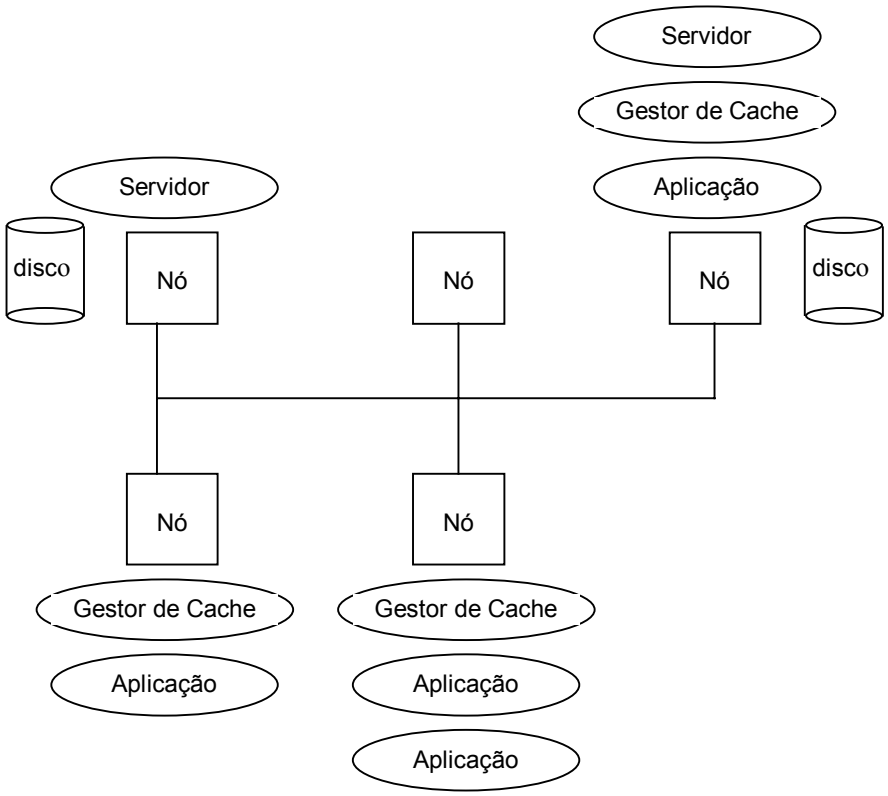


Figura 4.5: Arquitectura de processos do ObjectStore

ObjectStore oferece ainda outras funcionalidades de base de dados como:

- Transacções dinâmicas e lexicais;
- Recuperação automática de falhas, cópias de segurança incrementais, e ficheiro de registo de operações;
- Detecção automática de *deadlock*, controlo de concorrência multi-versão permitindo que uma operação de escrita não seja bloqueada quando outras operações de leitura estão a decorrer;
- Interrogações, integridade referencial através de relacionamentos;
- Biblioteca de colecções;
- Indexação automática;
- Autenticação e privilégios de acesso à base de dados e segmento;
- Evolução de objectos e configurações;
- Evolução de esquemas.

4.4.1 Processo de Desenvolvimento

O processo de desenvolvimento da aplicação para descrição arquivística pode dividir-se em duas partes fundamentais.

A primeira refere-se à implementação em ObjectStore/C++ do modelo obtido pela análise conceptual. Aqui procedeu-se à identificação das classes a implementar, dos relacionamentos e hierarquias existentes entre elas e da codificação das funções membro que determinam o seu comportamento. O ObjectStore obriga também à identificação das classes que podem ficar persistentes.

Também se pode englobar nesta parte a elaboração de um programa que vai criar a base de dados propriamente dita. Nele procede-se à criação da base de dados, definição de índices e de pontos de entrada na base de dados, assim como inserção de informação para efeitos de teste.

A segunda parte do desenvolvimento centrou-se na aplicação que utiliza a base de dados. A aplicação foi codificada em C++, implementando um conjunto de comandos Tcl para interacção com a base de dados. Nesta parte do desenvolvimento foram considerados vários grupos de operações, essencialmente sobre a base de dados, para serem codificados como comandos a serem executados num programa Tcl/tk. Deste modo criou-se uma aplicação que responde a eventos gerados pelo utilizador, através da interface gráfica criada pelo programa Tcl/tk, e consegue-se efectuar operações sobre a base de dados. A aplicação em C++ funciona como a "cola" entre a base de dados e a interface gráfica.

Apesar de naturalmente a segunda parte do desenvolvimento se seguir à primeira parte em termos temporais, acontece que o processo de desenvolvimento é intrinsecamente iterativo. Por este facto, e pela forte ligação existente entre as duas, a parte final do ciclo de desenvolvimento envolveu ambas as partes embora com predominância da segunda parte. A seguir, são abordados aspectos relevantes do processo de desenvolvimento.

Base de Dados de Descrição Arquivística

O desenvolvimento da base de dados de descrição arquivística implicou duas fases:

1.Codificação do esquema da base de dados;

2.Criação da base de dados.

Na primeira fase citada, procedeu-se ao mapeamento do modelo OMT para o esquema de classes da base de dados. Além das classes que fazem parte deste esquema foram acrescentadas as seguintes três classes:

- Data: esta classe permite o armazenamento de uma data sob a forma de três números inteiros para o dia, mês e ano. Além da interface funcional para ler e escrever cada um dos seus atributos, foi também definida uma função membro que retorna a data em formato cadeia de caracteres. Esta funcionalidade revelou-se útil para a utilização da classe na aplicação Tcl/tk desenvolvida;
- Quantidade: visto o tipo de unidades de descrição ser muito variado (microfilmes, folhas de papel, etc.), resolveu-se acrescentar a possibilidade de definir na quantidade, qual o tipo de elementos quantificado. Assim surge a classe quantidade que além de possuir um atributo *numero* do tipo inteiro para armazenar a quantidade, possui também um atributo

tipo definido como cadeia de caracteres para armazenamento do tipo de informação quantificada;

- Horário: a definição da classe horário permite o armazenamento de um horário de funcionamento. Esta classe contempla a possibilidade de definir uma hora de entrada e uma hora de saída. É usada na classe Arquivo.

Na declaração das classes criou-se um ficheiro para cada classe. Estes podiam conter: nome da classe, atributos da classe, funções membro, operadores e declaração de membros inversos para definição dos relacionamentos entre classes.

A implementação das funções membro das classes, do corpo das declarações de membros inversos e das funções membro que podem ser utilizadas em expressões de interrogação foi efectuada no ficheiro *arch.cc*.

No ficheiro fonte de esquema - *schema.cc* colocaram-se:

- Todas as classes criadas que poderiam tornar-se persistentes;
- Colecções de classes criadas que poderiam tornar-se persistentes;
- Funções membro que poderiam ser usadas em expressões de interrogação à base de dados.

Este ficheiro serviu de base ao gerador de esquemas do ObjectStore para a criação ficheiro fonte do esquema da aplicação - *osschema.cc* e da base de dados do esquema da aplicação - *arch.adb* (ver figura 4.6).

A segunda fase do desenvolvimento da base de dados consistiu na elaboração do programa C++ *arch_ini.cc* que deu origem ao programa *arch_ini* usado para a criação da base de dados *arch.db*.

A aplicação *arch_ini* tem as seguintes funções:

- criação da base de dados *arch.db*;
- criação dos pontos de entrada na base de dados. Estes pontos de entrada na base de dados permitem o acesso das aplicações à informação da base de dados. Foram criados pontos de entrada para colecções que funcionam como extensões das classes. As classes com esse tipo de extensões estão listadas na tabelas de baixo:

Nome da classe	Nome do ponto de entrada	Extensão da classe (conjunto)
Produtor	rootProds	os_Set<Produtor*>
Udescrição	rootUdescs	os_Set<Udescrição*>
Nível	rootNiveis	os_Set<Nivel*>
Uinstalação	rootUinsts	os_Set<Uinstalação*>
Detentor	rootDetentores	os_Set<Detentor*>
Arquivo	rootArquivos	os_Set<Arquivo*>
Local	rootLocais	os_Set<Local*>
Publicação	rootPubls	os_Set<Publicação*>

- definição de índices. Foram criados índices para os seguintes atributos: *Produtor.designação*, *Udescrição.titulo*, *Udescrição.codrefer*, *Uinstalação.referencia*, *Detentor.codigo*, *Detentor.nome*, *Nivel.descrição*. Para a criação de classes cujos atributos possam ser indexáveis com manutenção automática pelo ObjectStore é necessária a definição nas respectivas classes de um atributo do tipo *os_backptr*. A utilização de objectos deste tipo é necessária também para a definição de funções que possam ser usadas em expressões de interrogação;
- criação de informação para testes da base de dados. A informação de teste criada consistia em três esquemas de níveis, com respectivas unidades de descrição, instâncias de produtores e de detentores.

O ObjectStore permite a definição da opção de propagação da remoção na declaração de membros inverso. Quando esta opção é utilizada na declaração da classe1 de um membro inverso do tipo classe2, sempre que se remove um objecto da classe1 o objecto associado da classe2 também é removido.

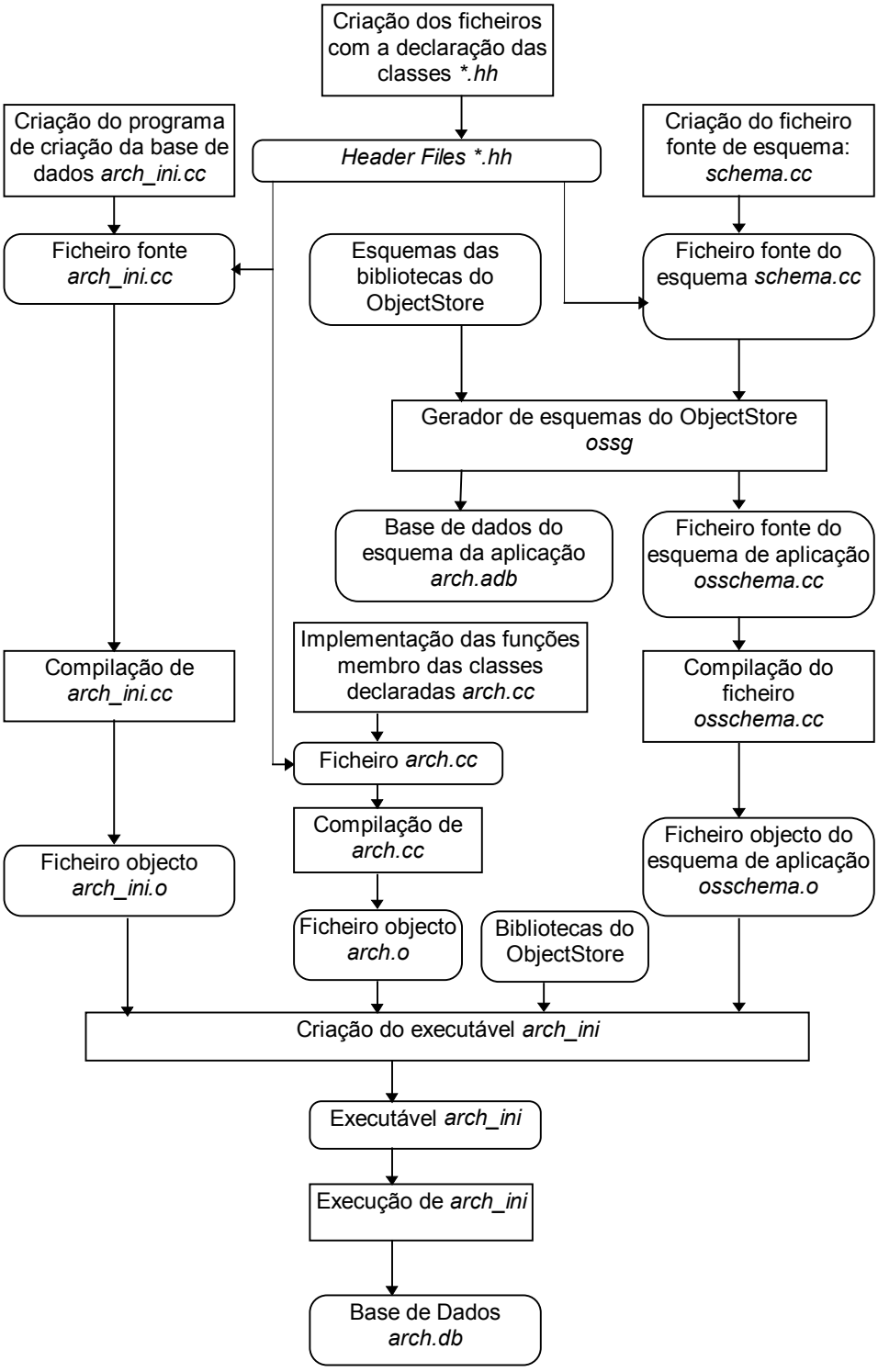


Figura 4.6: Processo de desenvolvimento da base de dados

Aplicação para utilização da Base de Dados de Descrição Arquivística

Tal como o desenvolvimento da base de dados, também o desenvolvimento da aplicação se pode dividir em duas partes:

1.Codificação da aplicação em C++;

2.Escrita do programa em Tcl/Tk

O executável da aplicação - *tkarch*, vai ser um novo interpretador Tcl/tk com extensão de comandos. Significa que qualquer programa escrito na linguagem Tcl/Tk pode ser executado pelo interpretador tkarch. Além de todas as instruções da linguagem Tcl e das instruções da popular extensão para criação e gestão da interface gráfica Tk, o executável tkarch reconhece também os comandos implementados para interacção com a base de dados de descrição arquivística.

Os comandos criados na aplicação C++ representam funções escritas em C++. Estas funções podem receber argumentos do programa Tcl/Tk. Alguns dos comandos implementados na aplicação C++ são descritos na tabela seguinte. A coluna da esquerda tem o nome do comando em Tcl/Tk, a do centro o nome da função da aplicação C++ que é chamada e a da direita tem uma breve descrição do comando.

Nome do comando Tcl/Tk	Nome da função da aplicação	Descrição do comando
initbd	InitbdCmd()	abre a base de dados.
closebd	ClosebdCmd()	fecha a base de dados.
getnvlst	GetnvlstCmd()	retorna toda a informação sobre um esquema de nível.
getnvsup	GetnvsupCmd()	retorna para o programa Tcl uma lista de todos os esquemas de níveis na base de dados.
getud	GetudCmd()	lê uma unidade de descrição de dados a partir do seu código de referência.
insud	InsudCmd	insere uma nova unidade de descrição na base de dados.
setud	SetudCmd()	altera uma unidade de descrição da base de dados.
getudsup	GetudsupCmd()	retorna todas as unidades de descrição que estão no topo da hierarquia do seu esquema de níveis.

Os comandos que interagem com a base de dados contêm transacções lexicais. Deste modo, as transacções com a base de dados têm a duração da execução de um comando, sendo principiadas no início da função associada ao comando e terminadas no fim do comando. A vantagem de uma abordagem deste género é o facto de evitar o uso de transacções de duração indefinida e dependentes do utilizador.

A definição de novos comandos específicos como extensão de Tcl é efectuada usando a função *Tcl_CreateCommand()* e indicando o nome do novo comando, assim como o nome da função C++ executada quando o comando é chamado. As chamadas à função *Tcl_CreateCommand()* são efectuadas dentro da função *Tcl_AppInit()* localizada em *tkMain.cc*.

É neste ficheiro que está localizada a função *main()*. Quando o interpretador tkarch é executado, é chamada a função - *Tk_Main()* dentro de *main()*. A função *Tk_Main()* vai criar o interpretador Tcl, criar a janela principal do programa e chamar *Tcl_AppInit()* onde são efectuadas

as inicializações como, por exemplo, o registo de novos comandos específicos. Após *Tcl_AppInit()* ser executado o interpretador vai começar a correr um programa Tcl ou fica em ciclo esperando que o utilizador digite um comando na linha de comandos.

O programa Tcl/Tk criado - *archivum.tcl*, é responsável por gerar a interface gráfica do utilizador e definir os mecanismos de resposta aos eventos provocados pelo utilizador. Por exemplo, quando o utilizador pressiona um botão da aplicação, vai ser executado um procedimento Tcl/Tk constituído por um grupo de comandos Tcl/Tk. Nestes comandos podem estar incluídos comandos específicos da aplicação que apenas são entendidos pelo interpretador tkarch.

A arquitectura da aplicação pode ser visualizada na figura seguinte.

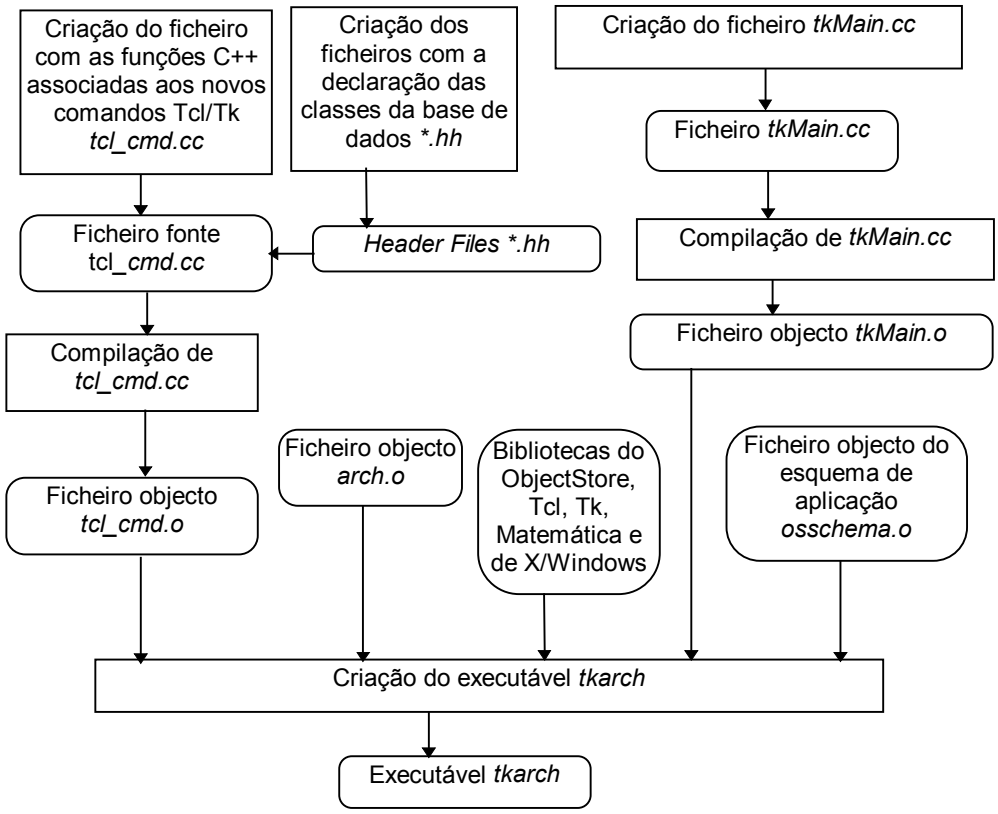


Figura 4.7: Processo de desenvolvimento da aplicação e base de dados

A aplicação C++ altera o valor de variáveis Tcl através das funções *Tcl_SetVar()* e *Tcl_SetVar2()*. Pode também ler o valor de variáveis Tcl através de *Tcl_GetVar()* e *Tcl_GetVar2()*. É deste modo que a aplicação C++ recebe a informação passada pelo utilizador através das variáveis Tcl/Tk, e que a informação lida da base de dados é enviada para a aplicação Tcl/Tk.

O vector global Tcl - *ud* armazena informação acerca de uma unidade de descrição. É através desta variável, que está ligada a caixas de texto que podem ou não ser alteradas pelo utilizador, que o programa Tcl comunica com o utilizador. Também é com recurso a esta variável, que é manipulada pela aplicação C++ através de *Tcl_GetVar2()* e *Tcl_SetVar2()*, que a aplicação

consegue enviar e receber informação sobre uma unidade de descrição. Para outras classes o mecanismo de comunicação e transmissão de informação usado é semelhante.

Como o conteúdo das variáveis Tcl é sempre do tipo cadeia de caracteres, quando se envia um número inteiro ou real de Tcl para a aplicação em C++, nesta última é necessário converter a cadeia de caracteres para numeral para ficar no formato desejado.

Uma das características da aplicação de descrição arquivística reside no facto do utilizador poder "navegar" na base de dados. Significa isto que o utilizador pode por exemplo começar a pesquisar um dado fundo, nesse fundo seleccionar uma série, depois nessa série uma sub-série e possivelmente continuar até ao nível mais baixo. Todas estas unidades de descrição representam um contexto de pesquisa em que o utilizador se encontra. Esse contexto de pesquisa pode aplicar-se a outras situações: um utilizador pode seleccionar uma unidade de descrição arbitrária e depois começar a investigar informação com ela relacionada como produtores, detentores ou localizações físicas.

O problema de armazenamento do contexto quando um utilizador está a "navegar" na base de dados foi resolvido na primeira situação descrita com recurso a dois vectores persistentes cujos elementos apontam para as unidades de descrição percorridas - um dos vectores, e para os níveis percorridos - o outro vector. Deste modo para voltar para a unidade de descrição anterior, basta ir buscar o seu endereço ao vector.

Para a segunda situação, recorre-se a um objecto da classe *Contexto*, cujos atributos são apontadores para as classes que circundam a de unidades de descrição.

Exemplo de um comando para Tcl/Tk implementado - *getudsup*

Apresenta-se a seguir um dos comandos implementados na aplicação. Embora a listagem esteja comentada, chama-se a atenção para a expressão de interrogação da base de dados. Esta interrogação devolve todas as unidades de descrição que não têm nenhuma outra u.d. acima na hierarquia de níveis. Cada objecto u.d. tem um atributo *col* que representa a colecção de u.d.s a quem ela está subordinada. Na expressão de interrogação procuram-se todas as u.d.s cuja colecção *col* seja vazia:

```
!col[:1:]
```

a expressão `col[:1:]` é avaliada para o valor lógico verdadeiro desde que a colecção *col* tenha pelo menos um elemento. A negação desta expressão é verdadeira sempre que *col* não possua nenhum elemento, o que significa que essa u.d. não tem nenhuma u.d. superior, portanto é uma u.d. do topo da hierarquia das u.d.s. A expressão de interrogação é aplicada ao conjunto *udescs*, que contém todas as u.d.s da base de dados. A expressão acima é aplicada a cada u.d. de *udescs*.O resultado da interrogação é atribuido à variável colecção *q_ud*.

A listagem completa do comando vem:

```
// comando "getudsup": retorna lista de todas as uds
// do topo da hierarquia de níveis
int GetudsupCmd(ClientData clientData,
    Tcl_Interp *interp, int argc, char *argv[])
{
char *indice;
```

```
    if (argc != 1) {
       interp->result = "Comando sem argumentos: getudsup";
       return TCL_ERROR; }

    // início de uma transacção tx_1 só de leitura
    OS_BEGIN_TXN(tx_1,0,os_transaction::read_only)
      os_typespec* typeUD = UDescrição::get_os_typespec();
      os_typespec*                    typeUDs                    =
os_Set<UDescrição*>::get_os_typespec();

    // devolve apontador para uma colecção de uds através
    // do ponto de entrada "rootUdescs"
      udescs = (os_Set<UDescrição*>*)
      (db->find_root("rootUdescs")->get_value(typeUDs));

    // construcao da interrogação
    ostrstream interrogaStream;
      interrogaStream << "!col[:1:]" << ends;

    // execucao da interrogação. Resultado devolvido para q_ud
    os_Set<UDescrição*>& q_ud =
      udescs->query("UDescrição*", interrogaStream.str(), db);

    interrogaStream.rdbuf()->freeze(0);
    indice = (char *) malloc(64);
    int i=1;

    // Criação de um cursor para percorrer a coleccao resultado
    // da interrogação
    os_Cursor<UDescrição*> curs_ud(q_ud);

    // itera o resultado e passa informação para
    // variáveis Tcl/Tk
    for (UDescrição* u = curs_ud.first(); curs_ud.more();
         u = curs_ud.next())
    {
    Tcl_PrintDouble(interp,(double) i, indice);
    Tcl_SetVar2(interp,"codref_ud_sup",indice,u->lerCodrefer(),
    TCL_LEAVE_ERR_MSG );
    i++;
    }

    // passa para Tcl/TK o numero de uds de topo existentes
    Tcl_SetVar2(interp,"codref_ud_sup","n",indice,TCL_LEAVE_ERR_MS
G);
```

```
free (indice);

// fim da transacção tx_1
OS_END_TXN(tx_1)
return TCL_OK;
}
```

4.4.2 Utilização da Aplicação

O facto de ter sido adoptada uma interface gráfica para a aplicação, torna a sua utilização muito mais agradável e intuitiva. O utilizador interage com a aplicação usando o teclado e o rato, e através de botões, caixas de texto, caixas de diálogo, menus, barras de rolamento horizontais e verticais e caixas de listas. Todos estes mecanismos gráficos de interacção com o utilizador preenchem o interior das janelas gráficas da aplicação.

Quando se corre a aplicação, digitando *archivum.tcl*, são executadas algumas tarefas de inicialização, como por exemplo a abertura da base de dados. Após esta fase o utilizador depara-se com o ecrã de entrada (figura seguinte).

A janela principal, está dividida em quatro partes principais. Em cima tem-se um menu de acesso à informação respeitante aos subesquemas do modelo conceptual, em baixo do lado esquerdo aparece recriado o esquema de níveis seleccionado e do lado direito o conteúdo da unidade de descrição seleccionada. Entre o menu e as duas partes inferiores, tem-se uma linha de botões.

No canto superior direito é possível seleccionar um dos esquemas existentes na base de dados (*Esquemas de Nível*) ou uma das unidades de descrição de topo (*U.D.s existentes*), como por exemplo um fundo, existentes na base de dados.

Quando a aplicação é iniciada, é seleccionado arbitrariamente um dos esquemas e uma unidade de descrição dentro deste.

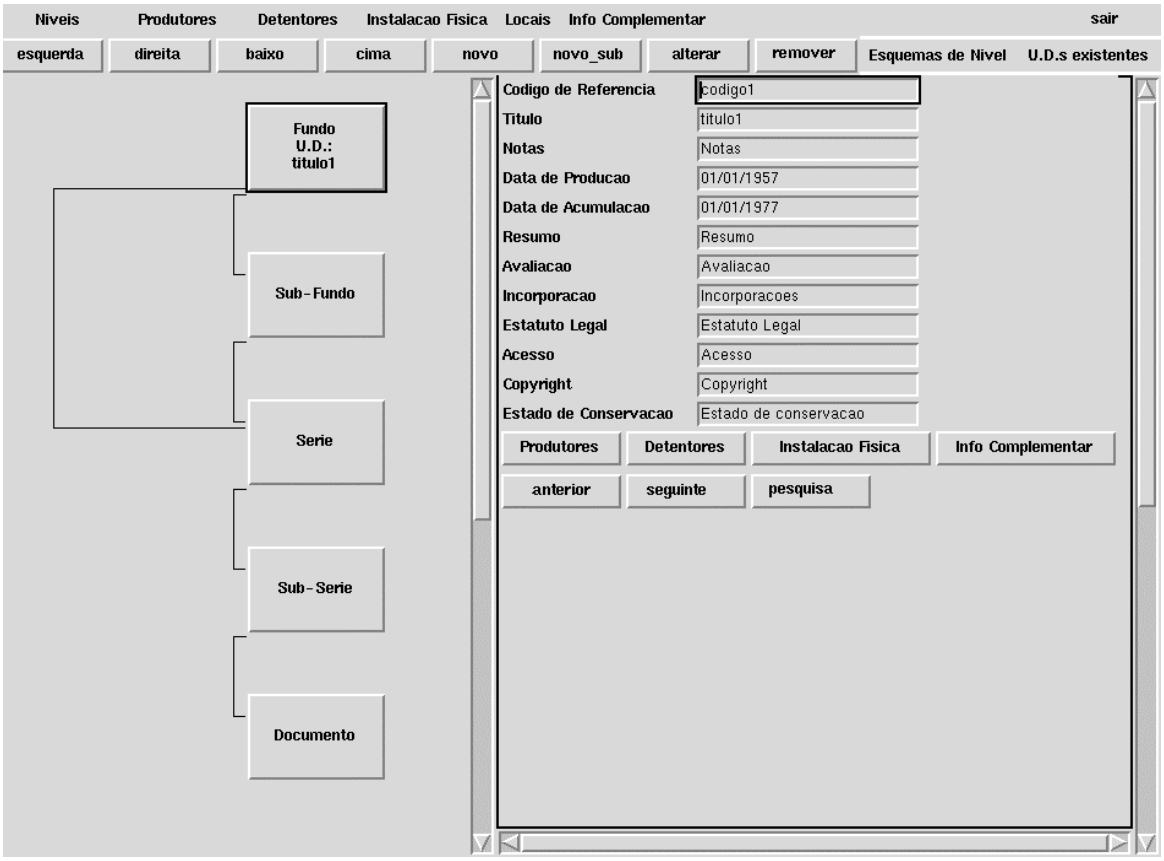


Figura 4.8: Janela principal da aplicação de descrição arquivística

Construção de Esquemas de Níveis

A construção de um novo esquema de níveis pode efectuar-se recorrendo ao menu *Níveis* da janela principal da aplicação. Através deste, tem-se acesso a uma ferramenta de construção de um novo esquema. Quando esta opção é seleccionada, é pedida ao utilizador a seguinte informação:

- Qual a designação do novo esquema;
- Qual o número de níveis máximo que o esquema terá;
- A descrição de cada nível desde o nível de topo até ao nível mais baixo.

Após a introdução dessa informação aparecem os níveis representados graficamente com a designação do esquema e a descrição de cada nível, sendo atribuído automaticamente a cada nível uma numeração e uma ligação entre níveis consecutivos. Nesta altura o utilizador pode acrescentar e retirar ligações entre níveis não consecutivos, indicando o nível de origem e o nível de destino. Pode paralelamente seguir a evolução da estrutura de níveis graficamente.

Após terminar a construção do novo esquema de níveis, deve pressionar o botão de gravar esquema, para voltar à janela principal da aplicação.

Navegação na Base de Dados

Dada a natureza da base de dados de descrição arquivística, uma das operações dominantes na utilização da aplicação será a de selecção de uma unidade de descrição de topo e a partir daí começar a descer na hierarquia de níveis. Quando se está a efectuar esta operação, se a unidade de descrição actual não for de topo, está sempre subordinada a uma unidade de descrição de um nível superior na hierarquia.

Não é possível, por navegação, chegar a uma unidade de descrição sem passar pela cadeia de unidades de descrição que lhe é hierarquicamente superior. A única excepção a esta regra é através de uma interrogação à base de dados.

Quando se está a efectuar operações deste tipo, na parte esquerda da janela principal aparece sobressaído o caminho seleccionado pelo utilizador e o nível actual. O utilizador pode também consultar relativamente ao nível actual, a lista das unidades de descrição correspondentes que são subordinadas à unidade de descrição anterior.

O utilizador pode navegar no esquema de níveis através dos botões nível anterior (*cima*) e próximo nível (*baixo*) e pode navegar no próprio nível (*esquerda* e *direita*) visualizando as unidades de descrição desse nível que estão subordinadas à mesma unidade de descrição.

Actualização da informação respeitante a Unidades de Descrição

O utilizador tem acesso ao conteúdo das caixas de texto correspondentes à informação sobre a unidade de descrição seleccionada, podendo aí fazer alterações. Essas alterações podem reflectir-se na unidade de descrição seleccionada, caso o utilizador pressione o botão *alterar*. Para inserir uma nova unidade de descrição, o utilizador tem duas hipóteses:

- Inserir a nova unidade de descrição como irmã da corrente (*novo*);
- Inserir a nova unidade de descrição como filha da corrente (*novo_sub*).

A remoção de uma unidade de descrição (*remover*), implica a remoção de todas as unidades de descrição que lhe são directa ou indirectamente subordinadas.

A informação complementar respeitante a uma unidade de descrição pode também sofrer as operações de inserção, alteração e remoção, estando obrigatoriamente ligada àquela.

Actualização de outra informação

A informação respeitante aos subesquemas: Produtores, Detentores, Instalação Física, Locais e Informação Complementar pode ser gerida através dos respectivos menus na janela principal. Não são permitidas operações de remoção de entidades que se encontrem ligadas a unidades de descrição. Para que tal seja possível, é necessário que seja quebrada a associação com as unidades de descrição ligadas.

Interrogações à Base de Dados

Usando interrogações à base de dados, é possível chegar a uma unidade de descrição arbitrária sem ser necessário percorrer a sua cadeia hierárquica. A realização de pesquisas em unidades de descrição pode ser efectuada usando as caixas de texto onde é visualizada a respectiva informação. Depois de preenchidas, pressionando o botão *pesquisa*, é possível visualizar as

unidades de descrição uma a uma que constituem o resultado da interrogação. Para tal devem ser utilizados os botões *anterior* e *seguinte*.

É ainda possível pesquisar informação sobre produtores, detentores e instalação física associados a uma unidade de descrição através dos respectivos menus da aplicação.

4.4.3 Carregamento da Base de Dados

O processo de carregamento da base de dados da aplicação teve como base um conjunto de ficheiros que constituíam uma amostra respeitante ao acervo do Arquivo Distrital do Porto (A.D.P.).

Este arquivo tem a informação descritiva do seu conteúdo num formato que exige o recurso a um processo de conversão para adequação da informação à base de dados do sistema de descrição arquivística desenvolvido.

A informação foi recebida em ficheiros no formato ISO 2709 para troca de informação. Esta norma organiza a informação em registos. Cada registo é composto por três segmentos lógicos, tendo os dois primeiros funções de controlo. O primeiro é de tamanho fixo, e o segundo tem um número variável de campos de tamanho fixo identificando o código, comprimento e localização de cada campo de informação no registo. Este segmento termina com um separador de campo. O terceiro segmento contém a informação propriamente dita na forma de campos de informação alfanumérica de comprimento variável. No fim de cada campo de informação está colocado um símbolo terminador de campo, e no fim de cada registo um símbolo terminador de registo.

Cada registo possui um campo, cujo código tem o valor 099, onde está localizada a natureza do registo. Os códigos dos registos, contemplados no grupo de ficheiros que constituiu a amostra, podem ser visualizados na tabela seguinte. A coluna mais à direita, identifica qual a entidade ou entidades correspondentes no sistema de descrição arquivística desenvolvido.

Valores possíveis para campos com código 099	Descrição do A.D.P.	Entidade(s) equivalente(s) no sistema de descrição arquivística
0.0	Entidade Detentora	Detentor
1.0	Grupo de Arquivos	Unidade Descrição
2.0	Fundo	Unidade Descrição
2.2	Secção	Unidade Descrição
2.5	Subsecção	Unidade Descrição
3.0	Série	Unidade Descrição
3.5	Subsérie	Unidade Descrição
4.0	Unidade de Instalação	Unidade Descrição, Unidade Instalação, Localização, Prateleira, Estante, Corpo e Depósito.
5.0	Peça	Unidade Descrição, Unidade Instalação, Matéria Scriptória, Notas de Publicação e Publicação

Cada campo dos registos, pode ainda ser dividido em subcampos. Esta divisão representa um nível interior à camada ISO 2709. A título ilustrativo é representada em baixo a estrutura possível de um registo *Grupo de Arquivos*:

Nível	Campos	Subcampos
Grupo de Arquivos	201	^a Código de Referência do Grupo de Arquivos ^b Nome do Grupo de Arquivos
	208	^a Designação do Grupo de Arquivos

Figura 4.9: Estrutura de um registo com código Grupo de Arquivos

Cada registo pode ou não conter todos os campos definidos para o seu tipo. O modo como se indica que um registo deriva de outro de nível mais alto é através da indicação dos subcampos chave dos registos superiores no registo mais baixo.

Exemplo 4.1: Um registo Fundo poderia ter um campo com código 201, dentro deste um subcampo ^a com o valor "GA NOT" indicando que esse fundo pertencia ao grupo de arquivos notariais. □

A solução adoptada na aplicação de conversão está esquematizada na figura seguinte.

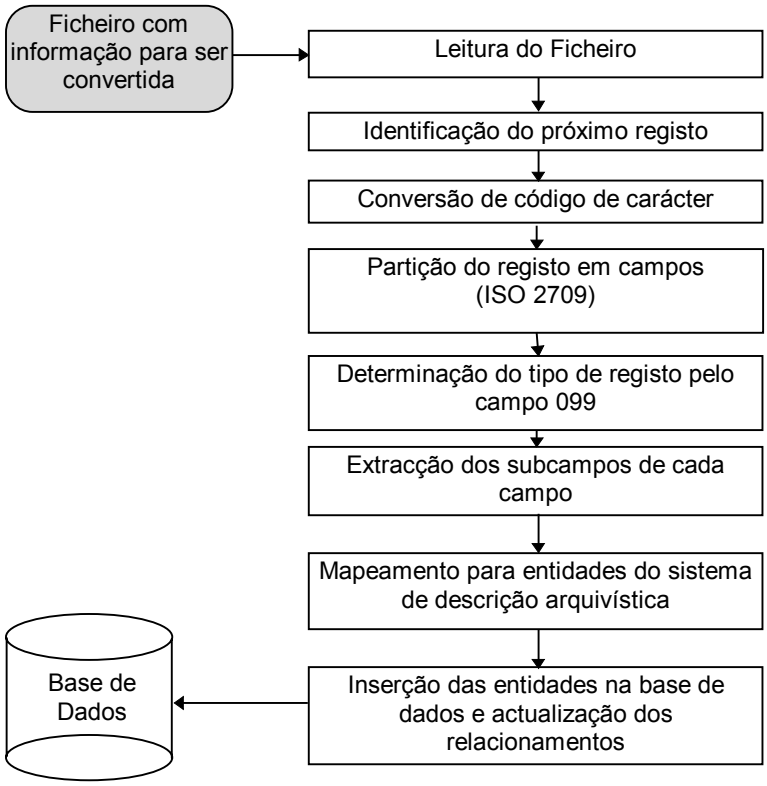


Figura 4.10: Diagrama da aplicação de conversão

A aplicação verifica qual o tipo de registo e quais os registos que lhe são hierarquicamente superiores, interroga a base de dados pela entidade que lhe está imediatamente acima e insere a(s) nova(s) entidade(s) na base de dados actualizando também os relacionamentos de modo a manter a informação sobre o esquema de níveis na base de dados.

4.5 Conclusões

A realização de um projecto de software para automatização de um sistema de descrição arquivística coloca muitos desafios. A aplicação descrita neste capítulo assume o carácter de protótipo, como tal não constitui uma versão estável, mas antes uma direcção possível para solucionar o problema proposto. Por esse motivo, este projecto enquadra-se naturalmente num projecto mais ambicioso com horizontes temporais diferentes deste.

Várias linhas de trabalho ficam em aberto. Enumera-se a seguir as que parecem ser as principais:

- Afinação da base de dados;
- Refinamento da aplicação;
- Disponibilização da base de dados para consulta pela Internet, assim como de aplicações clientes escritas em Java para consulta da mesma;
- Adaptação da base de dados e aplicações clientes a novos pacotes de normas ISAD;
- Inserção de um maior nível de funcionalidades na aplicação.

Visto as normas ISAD descritas neste capitulo serem relativamente recentes, salienta-se também a possibilidade de uma aplicação deste género ser de grande interesse para a realidade arquivística a nível nacional, constituindo um modo de simultaneamente se abrir às novas tecnologias de informação e alinhar pela regulamentação sectorial que rege especificamente a prática de descrição arquivística com as vantagens inerentes.

Quanto à questão da selecção do modelo de dados usado na aplicação: a existência de um elevado número de relacionamentos entre as entidades do modelo e a natureza hierárquica da organização das unidades de descrição, conduziu à escolha de um SGBDOO para a aplicação.

No entanto, o relativamente baixo dinamismo esperado para as operações de actualização da base de dados constitui um aspecto que poderia levar à escolha de uma implementação usando o modelo relacional.

5 Comparação de tecnologias

Após a apresentação no segundo e terceiro capítulos, do modelo relacional e do modelo orientado por objectos respectivamente, procura-se no presente efectuar a comparação dos dois modelos.

Inicia-se com uma vista sobre as novas aplicações de bases de dados e requisitos que essas implicam. As limitações dos sistemas relacionais conduziram ao desenvolvimento de extensões para o modelo ou levaram a outras soluções que permitissem um sistema relacional suportar as novas aplicações de bases de dados. Essa vertente evolutiva do modelo relacional e do seu standard de facto - SQL, são abordadas a seguir.

As linguagens de interrogação de bases de dados orientadas por objectos, deparam-se com especificidades do modelo em si. Assim, procede-se à enumeração de pontos que distinguem uma linguagem de interrogação de um SGBDOO e de um SGBD relacional. Aborda-se também a convergência da linguagem de interrogação - OQL com SQL3.

A análise dos dois modelos de um ponto de vista comparativo e focando alguns pontos específicos e o retirar de algumas conclusões encerram o capítulo.

5.1 Evolução das Necessidades das Aplicações de Bases de Dados

Durante a década de setenta, a comunidade cientifica ligada a bases de dados, investigou intensamente o conceito de modelo relacional, nomeadamente: linguagens de interrogação de alto nível, optimização de interrogações, teoria de normalização, estruturas de memorização da informação e estratégias de acesso. Os sistemas relacionais comerciais resultantes desse esforço vulgarizaram-se, estando disponíveis para quase todas as plataformas de hardware existentes e dominando o mercado de bases de dados.

Os sistemas de bases de dados relacionais têm sido, ao longo da sua existência, utilizados no desenvolvimento de aplicações em diversas áreas, sendo a sua predominância em ambientes empresarias como sistemas de suporte à decisão. Oferecendo um modo de armazenamento da informação poderoso, ferramentas para facilitar a formulação de interrogações e apresentação da informação da base de dados, e mecanismos de gestão de transacções que possibilitam encarar grupos de operações de uma forma atómica, potenciaram a obtenção de ganhos de produtividade.

A solidez, simplicidade e facilidade de utilização do modelo relacional, constituíram factores importantes para a sua adopção por numerosos SGBDs.

Pode-se, então, enumerar algumas características, geralmente requeridas em sistemas e aplicações tradicionais baseadas no modelo de dados relacional:

- Armazenamento de grandes quantidades de informação;
- Linguagens de interrogação simples;
- Ferramentas de desenvolvimento de formulários de ecrã e impressão;
- Controlo de concorrência fino;
- Recuperação de falhas;
- Independência dos dados;
- Distribuição dos dados por várias bases de dados.

A evolução tecnológica verificada desde então reflecte-se em organizações das mais variadas dimensões, que incessantemente procuram empregar tecnologias de bases de dados na construção de aplicações com requisitos cada vez mais elaborados.

Em [50] são apontados aspectos importantes que influenciarão o futuro das bases de dados. Um número substancial de tecnologias avançadas dependerá de novas tecnologias de bases de dados ainda não completamente compreendidas. A próxima geração de bases de dados terá pouco em comum com as aplicações de bases de dados convencionais usadas a nível empresarial. Estas envolverão um muito maior volume de informação, novas capacidades como extensão de tipos, suporte para multimédia, objectos complexos, processamento de regras e necessitarão de um repensar dos algoritmos usados em quase todas as operações de bases de dados. A cooperação entre diferentes organizações em problemas científicos, de engenharia e comerciais, irá requerer bases de dados distribuídas, heterogéneas e escaláveis. Isto trará a necessidade de lidar com questões relacionadas com inconsistência e segurança de bases de dados.

A lista seguinte, sem ser exaustiva, procura nomear algumas áreas cujas aplicações apresentam especificidades para além das aplicações tradicionais:

- Engenharia de Software Assistida por Computador (*Computer Aided Software Engineering* CASE): projecto, especificação, implementação, análise, depuração e evolução de programas de computador e sua documentação [49];
- Desenho Mecânico Assistido por Computador (*Mechanical Computer Aided Design* - MCAD) e Desenho Electrónico Assistido por Computador (*Electronics Computer Aided Design* - ECAD): o primeiro pode contemplar desde aplicações para projectos de automóveis e aeronaves até aplicações para projectos de edifícios. No segundo grupo incluem-se aplicações de projecto de circuitos integrados, placas PCB e outros sistemas electrónicos [30];
- Automatização de Escritório, automatização do sistema de informação como utilização de correio electrónico ou gestão de documentos [7];
- Edição Assistida por Computador e Hipertexto: gestão de documentos complexos;
- Aplicações Médicas e Científicas: representações complexas de informação química, biológica ou física;
- Fabricação Assistida por Computador (*Computer Aided Manufacturing - CAM*);

- Sistemas de Análise de Falhas [36];
- Sistemas de tempo real: controlo de processos por computador;
- Aplicações de Telecomunicações [63].

O próprio contexto tecnológico em que se enquadram as aplicações de bases de dados actuais faz com que à partida os requisitos de bases de dados destas sejam mais ambiciosos. Na aplicação de descrição arquivística, exposta no capítulo anterior, é equacionada a hipótese de disponibilização da informação da base de dados através da Internet. Este facto implica características de extensibilidade e escalabilidade que se esperam de uma base de dados deste tipo.

Novas características surgem como desejáveis pelas aplicações de bases de dados para áreas como as referidas acima e outras. Procura-se a seguir identificar algumas:

- Identificadores únicos de objectos: por exemplo, nas aplicações de engenharia e desenho os oid's gerados automaticamente podem representar objectos que não apresentam nenhum atributo de identificação única evidente para um ser humano, ou cuja identificação pode ser alterada (como, por exemplo, uma referência de uma peça);
- Objectos compostos: em aplicações de desenho é muitas vezes necessário definir objectos que contêm outros objectos como subcomponentes;
- Referências, integridade referencial, e relacionamentos: as novas aplicações apresentam modelos conceptuais mais elaborados que se servem intensamente de referências entre objectos. É importante em todo o universo de aplicações que o SGBD mantenha a integridade dessas referências automaticamente;
- Hierarquia de tipos: as aplicações que apresentam esquemas mais complexos beneficiam com a possibilidade de definir tipos de objectos que têm todos os atributos de um tipo existente e além disso acrescentam os seus próprios atributos ou métodos;
- Procedimentos associados aos dados: a capacidade de armazenar comportamento além de dados na base de dados aumenta a semântica da informação na base de dados, sendo este facto útil a muitas das novas aplicações;
- Encapsulamento de objectos: reforça a importância da reutilização, modularização e interoperabilidade, factores importantes na maioria das aplicações;
- Conjuntos e referências ordenadas: é útil por vezes armazenar uma ordem nos relacionamentos de um objecto. O mesmo acontece para colecções com ordenação (Listas ou vectores) que são suportadas de modo pouco natural em sistemas relacionais;
- Armazenamento de imagens ou outros blocos de dados de grandes dimensões: atributos de grandes dimensões são cada vez mais vulgarmente definidos dentro de objectos;
- Acesso remoto eficiente: com a vulgarização dos sistemas distribuídos é cada vez mais comum o acesso transparente a uma base de dados localizada noutro computador da rede. No entanto a articulação deste com aplicações que manipulam a informação de uma porção da base de dados durante longos períodos de tempo torna-se critica;

- Facilidade de alteração e evolução do esquema: o oferecer de capacidades para suporte da migração da informação do esquema actual para um novo esquema de bases de dados é cada vez mais importante nalgumas aplicações como por exemplo aplicações de desenho;
- Interface com Linguagem de Programação: a redução do fosso semântico e da desadaptação de impedâncias entre o domínio da aplicação e a modelização da base de dados para esse domínio é importante no desenvolvimento de aplicações que necessitam do serviço de persistência;
- Múltiplas versões da base de dados assim como configurações: esta possibilidade põe-se principalmente em aplicações de desenvolvimento, de desenho ou de projecto em que é importante manter informação sobre as diversas versões dos objectos;
- Bloqueios e transacções longas: a capacidade de impedir que uma parte de informação da base de dados (por exemplo, um objecto composto ou todo um projecto) seja alterada é importante em muitas aplicações de desenvolvimento, de desenho ou de projecto;
- Desempenho em ambiente de mono-utilização: em aplicações de desenvolvimento, de desenho ou de projecto, o utilizador pode estar durante longos períodos de tempo a trabalhar sobre uma porção da base de dados, bloqueando essa zona da base de dados para escrita por outros utilizadores.

Apesar de nessas aplicações os requisitos focados em cima se fazerem sentir, não invalida que esses e outros campos de aplicação não continuem a endereçar os requisitos presentes nas aplicações tradicionais. Portanto, a implementação dos requisitos listados acima, não deverá ser efectuada com prejuízo dos citados para aplicações tradicionais.

5.2 Evolução a partir do Modelo Relacional

Duas razões, não desprezáveis, para a utilização dos sistemas relacionais como ponto de partida para os sistemas futuros são: a manutenção e compatibilidade com a numerosa base de aplicações tradicionais existentes sobre sistemas relacionais; e a fácil adaptação e aprendizagem por parte dos utilizadores, programadores e administradores de sistemas relacionais.

A seguir, procura-se fazer uma súmula da actividade relacionada com a tentativa de adaptar sistemas relacionais a novas realidades aplicacionais.

5.2.1 Relações NF2

A normalização de dados em sistemas relacionais, apresentada no fim do segundo capítulo, constitui uma barreira para a definição de atributos não atómicos em colunas de tabelas como sejam estruturas ou colecções de dados. As relações que violam esta regra do modelo relacional, isto é, que não se encontram na primeira forma normal designam-se por NF2 (*Non First Normal Form*). Nestes sistemas é permitido que um atributo tenha uma relação como seu domínio. Os sistemas que suportam relações NF2 implementam variações da linguagem SQL, com vista a incorporar esta nova característica.

Em [26] é apresentado, como motivação para a utilização de sistemas que suportem esta característica, o facto de esta ser uma representação que evita a obrigação de definição, por

decomposição, de novas tabelas sempre que sejam considerados atributos compostos ou atributos colecção (por exemplo as linhas de uma factura) na relação original. No mesmo artigo, são definidas árvores de n-tuplos (*tuple trees*) que contêm toda a informação de um n-tuplo particular da base de dados de uma forma comprimida. O formato destas árvores de n-tuplos é definido usando esquemas de árvore (*schema trees*). As árvores de n-tuplos representam informação resultante da junção de todas as tabelas descritas no respectivo esquema de árvore. A adopção de árvores de n-tuplos permite a manipulação de conjuntos completos de linhas relacionadas com uma operação simples. É também disponibilizado um mecanismo para a visualização gráfica de uma árvore de n-tuplos usando Tcl/Tk.

5.2.2 Ligação entre Linguagens de Programação e SGBDs Relacionais

Um modo de oferecer o serviço de persistência a uma aplicação desenvolvida numa linguagem orientada por objectos é a adopção de um mecanismo que faça a ponte entre a aplicação e um sistema relacional [40], [37]. Algumas das vantagens advogadas para estes sistemas são a possibilidade de desfrutar, simultaneamente, dos benefícios de produtividade, qualidade e reutilização de aplicações orientadas por objectos e ao mesmo tempo facilitar o acesso à informação através de um modelo bem definido. Contudo, a ligação de dois modelos distintos leva a dificuldades de mapeamento de conceitos do modelo orientado por objectos para os sistemas relacionais. Como exemplo, temos o conceito de herança, nomeadamente herança múltipla.

5.2.3 Extensão dos Modelos Relacionais

A opção de extensão de sistemas relacionais com novas características, para suporte dos requisitos de aplicações não tradicionais, tem sido uma das principais abordagens para dotar sistemas de bases de dados com potencial para lidarem com as exigências aplicacionais actualmente existentes.

Assim, foi estimulado o aparecimento de projectos de investigação com o objectivo de desenvolver novos sistemas de bases de dados, que simultaneamente mantêm o que de melhor existe nos sistemas relacionais e incorporam facilidades requeridas pelas novas aplicações.

Entre as características de sistemas relacionais que se pretendiam manter intactas nos novos sistemas temos: a possibilidade de efectuar interrogações de um modo associativo usando uma linguagem declarativa; optimização de interrogações; realização de operações algébricas sobre conjuntos; e independência lógica da informação.

Em [53] é explorado um mecanismo para suporte de tipos definidos pelo utilizador em colunas de sistemas de bases de dados relacionais, de modo a enfrentar a necessidade de tipos de dados mais complexos e suas operações em aplicações como sistemas geográficos, aplicações científicas e mesmo em certos casos aplicações de negócios.

Na segunda metade da década de oitenta, foram desenvolvidos sistemas como Postgres [54] e Starsburst [25] mais poderosos que os sistemas relacionais antecedentes.

Em 1986, surge o sistema de gestão de bases de dados Postgres que focaliza o seu modelo de dados em torno de quatro conceitos: classes, herança, tipos e funções. Uma classe constitui uma colecção de objectos, possuindo cada objecto um identificador único (OID). A uma classe está associada uma designação, podendo herdar elementos de outras classes. Os tipos de dados em Postgres podem ser tipos base, vectores de tipos base e tipos compostos, sendo suportada a

definição de novos tipos base pelo utilizador. Postgres permite também a definição de funções escritas, usando a sua linguagem de interrogação ou outra linguagem de programação, podendo ser executadas directamente ou através da linguagem de interrogação. A interacção com a base de dados é efectuada primordialmente através de POSTQUEL, uma linguagem de interrogação declarativa, constituindo uma extensão das linguagens de interrogação relacionais com capacidade de definição de interrogações embutidas em operadores que usam conjuntos como operandos e cálculo da operação de fecho transitivo. Postgres implementa também um sistema de regras geral, que suporta a noção de acções despoletadas por eventos (*triggers*), gestão de vistas, definição de restrições de integridade, garantia de integridade referencial e controlo de versões.

O sistema de base de dados Starburst, desenvolvido pela IBM, tinha como objectivo a construção de um SGBD relacional que pudesse ser facilmente estendido para lidar com aplicações e tecnologias não suportadas pelos sistemas relacionais de então. Starburst consiste num processador de linguagem de interrogação (Corona) e um gestor de dados (Core). Corona compila interrogações expressas numa linguagem, que constitui uma extensão de SQL designada por Hydrogen, para um conjunto de chamadas dos serviços fornecidos pelo gestor de dados Core para aceder e modificar informação da base de dados. A expansibilidade exige um nível de conhecimento elevado do sistema assim como dos requisitos da aplicação a que este se destina. Extensões possíveis podem ser novos tipos de dados e operadores, suporte para objectos complexos, definição de sistemas de regras, novos métodos de acesso à base de dados ou estruturas de armazenamento externo.

As duas abordagens aos sistemas relacionais estendidos são de certa forma distintas. Postgres oferece uma extensão das capacidades de um sistema relacional com novas possibilidades, como por exemplo, suporte de objectos complexos, pronto a usar. Starburst assenta no conceito de disponibilização de uma unidade básica a partir da qual um sistema pode ser construído, sendo essa tarefa adstrita a um utilizador que deverá ser capaz de projectar e construir o sistema depois de fixados certos parâmetros.

5.2.4 Terceiro Manifesto

Em [17] no seguimento de ([3] e [57]), e com o intuito de substituir estes, C. J. Date e H. Darwen apresentam um manifesto para o futuro dos sistemas de gestão de bases de dados. Avançam com um conjunto de prescrições e proscrições ao modelo relacional, assim como um modelo possível para utilização de herança.

Referem que SQL constitui uma perversão do mesmo e por isso deve ser abandonada, revelando características de uma nova linguagem que a deverá substituir. A SQL apontam os seguintes aspectos como desvios do modelo relacional:

- Permissão de utilização de atributos anónimos como `SELECT A + B FROM R`, ou a duplicação de atributos em expressões de selecção como `SELECT R1.A, R2.A FROM R1, R2`, ou a possibilidade de definição de expressões abreviadas dependentes da ordem dos atributos como `SELECT * FROM R` ou `INSERT INTO R VALUES (...);`
- Existência de construções que dependem da definição de uma ordem para os n-tuplos de uma relação. A imposição de uma ordem, por exemplo na apresentação do conteúdo de uma relação, deveria implicar a sua conversão para algo que não é uma relação;

- Permissão de n-tuplos repetidos, de valores nulos, de operações sobre n-tuplos e de operações relacionadas com questões de nível físico.

Com o intuito de estabelecer fundações firmes para o futuro das bases de dados, devendo estas estar assentes no modelo relacional, os autores apontam também que aspectos como, tipos de dados definidos pelo utilizador e mecanismos de herança, devem ser considerados e suportados pelas bases de dados do futuro. No entanto, estes aspectos, referem ainda os autores, são ortogonais ao modelo relacional, ou seja, não implicam nenhuma extensão, correcção ou perversão deste modelo. Estipulam ainda um conjunto de prescrições e proscrições para alguns aspectos ortogonais ao modelo (caso de herança), assim como um conjunto de sugestões que deverão ser levadas em conta quer para o próprio modelo relacional quer para os aspectos ortogonais ao modelo.

Os autores defendem um conceito lato em relação ao conceito de domínio suportado pelo modelo relacional, à medida das linguagens de programação, advogam que um domínio ou um tipo deve estar encapsulado, ou seja:

- A representação de valores desse domínio deve estar escondida do utilizador;
- Os valores do domínio devem ser operados apenas por operadores definidos para esse domínio, podendo esses operadores ser de comparação, mas não só.

Esta noção de tipo corresponde à de classes de objectos de sistema orientados por objectos. Os autores defendem assim que o conceito no mundo relacional correspondente ao conceito de classes de objectos deverá ser: *Domínio = Classe de Objecto* e não *Relação = Classe de Objecto*. A herança deve ser definida, segundo os autores, em domínios e não em tipos linha como considerado por SQL3 (*ver ponto 5.3*) e pelos principais produtos de bases de dados que oferecem características orientadas por objectos em sistemas relacionais.

5.2.5 Evolução de Sistemas Relacionais Comerciais

Com a explosão da utilização da Internet nos meios empresariais, e da WWW como plataforma de aplicação, os utilizadores que desenvolvem aplicações para Internet, sentem ainda mais a necessidade de um ambiente robusto onde armazenar e manipular informação multimédia e outros tipos complexos de dados para serem utilizados de uma forma dinâmica. Como resultado dessa exigência premente, as empresas fabricantes de SGBDs relacionais, procuraram acelerar o seu processo de extensão dos seus produtos relacionais para suporte de novas características. Essas evoluções da arquitectura, levaram ao surgimento dos designados *servidores universais* ou *bases de dados universais*. Empresas como a Informix com o *Informix - Universal Server*, IBM com o *IBM DB2 - Universal Server* e Oracle com o *Oracle 8* caminham neste sentido, procurando oferecer uma única plataforma de base de dados que esteja preparada para dados complexos e que continue a dar suporte para a informação e aplicações existentes.

Os sistemas estendidos podem suportar tipos definidos pelo utilizador, quer ao nível de coluna, quer ao nível de linha. Tipos definidos pelo utilizador ao nível de coluna, designam-se por Tipos Distintos ou Tipos de Dados Abstractos (TDA). Os primeiros constituem uma extensão simples dos tipos básicos do sistema numa coluna, podendo ter uma designação que os inibirá, num sistema fortemente tipado, de serem comparados directamente com outros tipos que, embora derivados do mesmo tipo básico, tenham outra designação.

Os tipos de dados abstractos definem tipos de dados mais complexos, podendo albergar atributos como imagem, texto ou outros tipos de dados elaborados. A informação em TDAs é manipulada usando um conjunto de atributos e funções externas. São definidos usando SQL ou uma linguagem hospedeira.

Os Tipos Linha, descrevem uma linha ou um conjunto de colunas embutidas, numa tabela, e constituem um modo de representar entidades de uma forma hierárquica na base de dados, permitindo identificar múltiplas colunas relacionadas entre si. Os tipos referência podem definir relacionamentos entre tipos linha e identificar univocamente uma linha na base de dados. As referências substituem a definição de junções complexas em interrogações por expressões de travessia mais simples.

Outra característica possível nestes sistemas é o suporte de construtores de tipos de colecções usados para definir vectores, listas e conjuntos de outros tipos de dados. As colecções podem ser usadas para armazenar múltiplos valores numa coluna de uma tabela, podendo resultar em tabelas embutidas noutras tabelas.

Juntamente com os tipos definidos pelo utilizador, este pode ainda definir funções para manipulação de dados. Um SGBD relacional estendido deve ainda permitir que funções definidas pelo utilizador possam retornar valores complexos como tabelas que depois sejam manipuláveis.

Outras extensões importantes são o suporte para objectos de grandes dimensões, quer dentro da base de dados quer fora em ficheiros externos, a possibilidade de definição de regras e restrições de integridade aos novos tipos de dados e interrogações recursivas. O sistema estendido deve também suportar o standard SQL3 (ver a seguir), assim como outras linguagens para escrita de funções definidas pelo utilizador e procedimentos armazenados, como linguagens de 3ª geração e Java.

Os SGBDs estendidos assumem, assim, cada vez mais o papel de plataformas de desenvolvimento, onde é possível a construção de extensões pré-definidas para tratamento de tipos de dados mais complexos. Entre estas, pode-se citar a titulo de exemplo extensões para tratamento de texto, imagem, vídeo, audio e outras.

Em [19] procede-se a uma descrição de servidores universais e à comparação de alguns produtos existentes ou em fase de lançamento.

5.3 SQL3

A evolução para uma nova versão da norma SQL - SQL3, publicada conjuntamente pela ANSI e ISO que trabalham em cooperação, vem na sequência da disponibilização das versões SQL-86, SQL-89 e SQL-92. Esta versão revela-se importante para SGBDs que procuram estender o modelo relacional com novas características.

O objectivo subjacente centra-se na disponibilização de um modelo de dados mais poderoso, permitindo assim a representação de dados que não se adequam ao formato tabular tradicional das relações, e a possibilidade de definição de procedimentos, promovendo o SQL a linguagem computacionalmente completa, à imagem de linguagens de programação tradicionais.

SQL3 inclui novas características, das quais algumas das mais significativas são: hierarquias de generalização e especialização, herança múltipla, tipos de dados definidos pelo utilizador, sistema de regras activas - *triggers* e possibilidade de expressão de interrogações de forma recursiva.

Em 1993, os comités de desenvolvimento da ANSI e da ISO, decidiram dividir o processo de desenvolvimento num standard multi-partes. As partes definidas actualmente são [31]:

Parte 1: *Framework*, descrição não técnica de como o documento está estruturado.

Parte 2: *Foundation*, especificação do núcleo, incluindo as capacidades para Tipos de Dados Abstractos.

Parte 3: *SQL/CLI*, interface de programação para proporcionar conectividade à base de dados a partir de aplicações.

Parte 4: *SQL/PSM*, a especificação dos procedimentos armazenáveis, incluindo a completude computacional.

Parte 5: *SQL/Bindings*, o SQL dinâmico e SQL embutido como evoluções do SQL-92.

Parte 6: *SQL/XA*, uma especialização do SQL para a interface XA da X/Open.

Parte 7: *SQL/Temporal*, adiciona capacidades ligadas a problemas temporais.

Parte 8: *SQL/Extended Objects*, define extensões dos tipos existentes em SQL.

Parte 9: *SQL/Virtual Tables*, define extensões para o SQL suportar tabelas abstractas.

Está previsto que o processo de normalização das partes 1, 2 e 5, dê origem a um standard ISO/IEC em finais de 1998. O horizonte temporal para a disponibilização de todo o standard, aponta para o ano de 1999.

A colaboração dos organismos de normalização ANSI e ISO no desenvolvimento do novo standard implica que os avanços registados por ambas as entidades tenham de ser fundidos periodicamente, havendo uma geração contínua de propostas de alteração, que são objecto de aprovação pelos dois organismos. Este processo provoca o aparecimento nos documentos de carácter provisório que compõem a norma - *Working Drafts*, de partes propostas por um dos organismos mas ainda não aceites pelo outro. Acontecem também situações em que os organismos têm propostas diferentes para o mesmo problema.

5.3.1 Tipos de Dados Abstractos

Uma das ideias básicas subjacentes à extensão para objectos de SQL é que, para além dos tipos primários definidos por SQL, seja também possível a definição de tipos de dados pelo utilizador.

Um tipo de dados definido pelo utilizador pode ser *Tipo Distinto*, representando um tipo cujas instâncias são derivadas de instâncias de valores de um tipo já existente ou tipo origem. Estes tipos podem ser utilizados para a partir de um tipo existente obter outro tipo por conversão do tipo original usando - CAST.

O segundo tipo definido pelo utilizador é o *Tipos de Dados Abstractos* [32]. Este pode ser usado do mesmo modo que os tipos primários SQL.

Um TDA suporta encapsulamento como definido nos sistemas orientados por objectos. Os TDAs estão vocacionados para serem usados como componentes de n-tuplos em tabelas, e não para serem eles próprios n-tuplos [59].

Por exemplo, colunas em sistemas relacionais, podem ser definidas como tendo valores de tipos definidos pelo utilizador, para além de tipos do sistema.

A definição de TDA encapsula atributos e operações na mesma entidade. Em SQL3 um TDA é definido pela especificação de um conjunto de declarações dos atributos armazenados que representam o valor do TDA, um conjunto de operações que definem a relação de igualdade e outros métodos de ordenamento do TDA, e operações que representam o comportamento do TDA. As operações são implementadas através de procedimentos designados por *rotinas*.

Uma especificação de um TDA pode conter [32]:

- O seu nome;
- Uma indicação da especificação de ordem para os TDAs, como: EQUALS ONLY ou ORDER FULL seguidos do método de ordenamento;
- O método de ordenamento do TDA, que pode ser: RELATIVE, é indicada uma função de ordenamento relativo; HASH, é indicada uma função de disprsão; EQUALS, é indicado o nome da função para determinar a igualdade;
- Os nomes dos TDAs supertipos directos do TDA em causa, se existirem;
- A descrição de cada atributo do TDA;
- O grau do TDA (número de atributos descritos);
- A descrição de cada operação que tem o TDA como parâmetro ou resultado;
- O valor da opção por defeito, se existir.

Exemplo 5.1: um exemplo de declaração de TDA adaptado de [42] e [11] é:

```
CREATE ABSTRACT DATA TYPE tipo_empregado UNDER tipo_pessoa
CONSTRUCTOR c_empregado
(
nome VARCHAR NOT NULL,
idade INTEGER,
endereço VARCHAR,

PRIVATE
data_nascimento DATE CHECK (data_nascimento < '1995-01-01'),

PUBLIC
EQUALS ONLY BY STATE,

departamento tipo_departamento,
salário_anual INTEGER,

  FUNCTION c_empregado (E tipo_empregado, N VARCHAR, D DATE, S
INTEGER) RETURNS tipo_empregado;
  BEGIN
    SET E.nome = N;
    SET E.data_nascimento = D;
    SET E.salário_anual = S;
  END FUNCTION,

  PROCEDURE contrata_e (IN P tipo_pessoa);
    -- operações de criação de instância de tipo_empregado
```

```
  -- a partir de uma pessoa
  ...
END,

FUNCTION salário_mensal (IN P tipo_empregado) RETURNS REAL;
  -- calcula e retorna salario mensal
  -- de um empregado
  ...
END,

PROCEDURE despede_e (P tipo_empregado);
  -- operação de remoção do empregado
  ...
END
)
TYPE DEFAULT NULL;
```

O exemplo anterior permite visualizar a utilização de componentes (atributos ou rotinas) privados e públicos. Os primeiros apenas podem ser usados em rotinas encaixadas dentro da definição do TDA. O primeiro atributo de um TDA, se nada for especificado em contrário, é público. Existe um terceiro nível de encapsulamento para além de público e privado designado pela palavra reservada PROTECTED. Componentes de um TDA que estão sob o nível PROTECTED, estão parcialmente encapsulados, são visíveis de dentro do seu TDA e de todos os subtipos desse TDA.

Todos os atributos de um TDA, têm implicitamente definidas duas funções para cada atributo, ambas com o mesmo nome do atributo. Estas designam-se por: *Observer* e *Mutator*. A primeira, tem apenas um argumento cujo tipo é o TDA onde o atributo está definido, e retorna o valor do atributo da instância de TDA passada. A segunda função tem dois argumentos, o primeiro é o tipo do TDA onde está definido o atributo e o segundo é do tipo do atributo. Esta segunda função substitui o valor do atributo na instância de TDA passada pelo valor do segundo argumento. Este nível de encapsulamento permite obter algum grau de independência lógica dos dados.

Para além das funções referidas no parágrafo anterior, existe uma terceira função implícita de um TDA de SQL3, a função *construtora por defeito*. Esta função pública é usada para criar novas instâncias do TDA. Tem como nome pré-definido o nome da classe, não possui argumentos e tem como valor de retorno uma nova instância do TDA em questão, com os atributos tendo como valores os definidos por defeito.

As restantes rotinas (procedimentos ou funções) definidas pelo utilizador, caracterizam o comportamento do TDA, sendo também encapsuladas dentro da definição do TDA.

Na indicação da condição de ordenamento EQUALS ONLY ou ORDER FULL, é também possível a definição de funções pelo utilizador para determinação da ordenação usada: nas definições de predicados e na cláusula ORDER BY; nos testes de igualdade de TDAs, nas cláusulas GROUP BY e HAVING; e nos predicados UNIQUE e DISTINCT.

A função passada na proposição EQUALS ONLY, retorna um valor verdadeiro se e só se os dois argumentos TDAs comparados forem iguais segundo o critério implementado. A função implícita - STATE, usada por defeito, vai comparar por igualdade, atributo a atributo, e retorna verdadeiro se todas as comparações forem verdadeiras.

A função que se segue a RELATIVE, aplicada a TDAs X e Y, retorna um valor Z que é menor do que zero (se X < Y), igual a zero (se X = Y) ou maior do que zero (se X > Y).

Um TDA pode ser especificado como tipo de dados de colunas em tabelas SQL, parâmetros em procedimentos e funções, atributos na definição de outros TDAs, variáveis em expressões SQL compostas e outros casos.

O modo como os TDAs podem ser armazenado de forma persistente na base de dados, é como valores de colunas de uma tabela. Assim, para se armazenar instâncias do TDA definido no exemplo anterior, teria de ser criada uma tabela:

```
CREATE TABLE pessoas
  ( empregado_info      tipo_empregado );
```

Não existe possibilidade de nomear instâncias individuais de um TDA e armazena-las de modo persistente na base de dados usando apenas os seus nomes. Não existe também nenhum local onde todas as instâncias de um dado TDA estejam presentes, funcionando como extensão do TDA. Para que tal aconteça o utilizador tem de criar esse local explicitamente. As instâncias têm de ser armazenadas numa ou mais tabelas como valores de colunas, para que seja possível aplicar-lhes operações.

A norma prevê suporte sintáctico para efectuar interrogações sobre atributos armazenados ou virtuais dos TDAs que pertencem a tabelas. Suponhamos que a seguinte tabela é definida usando um TDA *tipo_pessoa* que continha um atributo *nome* e outro *idade*:

```
CREATE TABLE empregados
  ( pessoa_info         tipo_pessoa,
    cônjuge             tipo_pessoa,
    ...
  );
```

Neste caso, uma interrogação para achar os nomes das pessoas mais velhas do que 35 anos seria:

```
SELECT nome(e.pessoa_info)
FROM empregados e
WHERE idade(e.pessoa_info) > 35;
```

Subtipos e Herança de TDAs

Um TDA pode ser definido como subtipo de um ou mais TDAs, ou seja, suporta herança simples e múltipla. Neste caso, o TDA é designado como o subtipo directo dos TDAs especificados a seguir à palavra UNDER, e estes são designados como seus supertipos directos.

```
CREATE ABSTRACT DATA TYPE estudante_tipo UNDER pessoa_tipo
...
```

Um tipo pode ter mais do que um subtipo e mais do que um supertipo. Uma instância de um subtipo é considerada instância de todos os seus supertipos. Uma instância de um subtipo pode ser usada em qualquer parte onde seja possível usar um dos seus supertipos.

Todas as instâncias de TDAs estão associadas com o seu tipo mais específico, que corresponde ao subtipo mais baixo atribuído à instância. A qualquer momento, uma instância deve ter exactamente um único tipo especifico. O tipo mais específico de uma instância não tem de ser obrigatoriamente um nó exterior de uma hierarquia de tipos.

Um subtipo pode definir funções como qualquer outro TDA. Um subtipo pode também definir operações que têm o mesmo nome que as operações definidas para outros tipos incluindo os seus supertipos, implementando desta maneira suporte para polimorfismo.

5.3.2 Tipo Linha e Tabelas

Há em SQL3 uma outra manifestação da filosofia de orientação por objectos, para além dos tipos de dados abstractos, que consiste nos tipos linha os quais definem objectos linha, que são essencialmente n-tuplos.

Os tipos linha, tal como os tipos de dados abstractos, também suportam herança. É possível, na definição de tipos linha, usar atributos que são também eles tipos linha.

A definição de um tipo linha consiste em:

- CREATE ROW TYPE <nome do tipo linha>
- Lista de supertipos desse tipo linha, com possibilidade de renomeação dos atributos herdados;
- Lista com os nomes e tipos dos atributos.

Exemplo 5.2: criação de um hierarquia de tipos linha

```
CREATE ROW PessoaTipo
(
nome CHAR(40),
idade INTEGER
);

CREATE ROW TipoDepartamento
(
nome CHAR(40),
descrição VARCHAR
);

CREATE ROW TYPE EmpregadoTipo UNDER PessoaTipo
(
departamento TipoDepartamento,
);
```

Declaração de Relações

Após a criação de tipos linhas, é possível usá-los na definição de relações do seguinte modo:

- CREATE TABLE <nome da tabela>

- OF <nome do tipo linha>
- Opcionalmente com valor de referência - WITH REF VALUE
- UNDER <lista de supertabelas da tabela criada>
- Lista opcional de colunas da tabela.

Para criação de uma tabela de empregados, poderia usar-se:

```
CREATE TABLE Empregado OF EmpregadoTipo
```

Pode-se ver uma tabela como uma espécie de extensão da classe correspondente ao tipo do seu n-tuplo. Contudo, um tipo linha pode ser usado para obter mais do que uma tabela. É também possível, como pode ser visto no exemplo acima, a inclusão de tipos linha como atributos de tipos linha.

Acesso a componentes de um tipo linha

Podendo ser a estrutura dos componentes em SQL3 mais complexa, é necessário um mecanismo para se conseguir chegar a toda a informação. Assim, para obter o nome de todos os empregados e os nomes dos respectivos departamentos, pode definir-se a seguinte interrogação:

```
SELECT Empregado.nome, Empregado.departamento..nome
FROM Empregado
WHERE Empregado.idade > 50
```

sendo usada uma notação de duplo ponto para indicar o acesso a um componente dentro de um tipo linha.

Referências

O conceito de identidade de objecto presente em linguagens orientadas por objectos é obtido em SQL3 através da noção de referência. Um componente de um tipo linha pode ter como seu tipo uma referência para outro tipo linha. Sendo T um tipo linha, então REF(T) é o tipo de uma referência para um n-tuplo do tipo T [35]. Se pensarmos no n-tuplo como um objecto, então a referência será o seu OID.

Para o exemplo dado anteriormente, poderíamos definir uma representação alternativa para o tipo linha EmpregadoTipo:

```
CREATE ROW TYPE EmpregadoTipo UNDER PessoaTipo
(
departamento REF(TipoDepartamento),
idade INTEGER
);
```

em que o componente departamento, deixa de ser um tipo linha e passa a ser uma referência para um tipo linha.

Para desreferenciação, usa-se um operador de desreferenciação semelhante ao da linguagem C e C++, ou seja, a seta para o lado direito $\rightarrow$. Assim, seja x uma referência para um n-tuplo t, e a um atributo de t, então $x \rightarrow a$ representa o valor do atributo a no n-tuplo t. Este operador pode substituir certas operações de junção nalgumas interrogações:

```
SELECT Empregado.nome
FROM Empregado
```

```
WHERE departamento→nome = 'Financeiro'
```

Contexto de Referências

Um atributo referência para um tipo linha tem a possibilidade de referir qualquer um dos n-tuplos do conjunto de todas as tabelas definidas com esse tipo linha. Isso pode ser custoso em termos de performance quando se executa uma pesquisa como a anterior, pois obriga a que sejam percorridas todas as tabelas definidas com o tipo linha em questão. Para minorar este problema é possível a definição de contextos, ou seja, indicar as tabelas abrangidas pela referência, do conjunto de tabelas possíveis. Estas últimas são as tabelas cujo tipo linha é compatível com o da referência. Define-se então:

SCOPE FOR <atributo referência> IS <lista das tabelas abrangidas pelo âmbito>

Identificadores de Objecto como Valores

Embora seja imediato associar indicadores únicos para objectos como valores internos gerados pelo sistema e não acessíveis através de linguagens de interrogação, em SQL3 isso pode ser contornado permitindo a referência explícita de identificadores únicos de objecto.

Na declaração de tipos linha, existe a possibilidade de inclusão de um atributo referência para tipos linha do mesmo tipo linha desse atributo. Se incluirmos na declaração do tipo linha ou de uma tabela a proposição:

VALUES FOR <atributo> ARE SYSTEM GENERATED

então os valores desse atributo serão referências para o mesmo n-tuplo no qual essas referências aparecem. Esse atributo pode pois funcionar simultaneamente como chave primária da relação e como identificador único para os seus n-tuplos.

5.3.3 Procedimentos

As rotinas escritas na linguagem definida na norma SQL3 permitem a especificação do comportamento de um TDA ou podem ser associadas a tabelas evitando o recurso a uma linguagem hospedeira.

É possível, assim, criar um comportamento mais complexo para o programa de aplicação através de uma simples chamada. A seguir são descritas algumas das instruções que SQL3 oferece para a criação de rotinas internas SQL são [33]:

- Instrução composta: permitindo que uma colecção de instruções seja agrupada num bloco. Define também um contexto local, no qual é possível a definição de variáveis, cursores e condições para lidar com excepções;
- Instrução *case*: para selecção de um caminho de execução baseada em critérios de escolha;
- Instrução *if*: para escolha de um de entre dois caminhos possíveis de execução com base no valor lógico de uma ou mais condições;
- Instruções de *loop*, *while* e *repeat*: todas elas permitem a execução repetida de um bloco de operações até que um critério de paragem seja verdadeiro. Esse critério pode ser avaliado no início do bloco de operações - *WHILE*, no fim do bloco de operações -

REPEAT, ou pode ser provocada uma interrupção durante a execução do ciclo de execução usando, por exemplo, uma situação de excepção - *LOOP*;

- Instrução *leave*: continua a execução de um trecho de código, abandonando o grupo de instruções com a etiqueta passada como argumento para a instrução *leave*. É usada para interromper a execução de um grupo de instruções com um determinado nome como por exemplo um *LOOP*;
- Instrução *for*: permite a execução de um bloco de operações para cada linha de uma tabela. As linhas da tabela são atravessadas usando um cursor cuja definição é feita dentro do ciclo *for*;
- Instrução de atribuição *SET*: permite atribuir um valor a uma variável ou parâmetro SQL ou uma variável ou parâmetro de uma linguagem hospedeira.

A norma considera também a possibilidade de definição de módulos usando `CREATE MODULE`, onde é possível o armazenamento persistente de procedimentos SQL. Para os procedimentos definidos em módulos persistentes, contidos dentro de esquemas, é possível a sua invocação a partir de qualquer instrução dentro desse esquema.

5.3.4 Recursão

Em SQL3, ao contrário das versões anteriores, é possível definir relações de um modo intencional usando um sistema de regras, e tendo como base relações que efectivamente estão armazenadas em tabelas, ou seja, a extensão da base de dados. Usando a instrução WITH, é possível obter uma ou mais tabelas temporárias que podem opcionalmente ser obtidas de forma recursiva.

A instrução WITH tem a forma:

- A palavra WITH
- Uma ou mais definições, em que estas são separadas por virgulas e cada uma delas tem: opcionalmente a definição de recursividade usando a palavra - RECURSIVE, o nome da relação a ser definida, a palavra AS, e a interrogação que define a relação;
- A interrogação que refere a qualquer das relações definidas anteriormente e forma o resultado final.

Utilizando WITH, a resolução do problema exposto no exemplo 2.7, e baseada em [59] poderia ser:

```
WITH
Pares AS SELECT Aeroporto_origem, Aeroporto_destino FROM VOO
RECURSIVE LIGAM(Aeroporto_origem, Aeroporto_destino) AS
  Pares
  UNION
  (SELECT Pares.Aeroporto_origem, LIGAM.Aeroporto_destino
   FROM Pares, LIGAM
   WHERE Pares.Aeroporto_destino = LIGAM.Aeroporto_origem )
SELECT * FROM LIGAM;
```

que implementa as duas regras definidas para obter os aeroportos passíveis de serem ligados por voos da tabela.

5.3.5 Rotinas

Uma rotina pode ser uma função (FUNCTION), retornando valores ou um procedimento (PROCEDURE) sem valor de retorno. Um parâmetro pode ser de entrada IN, saída OUT, ou ambos INOUT. Rotinas diferentes podem ter o mesmo nome, tal é designado por *sobrecarga*, podendo ser usadas para permitir que um subtipo redefina uma operação herdada de um supertipo. A lista de parâmetros dessas rotinas deve ser suficientemente diferente para distinguir qual das rotinas deve ser usada numa dada invocação. SQL3 implementa o que se designa por modelo orientado por objectos *generalizado*, em que os tipos de todos os argumentos de uma dada rotina são tomados em consideração para a determinação de qual a rotina a ser invocada. O facto de ser usado este modelo em vez do modelo orientado por objectos *clássico*, usado em C++ ou Smalltalk, em que apenas um tipo especificado na invocação é considerado para determinação da rotina a ser invocada, torna a implementação do modelo mais complexa do que a correspondente nas linguagens referidas. Uma *rotina não determinística* pode retornar diferentes resultados para múltiplas chamadas com os mesmos argumentos de entrada em todas elas.

As rotinas podem ser expressas em SQL ou ser externas. O corpo de uma rotina SQL é totalmente escrito em na linguagem definida pela norma. Para tal, foram especificadas novas declarações para tornar SQL uma linguagem computacionalmente completa, como por exemplo, estruturas de controlo.

No caso de uma rotina externa, o corpo desta é escrito numa linguagem de programação convencional. Se a função for externa, o corpo da rotina em SQL é uma referência externa na forma:

```
EXTERNAL NAME <nome da função externa>
LANGUAGE <nome da linguagem>
<atributo determinístico>
```

5.3.6 Ligações a Linguagens de Programação

O standard SQL3 define ligações para diversas linguagens de programação. Em SQL embutido, as instruções SQL são embutidas na linguagem de programação hospedeira. Esta pode ser: Ada, C, COBOL, Fortran, MUMPS, Pascal e PL/I [34]. Continua a pôr-se no entanto a questão da desadaptação de impedâncias entre as duas linguagens.

A passagem de estruturas complexas de SQL para variáveis da linguagem hospedeira é feita através de localizadores - LOCATOR, gerados por SQL e que mapeam um conjunto de octectos numa variável SQL. Estes permitem a referenciação de uma estrutura SQL como por exemplo um TDA por parte de uma variável externa de uma linguagem hospedeira.

5.3.7 Triggers e Asserções

Triggers são acções condicionadas por eventos. Um *trigger* é definido sobre um determinado evento como, por exemplo, a inserção, remoção ou actualização de uma linha numa tabela ou o fim de uma transacção. Associada a um *trigger* está uma condição que será verificada sempre que o evento definido ocorrer. Por último, se a condição se verificar é despoletada uma acção que está associada com o *trigger* que pode ser qualquer sequência de operações realizada sobre a base de dados.

Em SQL3 as declarações de triggers permitem algumas opções definidas pelos utilizadores. Entre essas, as principais são [59]:

- A acção do *trigger* pode ser executada antes, depois ou em vez do evento definido no *trigger*;
- A acção pode referir-se aos valores já actualizados dos n-tuplos que foram alterados pelo evento definido para o *trigger* ou aos valores existentes antes da alteração;
- Eventos de actualização podem ser restritos a uma coluna particular ou a um conjunto de colunas;
- O programador pode especificar se a acção a tomar lugar será efectuada: uma vez por n-tuplo modificado ou uma vez para todos os n-tuplos.

Asserções

Uma asserção em SQL3 permite implementar uma restrição de integridade através de uma proposição que não pode ser violada. Em SQL3 é incluída na asserção a lista de eventos que vão provocar a verificação da condição da asserção, ou seja, quando é que esta tem de ser verificada.

5.3.8 Construtores de Tipos Colecção

Uma colecção é um valor compósito com zero ou mais elementos do mesmo tipo de dados. Um tipo de dados colecção especifico, é obtido com o par <construtor da colecção> <tipo de dados dos elementos dessa colecção>. Os tipos de construtores existentes para colecções são:

- Conjunto - SET: onde não é permitido a existência de elementos repetidos;
- Multi-conjunto - MULTISET: é possível repetição de elementos;
- Lista - LIST: ao contrário dos dois anteriores, em listas é armazenada a ordem dos elementos;
- Vector - ARRAY: também mantêm informação sobre a ordem dos seus elementos. Um vector tem definido um tamanho máximo, pelo que o seu tamanho não pode exceder o tamanho máximo definido.

5.4 Linguagens de Interrogação

Embora no paradigma de orientação por objectos, o conceito de linguagem de interrogação não seja fundamental, este constitui um requisito importante de um SGBD. Em sistemas de gestão de bases de dados orientadas por objectos, o papel de uma linguagem de interrogação orientada por objectos centra-se essencialmente nas operações de pesquisa à base de dados. Isto porque operações de manipulação de objectos da base de dados são feitos, vulgarmente, recorrendo à linguagem hospedeira através da ligação desta para com a base de dados fornecida pelo SGBDOO.

Em [8] é definida uma linguagem formal (*object-oriented predicate calculus - oopc*) apenas de interrogação para bases de dados cujo modelo de dados seja o orientado por objectos. As

principais características desta linguagem são: oferecer dois tipos de igualdade, baseada em valores e em identidade; expressões de travessia usadas para formulação de junções implícitas quando na presença de caminhos de travessia definidos entre classes; predicados alternativos para expressar diferentes restrições em instâncias de diferentes subclasses numa dada hierarquia de classes; e uso de métodos nas expressões de interrogação, fornecendo esta última característica extensibilidade à linguagem de interrogação na medida em que permite a definição de novos operadores definidos pelos utilizadores para serem aplicados nas interrogações.

Embora a estrutura sintáctica básica entre as linguagens de interrogação definidas para modelos relacionais e as definidas para modelos orientados por objectos seja a mesma, no mesmo artigo são referidas algumas diferenças entre as duas devido à diferente natureza dos dois modelos em questão:

- O modelo relacional não suporta a noção de identidade. Não é possível aplicar operadores de comparação de identidade neste modelo. A identidade é simulada com recurso a chaves primárias que são comparadas entre si por valor;
- Enquanto as estruturas complexas são definidas em modelos orientados por objectos com recurso a propriedades multi-valor, em modelos relacionais estas são decompostas em várias relações, havendo a necessidade de efectuar junções para definir um predicado de selecção de uma classe;
- Os relacionamentos são expressos em modelos relacionais com recurso a várias relações. Enquanto uma linguagem de interrogação OO pode facilmente atravessar um relacionamento realizando uma navegação, em sistemas relacionais essa navegação implica o explicitar predicados de junção;
- Os métodos não são suportados em linguagens de interrogação relacionais, banindo a possibilidade de extensão da linguagem por definição de novas propriedades;
- Atributos derivados são obtidos através de métodos, expressos numa linguagem de programação, em sistemas orientados por objectos. No modelo relacional estes são obtidos usando o mecanismo de definição de vistas. As vistas são no entanto obtidas através de interrogações que têm um poder limitado relativamente a métodos que são implementados numa linguagem computacionalmente mais poderosa;
- A hierarquia de classes é uma noção não suportada pelo modelo relacional. O contexto de acesso de uma interrogação relacional limita-se à relação declarada na cláusula `FROM`, enquanto numa linguagem de interrogação objecto pode alargar-se a toda uma hierarquia de classes.

Embora a definição de vistas sobre a base de dados seja uma característica importante na medida em que permite diferentes interpretações do esquema da base de dados, estas não são contempladas em [8]. No entanto [39] descreve uma linguagem de interrogação objecto designada por OO-SQL que constitui uma extensão da expressão SELECT de SQL-92, e que contempla a possibilidade de elaborar vistas sobre a base de dados. Estas podem ser de dois tipos: vistas relacionais, possibilitando a ferramentas relacionais aceder à base de dados orientada por objectos ou vistas de objectos que podem conter atributos com valor do tipo colecção ou do tipo referência, podendo estas vistas ser utilizadas na linguagem na qual são codificadas as aplicações (C++) através de uma interface (*Call Level Interface - CLI*), oferecendo assim uma declaratividade nas

interrogações. OO-SQL, pode ser usada para interrogação da base de dados orientada por objectos de um modo interactivo ou através da aplicação, usando a interface *CLI*.

No ponto seguinte aborda-se a possibilidade de convergência entre OQL, a linguagem adoptada pela norma ODMG, e inspirada na linguagem de interrogação do SGBDOO O2 [4].

5.4.1 Convergência SQL3/OQL

Com o desenvolvimento de SQL3 com suporte para objectos e, paralelamente, da linguagem de interrogação OQL como parte integrante do standard ODMG, adivinhava-se a possibilidade de estabelecer ligações entre as duas linguagens permitindo que bases de dados em SQL pudessem trabalhar conjuntamente com sistemas orientados por objectos.

Surge então um grupo de trabalho, criado no âmbito do projecto de desenvolvimento de SQL3 pela ISO e pela ANSI, com o objectivo de estudar um denominador comum para as duas linguagens. Constituindo OQL uma linguagem compacta e simples devido à sua ortogonalidade e ao facto de ter sido desde o princípio desenhada como uma linguagem de interrogação para objectos, aparece como principal factor motivador do grupo a possibilidade de incorporação de algumas propriedades de OQL na linguagem SQL3, no contexto do estudo da convergência possível entre as duas linguagens [6].

O grupo define como objectivo tornar OQL a linguagem de interrogação de SQL3. Enquanto SQL3 engloba os componentes de definição, manipulação e interrogação de dados constituindo mesmo uma linguagem computacionalmente completa, a norma ODMG inclui uma linguagem de definição independente (ODL) além de OQL. A manipulação é efectuada directamente na linguagem hospedeira através de ligações definidas pela ODMG. A OQL inclui apenas a linguagem de interrogação, não contemplando os operadores INSERT, UPDATE e DELETE existentes na parte de manipulação de dados de SQL. No entanto OQL pode efectuar actualizações na base de dados indirectamente, através da invocação de métodos nas expressões de interrogação.

A primeira proposta [51] estende o domínio alvo das interrogações feitas em SQL3 de tabelas para colecções de objectos, permitindo também que uma interrogação em SQL3 retorne uma colecção de objectos. As interrogações SQL possuem a propriedade de fecho relacional ou seja têm como argumentos tabelas e retornam tabelas. Com esta proposta generalizam-se as interrogações de modo a serem fechadas sobre colecções. Sintacticamente introduz-se a palavra reservada ITEM a seguir a SELECT para especificar que uma interrogação deve devolver um tipo multi-conjunto, e DISTINCT ITEM para retornar um conjunto, aplicando-se as regras de OQL para determinação do tipo. Adopta-se também a introdução de FROM *x* IN *colecção*, para permitir a interrogação de uma colecção como em OQL.

A segunda proposta [52] permite a aplicação de funções de comparação quer a tabelas quer a colecções, definindo a sua semântica. Adiciona a palavra reservada ELEMENT para selecção de um elemento simples de uma colecção. O operador THE é usado também para extrair um campo de um elemento simples numa colecção com apenas um elemento.

Ambas as propostas foram consideradas pelo comité responsável pelo SQL3 da ANSI.

5.5 SGBDOO vs SGBDR

No inicio da década de setenta, E. F. Codd apresenta o modelo de dados relacional, introduz o conceito de independência física, e define a álgebra e o cálculo relacionais. Constrói assim um modelo simples assente em sólidas fundações matemáticas, sendo esses factores determinantes para o seu elevado sucesso nos anos que se lhe seguiram.

No entanto este modelo apresenta limitações no modo como captura propriedades estruturais de entidades do mundo real. Falha também pela ausência da construção de hierarquias de agregação para modelização de objectos complexos e de generalização para modelização de relacionamentos superclasse-subclasse. No modelo relacional, os dados que descrevem objectos complexos são espalhados por várias relações normalizadas e acedidos através de operações de junção que se podem revelar custosas em termos de desempenho.

No início dos anos oitenta, o paradigma orientado por objectos permite que sejam definidas construções estruturais mais ricas que o modelo relacional, assim como propriedades comportamentais de objectos especificadas ambas a um nível lógico, portanto independente da sua implementação física. Algumas características do paradigma orientado por objectos demonstram o seu potencial na modelização de dados e desenvolvimento de sistemas [60]. Entre estas temos: tipos de dados abstractos, herança, encapsulamento e polimorfismo.

Contudo, uma das críticas apontadas à tecnologia orientada por objectos aplicada à modelização de dados e bases de dados é a falta de um fundamento unificador, de um modelo formal universal com características consensuais. Sem isso, a tecnologia orientada por objectos não apresenta a simplicidade e robustez matemática do modelo relacional.

O suporte para Tipos de Dados Abstractos, juntamente com os mecanismos de herança, constitui um dos grandes trunfos do modelo por objectos sobre o modelo relacional. Na verdade, neste último, uma das abordagens comuns para a memorização de estruturas complexas definidas na linguagem de programação da aplicação é proceder à codificação de objectos complexos, como por exemplo um documento com os índices, capítulos, secções e outra informação respeitante ao documento, como um simples campo da base de dados designado por BLOB (*Binary Large Object*).

Relacionada com a questão anterior está a da desadaptação de impedâncias entre as linguagens de programação que trabalham um registo de cada vez, e a linguagem de interrogação cujos operandos e resultados são relações inteiras. SGBDOOs têm talvez aqui a sua maior vantagem sobre os sistemas relacionais, pois oferecem um serviço persistente à linguagem de um modo harmonioso e consistente com a linguagem de programação, constituindo a manipulação da informação na base de dados uma extensão da linguagem que não choca com a sua sintaxe nem semântica. Nos sistema relacionais, onde as operações de manipulação da base de dados são realizadas recorrendo à linguagem de interrogação e manipulação de dados, é sempre necessário a existência de dois ambientes de desenvolvimento distintos cuja interface constitui muitas vezes um gargalo. Enquanto por exemplo SQL é uma linguagem orientada às relações não suportando a noção de apontador ou estrutura complexa, uma linguagem como C não inclui no seu modelo de dados possibilidade de tratamento de conjuntos.

Outra questão importante é o modo como são efectuados os relacionamentos nos dois modelos. Enquanto as referências no modelo relacional se fazem recorrendo a chaves primárias, isto é, a valores de atributos, no modelo por objectos recorre-se ao conceito de OID. Em

relacionamentos binários muitos-para-muitos sem atributos, é necessária a definição de uma relação de associação no modelo relacional, enquanto no modelo por objectos tal é suportado por atributos que são colecções de referências entre as classes relacionadas. Além disso, no modelo por objectos uma relação do tipo muitos pode ser ordenada se for usada uma lista de OIDs para os objectos referenciados.

O princípio da integridade referencial é reforçado em ambos os modelos, quer recorrendo ao uso de chaves no modelo relacional, quer à noção de membros inversos com manutenções automáticas no modelo por objectos.

Um dos aspectos apontados ao modelo relacional é o custo das operações de junção entre relações comparado com o seguimento de apontadores do modelo por objectos. Estas operações de junção de relações são vulgarmente necessárias quando se tem um elevado número de relacionamentos na base de dados, como por exemplo em aplicações de engenharia, ou quando se procura representar uma estrutura complexa recorrendo ao seu "alisamento" em várias relações. Esta questão constitui um dos grandes factores motivadores para o uso de tecnologias orientadas por objectos em aplicações que exigem um elevado numero de operações de travessia.

A seguir continua-se a confrontação dos dois modelos em dois tópicos específicos. O assunto do primeiro tópico, desempenho, revela-se de fundamental importância em determinadas aplicações de bases de dados. O segundo tópico visa a Internet, dado haver uma ligação cada vez mais forte entre esta e bases de dados.

Desempenho

Na selecção da tecnologia de bases de dados adequada a uma aplicação, um dos factores importantes a considerar é o desempenho da base de dados.

Frequentemente, os utilizadores responsáveis pelo desenvolvimento de aplicações pretendem comparar o desempenho da sua aplicação em diferentes bases de dados para que possam seleccionar a que melhor se ajusta. Este constitui o problema de *medição de desempenho para uma aplicação* [58].

Em [12] é descrita uma técnica de medição de desempenho de bases de dados, adequada para operações relacionadas com aplicações de engenharia, designada por OO1 (*Object Operations version 1*). O objectivo desta técnica é a medição do desempenho de operações comuns que não possam ser expressas usando a linguagem SQL. Esta técnica de medição é aplicável quer a bases de dados relacionais quer a bases de dados orientadas por objectos podendo ainda aplicar-se a outros tipos de bases de dados como, hierárquicas e reticuladas.

O esquema considerado para a base de dados é composto por apenas dois registos lógicos:

```
Peça: RECORD[id:INT;tipo:STRING[10];x,y:INT;construção:DATE]
Conexão: RECORD[de:Peça-id;para:Peça-id;tipo:STRING[10];cmp:INT]
```

As operações consideradas na determinação da medida de desempenho são:

- Pesquisa aleatória de 1000 Peças;
- Travessia - selecção de uma *Peça* aleatoriamente, e pesquisa de todas as peças a ela ligadas até ao sétimo nível;
- Inserção de 100 *Peças* e de três *Conexões* para cada peça.

Estas operações podem ser efectuadas com várias configurações. Memória *cache* vazia (*cold*) ou cheia (*warm*), para pequenas (4 MB) ou grandes bases de dados (40 MB), e para bases de dados locais ou remotas.

O mesmo artigo apresenta resultados comparativos de medição de desempenho de um SGBD relacional e de um SGBDOO. Em bases de dados de pequenas dimensões, o SGBDOO apresentado consegue melhores resultados, principalmente em operações de travessia. O SGBDOO também tira melhor partido da memória *cache* cheia. Para bases de dados de grande dimensão, os desempenhos das duas bases de dados aproximam-se mais, havendo mesmo melhor desempenho em operações de pesquisa por parte do SGBD relacional.

O autor salienta o facto de a partir dos resultados não se poder concluir que um modelo é melhor do que o outro para aplicações de engenharia, podendo a diferença de magnitude no desempenho dever-se a questões de arquitectura em vez de questões relacionadas com o modelo.

Os fabricantes de bases de dados orientadas por objectos adoptam diferentes arquitecturas e soluções de implementação para os seus SGBDOOs. Esta diversidade provoca diferenças de desempenho entre sistemas. Em [10] é definida uma técnica de medição de desempenho especificamente desenvolvida para SGBDOOs designada por OO7. Esta técnica basicamente considera o mesmo tipo de operações que OO1. No entanto, utiliza um esquema bastante mais complexo, definindo módulos constituídos por objectos relacionados entre si de uma forma hierárquica. Este conjunto de testes permite fazer uma análise mais fina do desempenho relativo dos SGBDOOs. Não é contudo adequado para comparar estes com os sistemas relacionais.

Internet

A generalização do uso da Internet e a integração dos sistemas de informação internos das empresas - Intranet com a Internet - levaram a uma maior apetência por informação em múltiplos formatos, desde imagens fixas a audio e vídeo. O tratamento e armazenamento da informação num formato para utilização imediata na Internet sem necessidade de conversões, constitui também um factor no qual as bases de dados orientadas por objectos, dada a sua capacidade de armazenarem tipos de dados complexos directamente, estão melhor colocadas do que os sistemas relacionais. Estes últimos, embora possam armazenar informação que pode ser consultada de modo dinâmico na Internet, necessitam de uma ferramenta específica (muitas vezes um servidor de HTML fornecido pelo próprio fabricante do SGBD relacional) para fazer a ligação à base de dados.

Com a possibilidade de envio de aplicações na Internet, nomeadamente com o advento da linguagem Java, as bases de dados orientadas por objectos têm vindo a oferecer serviços de persistência para a informação na *Worl Wide Web*, directamente por aplicações enviadas para o cliente. O armazenamento de estruturas de dados Java directamente na base de dados e a possibilidade de se poder aceder a várias bases de dados, localizadas em diferentes pontos da Internet, pela mesma aplicação Java, sem necessidade de camadas intermédias, constituem factores importantes para a utilização desta solução.

5.6 Conclusões

A previsão de que a década de noventa iria constituir o marco de viragem em grande escala das bases de dados relacionais para as bases de dados orientadas por objectos não se verificou até agora.

Na realidade vários factores externos à trilogia: aplicações, tecnologias e standards, contribuem para a resiliência das bases de dados relacionais:

- Forças de mercado - os fabricantes de SGBDs relacionais, que naturalmente não estão interessados em perder cota de mercado, empenham-se em tentar adaptar os seus produtos aos novos desafios das bases de dados e publicitam as capacidades OO que as novas versões dos seus sistemas vão apresentando;
- Unanimidade em torno de modelo proposto por Codd - levou à concentração de esforços em torno de um denominador comum. O mesmo não acontece em SGBDOOs o que levou ao aparecimento de uma colecção de diferentes dialectos do modelo levando a uma maior dificuldade na adopção de standards;
- Forças de inércia - dada a delicadeza com que os sistemas de informação são tratados nas organizações, estas preferem uma solução evolucionária, cuja estratégia se baseie em conversão de software, a uma solução revolucionária, que exiga a abolição de investimentos passados em software.

No entanto, os SGBDOOs conquistaram nichos de mercado onde os sistemas relacionais não se ajustaram. É este o caso de sistemas de informação geográfica, de automatização de escritório, sistemas de telecomunicações, de CAD e outras aplicações de projecto em Engenharia. A explosão do uso da Internet e a popularidade da linguagem Java bem como o elevado ritmo de desenvolvimento tecnológico ao nível das telecomunicações e das normas de compressão de informação multimédia, tornaram frequente a circulação de informação multimédia na Internet, a qual cria expectativas de diversificação do uso dos SGBDOOs. Esta perspectiva do mercado permite a sustentação de uma aposta no cada vez maior desenvolvimento de SGBDOOs que estejam ao nível dos sistemas relacionais mesmo nos requisitos há muito impostos pelas aplicações tradicionais. Questões como segurança ou disponibilização de linguagens de interrogação declarativas começam a equivaler-se nas duas tecnologias.

6 Conclusões

A área dos Sistemas de Gestão de Bases de Dados, impulsionada pela evolução da trilogia: aplicações, tecnologias e normas, está em constante evolução. A magnitude do mercado dos SGBDs faz com que o rumo evolucionário seja inevitavelmente influenciado por este factor. Devido à cada vez mais importante fatia de aplicações que requerem algo que não encontram satisfatoriamente em produtos derivados do modelo relacional e do seu principal dialecto associado, o SQL, os principais fornecedores de SGBDs relacionais procuram incorporar as melhores componentes da tecnologia orientada por objectos nos seus sistemas. Paralelamente, os principais SGBDs orientados por objectos procuram incorporar nos seus produtos as melhores características dos sistemas relacionais, sem compromisso dos benefícios derivados do uso de classes, herança e encapsulamento.

O resultado é o aparecimento de sistemas de bases de dados com implementações que traduzem um casamento entre as duas tecnologias, assentes numa tecnologia híbrida que tenta reunir os pontos fortes de cada um dos modelos.

O trabalho exposto, teve como tema central a confrontação dos dois modelos, tendo havido a preocupação de, nos capítulos iniciais, tentar descrever separadamente cada um dos modelos, e posteriormente fazer uma súmula do estado da arte. A realização da aplicação de descrição arquivística dotou o autor de uma maior sensibilidade para o estudo em causa.

6.1 Resultados Obtidos e Desenvolvimento Futuro

Dadas as características do trabalho exposto, surge uma divisão natural na análise dos resultados. Pode-se estabelecer um primeiro conjunto de resultados directamente dependentes da implementação do sistema de descrição arquivística, e um segundo grupo de resultados adstritos ao estudo de natureza mais teórica paralelamente realizado.

Assim para o primeiro grupo temos:

- O Sistema de Gestão de Base de Dados orientado por objectos revelou-se suficientemente poderoso para a implementação da estrutura de natureza hierárquica do problema de descrição arquivística;
- O mesmo sistema revelou-se também adequado para lidar com o elevado número de entidades consagradas no modelo e a complexidade de relacionamentos de vários tipos existentes entre elas;

- A interface de programação da extensão Tk da linguagem Tcl, para realização da interface gráfica com o utilizador, revelou-se suficientemente poderosa, embora não se tenha explorado a capacidade de visualização multimédia;
- O desenvolvimento de uma aplicação de bases de dados orientada por objectos, implica o domínio de uma linguagem de programação genérica, vulgarmente orientada por objectos, o que não é necessariamente o caso nos sistemas relacionais. Esse factor torna mais lenta a curva de aprendizagem.

Na análise teórica, algumas das ilações importantes a retirar são a seguir expostas:

- SGBDs Orientados por Objectos começam a oferecer um nível de maturidade mais elevado tentando desse modo invadir áreas de aplicação tradicionalmente dedicadas aos sistemas relacionais;
- O surgimento de sistemas cada vez mais pequenos, de suportes de armazenamento de cada vez maior capacidade, está a despertar um maior interesse no armazenamento de informação em bases de dados no mais variados formatos;
- Áreas que tradicionalmente se socorriam de soluções à medida. utilizam cada vez mais SGBDs;
- A explosão do uso da Internet, o advento da linguagem orientada por objectos Java e a apetência por informação multimédia, obrigaram os SGBDs a considerarem esses aspectos e adaptarem-se;
- Os principais sistemas de bases de dados relacionais, procuram adaptar-se a uma nova realidade, associada com os três pontos anteriormente descritos, através da introdução de capacidades derivadas da tecnologia orientada por objectos. Surgem assim os SGBDs híbridos designados por SGBDRO - Sistemas de Gestão de Bases de Dados Objecto Relacional.

Relativamente a desenvolvimentos futuros que poderão ser efectuados com base na implementação da aplicação de descrição arquivística temos:

- Acompanhamento da evolução de normas relacionadas com o pacote de normas de descrição arquivística lançadas pela *ISAD(G)*;
- Melhoramento das capacidades de interrogação *ad hoc* para a base de dados;
- Elaboração de uma aplicação de pesquisa, eventualmente em Java, para a base de dados de descrição arquivística;
- Estudo da possibilidade de inclusão de novas classes que compartimentem informação em formato multimédia como audio, vídeo ou imagens fixas;
- Refinamento da interface de programação entre a aplicação Tcl/Tk e a base de dados e estudo da possibilidade de visualização de informação multimédia usando essa interface;
- Avaliação da aplicação em ambiente de utilização real, nomeadamente do seu maior ou menor grau de adequação e facilidade de utilização.

Seria ainda interessante a implementação da mesma aplicação usando um SGBD relacional. Tal validaria de um modo muito mais firme algumas das ilações tiradas neste trabalho.

6.2 Tendências Evolutivas

Os utilizadores de SGBDs procuram sistemas que:

- Ofereçam bons níveis de desempenho;
- Sejam de fácil utilização;
- Sejam extensíveis;
- Satisfaçam os requisitos das suas aplicações.

O futuro das bases de dados poderá não estar ligado a um modelo de dados específico, que sairá vitorioso sobre o outro. Antes poderá haver, como acontece presentemente, contributos de ambos os modelos para a elaboração de melhores SGBDs, não havendo domínio claro de nenhum deles.

Referências

[1] M. M. Astrahan et al., “System R: Relational Approach to database Management”. In “Readings in Database Systems - 2º edition”, Morgan Kaufmann, 1994.

[2] M. P. Atkinson et al., “An Orthogonally Persistent Java”, SIGMOD Record Vol. 25, Num. 4, Dezembro de 1996.

[3] M. Atkinson, F. Bancilhon et al., “The Object-Oriented Database System Manifesto”, In “Readings in Database Systems - 2º edition”, Morgan Kaufmann, 1994.

[4] F. Bancilhon, S. Cluet e C. Delobel, “A Query Language for the O2 Object-Oriented Database System”, Proc. do 2º workshop em Linguagens de Programação de Bases de Dados, Gleneden Beach, Junho de 1989.

[5] Francois Bancilhon, “Object Databases”, ACM Computing Surveys, Vol. 28, No. 1, Março de 1996.

[6] Francois Bancilhon e Amelia Carlson, “Providing Rich Query Funcionality”, ISO DBL LHR-078 e ANSI X3H2 95-462, Janeiro de 1996.

[7] Bob Beck e Steve Hartley, “Persistence Storage for a Workflow Tool Implemented in Smalltalk”, OOPSLA 94, Portland - USA, Outubro de 1994.

[8] Elisa Bertino et al., “Object-Oriented Query Languages: The Notions and the Issues”, IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 3, Junho de 1992.

[9] D. M. Capper, “C++ for Scientists, Engineers and Mathematicians”, Springer-Verlag 1994.

[10] Michael J. Carey, David J. DeWitt e Jeffrey F. Naughton, “The OO7 Benchmark”, University of Wisconsin.

[11] R. G. G. Cattell, “Object Data Management: object oriented and extended relational”, Addison Wesley, 1994.

[12] R. G. G. Cattell e J. Skeen, “Object Operations Benchmark”, ACM Transactions on Database Systems, Vol. 17, No. 1, Março de 1992.

[13] R. G. G. Cattell et. al, “The Object Database Standard: ODMG 2.0”, Morgan Kaufmann Publishers, Inc., 1997.

[14] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks”, CACM, Vol. 13, Num. 6, Junho de 1970.

[15] E. F. Codd, "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks", IBM Research Report RJ599, Agosto de 1969.

[16] O. J. Dahl, B. Myhrhaug, e K Nygaard, "The SIMULA 67 Common Base Language", Pub. S-22, Norwegian Computing Center, Oslo, Norway, 1970.

[17] Hugh Darwen e C. J. Date, "The Third Manifesto, version two", Working Draft de 11 de Janeiro de 1996.

[18] C. J. Date, "An Introdution to Database Systems (6th edition)", Addison Wesley, 1995.

[19] Judith R. Davis, "Extended Relational DBMSs: The Technology, Part 1" e "Universal Servers: The Players, Part 2", DBMS Online - dbmsmag.com, Junho e Julho de 1997.

[20] Claude Delobel et al., "Databases: From Relational to Object-Oriented Systems", International Thomson Publishing.

[21] R. Elmasri, S. B. Navathe, "Fundamentals of Database Systems", The Benjamin/Cummings Publishing Company, Inc., 1989.

[22] Fabrizio Ferrandina et al., "Schema Evolution in Object Databases: Measuring the Performance of Immediate and Deferred Updates".

[23] Ana Franqueira e outros, "ISAD(G) Normas Gerais Internacionais de Descrição em Arquivo - tradução portuguesa", cadernos BAD (2) 1995, p. 87-116.

[24] A. Goldberg, D. Robson, Smalltalk-80: The language and its Implementation, Addison-Wesley, Reading, Massachusetts, 1983.

[25] Laura M. Haas et al., "Starburst Mid-Flight: As the Dust Clears", IEEE Transactions on Knowledge and Data Engineering, Março de 1990.

[26] Eric H. Herrin, "Schema and Tuple Trees: An Intuitive Structure for Representing Relational Data", University of Kentucky, Abril de 1995.

[27] International Standards Organization, "Linguagem de Base de Dados SQL", Documento ISO-9075-1987(E)". Disponível também como American National Standards Institute, Documento X3.135-1986.

[28] International Standards Organization, "Linguagem de Base de Dados SQL com melhoramentos de integridade", Documento ISO-9075-1989(E)". Disponível também como American National Standards Institute, Documento X3.135-1989.

[29] International Standards Organization, "Linguagem de Base de Dados SQL", Documento ISO-9075-1992(E)". Disponível também como American National Standards Institute, Documento X3.135-1992.

[30] Hiroshi Ishikawa et al., "The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System", ACM Transactions on Database Systems, Vol. 18, No. 1, Março de 1993.

[31] ISO-ANSI Working Draft, "Database Language SQL - Part 1: Framework (SQL/Framework)", Abril de1997.

[32] ISO-ANSI Working Draft, “Database Language SQL - Part 2: Foundation (SQL/Foundation)”, Abril de1997.

[33] ISO-ANSI Working Draft, “Database Language SQL - Part 4: Persistent Stored Modules (SQL/PSM)”, Abril de1997.

[34] ISO-ANSI Working Draft, “Database Language SQL - Part 5: Host Language Bindings (SQL/Bindings)”, Abril de1997.

[35] ISO-ANSI Working Draft, “Database Language SQL - Part 8: Extended Objects (SQL/Object)”, Abril de1997.

[36] Takahiro Ito, “Database with LSI Failure Analysis Navigator”, IEICE Trans. Electron., Vol. E79-C, No. 3, Março de 1996.

[37] Arthur M. Keller et al., “Persistence Software: Bridging Object-Oriented Programming and Relational Databases”.

[38] BrianW. Kernighan e Dennis M. Ritchie, “The C Programming Language”, Prentice-Hall, New Jersey, 1988.

[39] Jerry Kiernan e Michael J. Carey, “Extending SQL-92 for OODB Access: Design and Implementation Experience”, OOPSLA 94, Portland - USA, Outubro de 1994.

[40] Franck Lebastard, “Is an object layer on a relational database more attractive than an object database?”, CERMICS/INRIA.

[41] David Maier e Stanley Zdonik, “Fundamentals of Object-Oriented Databases”, em Readings in Object-Oriented Database Systems, Morgan-Kaufmann, 1989.

[42] Frank Manola, “An Evaluation of Object-Oriented DBMS Developments”, Agosto de 1994, GTE Laboratories Incorporated.

[43] Jim Melton, Alan Simon, “Understanding the new SQL: a complete guide”, Morgan Kaufmann Publishers, Inc., 1993.

[44] Object Design, “ObjecStore Building Applications”, Release 4.0.2, Junho de 1995.

[45] Object Design, “ObjecStore C++ API Reference”, Release 4.0.2, Junho de 1995.

[46] Object Design, “ObjecStore C++ API User Guide”, Release 4.0.2, Junho de 1995.

[47] Object Management Group, “A Discussion of the Object Management Architecture”, Janeiro de 1997.

[48] Object Management Group, editor: Richard Soley, “Object Management Architecture Guide - Revision 3.0”, Junho de 1995.

[49] Tamiya Onodera, “Experience with representing C++ Program Information in an Object-Oriented Database”, OOPSLA 94, Portland - USA, Outubro de 1994.

[50] Avi Silberschatz, Michael Stonebraker, Jeff Ullman, “Database Systems: Achievements and Opportunities”.

[51] SQL3/ODMG merger group, “Convergence with ODMG collection queries”, ANSI X3H2 96-112, Fevereiro de 1996.

[52] SQL3/ODMG merger group, “Orthogonality of Collection Expressions”, ANSI X3H2 96-214, 1996.

[53] Michael Stonebraker, “Inclusion of New Types in Relational Data Base Systems”, Proceedings of the 1986 IEEE Data Engineering Conference, Fevereiro de 1986.

[54] Michael Stonebraker, Greg Kemnitz, “The Postgres Next-Generation Database Management System”.

[55] Bjarne Stroustrup, The C++ Programming Language, Addison Wesley, 1986.

[56] Nobutaka Suziki, “Specialization Constraints for a Complex Object Model Supporting Selective Inheritance” in IEICE Trans. Inf. & Syst., vol. E78-D, num. 11, Novembro de 1995.

[57] The Committee for Advanced DBMS Function, “Third-Generation Database System Manifesto”, Memorandum UCB/ERL M90/28, University of California, Berkeley, 1990.

[58] Ashutosh Tiwary et al., ”Evaluation of OO7 as a system and an application benchmark”.

[59] J. D. Ullman, J. Widon, “A First Course in Database Systems”, Prentice-Hall, Inc., 1997.

[60] Rainer Unland, Gunter Schlageter, “Object-Oriented Database Systems: Concepts and Perspectives”, Lecture Notes in Computer Science - Symposium IBM, Springer-Verlag, 1990.

[61] Gottfried Vossen, “On Formal Models for Object-Oreinted Databases”, OOPS messenger, vol. 6, num. 3, Julho de 1995.

[62] Brent Welch, “Pratical Programming in Tcl and Tk”, Prentice-Hall, Maio de 1995.

[63] Rahav Yairi, “A Gradual Integration of OODBMS to Legacy Telecommunications Systems”, Nokia Research Center, Finlândia.

Anexo A

Neste anexo é apresentado o modelo conceptual da aplicação de descrição arquivística. Este modelo foi desenvolvido em OMT, compreendendo apenas a parte estrutural. O essencial do modelo foi elaborado no âmbito de um projecto a decorrer no INESC - Porto, designado JNICT - Archivum. Na análise conceptual da aplicação desenvolvida, foram contemplados todos os elementos de descrição das normas ISAD. Estes surgem como atributos de objectos no modelo obtido.

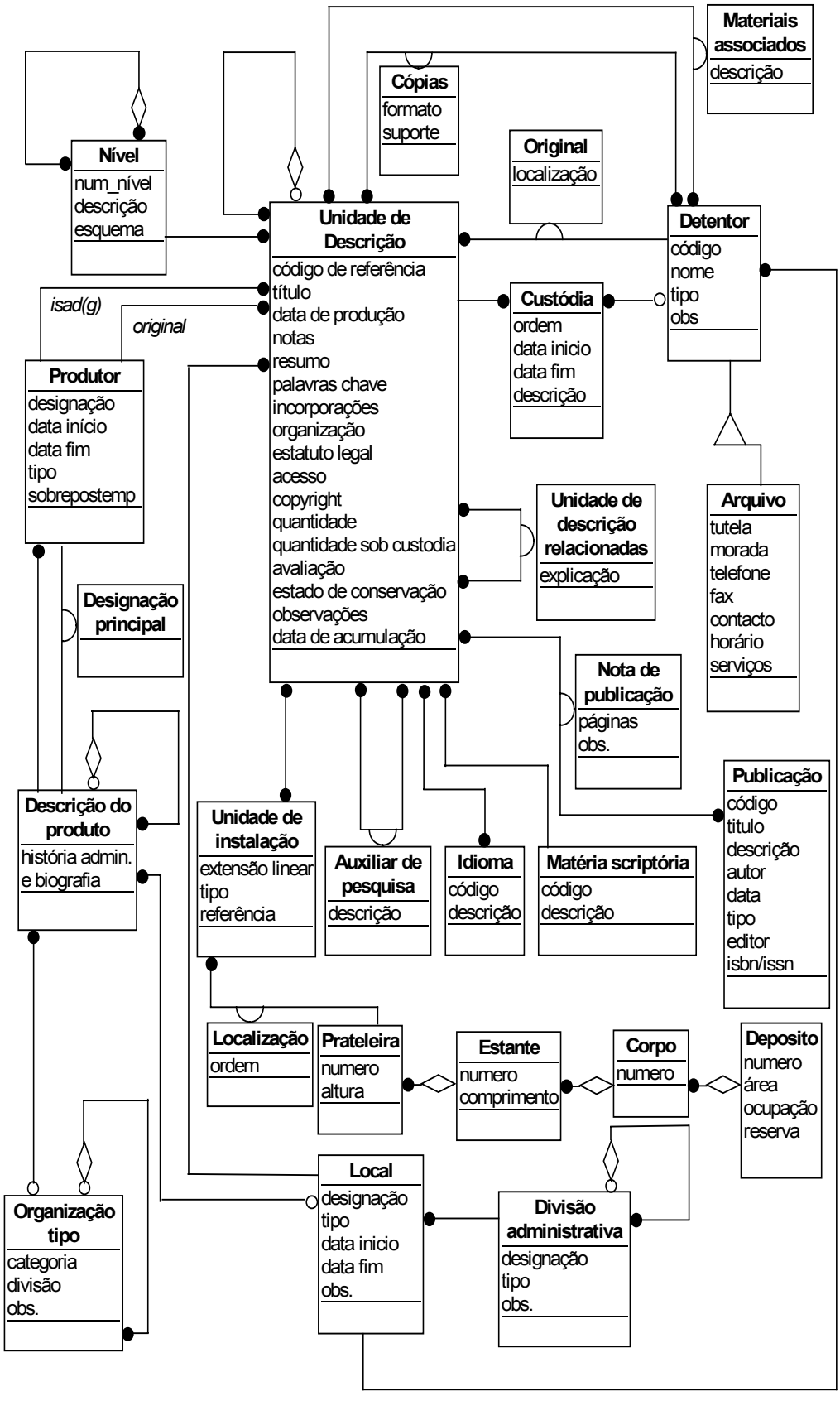


Figura A.1: Modelo conceptual da aplicação de descrição arquivística