EXTENSÕES NO SGBD COPPEREL

PARA APLICAÇÕES NÃO CONVENCIONAIS

Cláudio Newton Ferreira Trotta

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Jano Moreira de Souza, Ph.D.
(presidente)

Profa. Ana Maria de Carvalho Moura, Dr. Ing.

Profa. Ana Regina Cavalcanti da Rocha, D.Sc.

Prof. Rubens Nascimento Melo, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 1989

TROTTA, CLÁUDIO NEWTON FERREIRA
Extensões no SGBD COPPEREL para Aplicações Não Convencionais. [Rio de Janeiro] 1989.
X, 209 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1989)
Tese - Universidade Federal do Rio de Janeiro, COPPE.
1 - Banco de Dados Não Convencionais
I. COPPE/UFRJ II. Título (Série)

Para Janina e Julinha.

AGRADECIMENTOS

À Janina, pelo seu amor e paciência intermináveis.

Aos meus pais, pelo ambiente de apoio e liberdade que propiciaram.

À todos os meus irmãos e mestres, onde quer que estejam.

Ao Jano Moreira de Souza pela orientação e acolhida na sua linha de pesquisa.

À Marta Lima de Queirós Mattoso pelo suporte no COPPEREL e diversas idéias preciosas.

À Lígia Alves Barros e ao Cláudio D'Ipolitto, por fazerem tudo começar.

Ao Olavo Vieira de Farias por criar as condições para ser possivel terminar este trabalho, e, também, pelo incentivo determinado e silencioso.

Ao Marcelo Ribeiro Costa pela participação na implementação da interface.

Ao Leonardo Paiva de Araújo pela participação na implementação da interface e dos campos longos e, também, pela confecção das figuras.

Aos demais amigos e professores do Programa de Engenharia de Sistemas, em particular às professoras Ana Regina C. Rocha e Leila M. Rippol Eisirik.

Aos amigos da EC, especialmente ao David Naidin, Luis Henrique Pessanha, Giovanni Romanelli e Sergio Pintor, por cobrirem minhas diversas ausências.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Mestre em Ciências (M.Sc.)

EXTENSÕES NO SGBD COPPEREL
PARA APLICAÇÕES NÃO CONVENCIONAIS

Cláudio Newton Ferreira Trotta
Março, 1989

Orientador: Jano Moreira de Souza
Programa: Engenharia de Sistemas e Computação

Este trabalho discute algumas propostas, existentes na literatura, de SGBDs e modelos de dados mais adequados à aplicações não convencionais, buscando os requisitos de funcionalidade desejáveis para SGBDs voltados para aplicações desta categoria. A partir desse estudo, três caminhos complementares para a adaptação do SGBD COPPEREL são propostos: i) interface para linguagens hospedeiras, ii) campos longos e iii) objetos complexos. A interface para linguagens hospedeiras foi construída, bem como foram implementados os algoritmos para tratamento eficiente de campos longos. Os resultados deste trabalho estão servindo de base para outras pesquisas nas áreas de Banco de Dados e Engenharia de Software, mais notadamente, para a construção de SGBDs para suportar ambientes de desenvolvimento de software.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

EXTENDING THE DBMS COPPEREL
FOR NON CONVENTIONAL APPLICATIONS

Cláudio Newton Ferreira Trotta
March, 1989

Thesis Supervisor: Jano Moreira de Souza
Department: Computer Science and System Engineering

This work discuss some existing DBMS and data models for non-conventional applications, looking for the desirable functionality requirements for a DBMS of this sort. From this study, three complementary ways of extending the COPPEREL are proposed: i) an interface with host languages, ii) long fields and iii) complex objects. The interface has been implemented, and algorithms for the efficient handling of long fields have also been developed and implemented. The results of this work are already being used in further research in the fields of Database and Software Engineering, specially for building DBMS to support Software Development Environments.

ÍNDICE

Pág.

INDICE DE FIGURAS

CAPÍTULO I

INTRODUÇÃO

I.1) Motivação.

Cada vez mais, profissionais de diversas áreas estão utilizando-se do computador como forma de automatizar, aumentar a produtividade ou mesmo explorar novas possibilidades do seu ramo de atividade.

Dentro deste cenário, os Sistemas de Gerência de Banco de Dados (SGBDs) têm sido usados como a base de sistemas aplicativos onde o usuário pode armazenar, de forma centralizada, as informações que necessita. O uso de SGBDs traz uma série de facilidades, tais como: compartilhamento dos dados, controle centralizado, segurança, confiabilidade, integridade lógica, além de mecanismos para recuperar informações através de linguagens altamente expressivas.

À medida que novas possibilidades de aplicações do computador foram surgindo, em parte devido ao próprio desenvolvimento da informática, motivadas pelo avanço e barateamento do hardware (estações gráficas, estações de trabalho, redes), os novos usuários gostariam de se beneficiar das vantagens de usar SGBDs para suas novas aplicações, tais como : Projeto e Manufatura assistido por computador, Ambientes de Desenvolvimento de Software, Automação de Escritórios, Sistemas de Apoio à Decisão, Sistemas Especialistas, entre outros.

O problema que ocorre é que os SGBDs existentes foram construídos visando a atender os sistemas aplicativos mais tradicionais, conhecidos, também, como aplicações comerciais (folha de pagamento, contabilidade, entre outros) não sendo apropriados para as novas classes de aplicações emergentes.

A inadequação de SGBDs tradicionais às novas aplicações é, em grande parte, devido à "distância semântica" que

existe entre a realidade e a representação desta no SGBD, a qual é muito maior se comparada às aplicações comerciais. O mini-mundo, a respeito do qual se deseja armazenar informações, não é satisfatoriamente capturado pelo modelo no qual se fundamenta o SGBD em que será representado. A partir daí, a integridade lógica dos elementos armazenados é difícil de ser mantida; a linguagem de manipulação não é natural e o custo operacional (acesso, concorrência, recuperação de falhas) é muito grande.

Desta motivação, surge a necessidade de pesquisar novas soluções mais adaptadas às novas classes de aplicações. Estas têm sido conhecidas na literatura como Sistemas de Gerência de Bancos de Dados Não Convencionais. Os desafios ainda são provocantes. Atualmente, nenhum modelo de dados é amplamente aceito para representar a realidade de qualquer uma das novas classes de aplicação; nem tampouco existe uma solução definitiva para os problemas de controle operacional.

Muitos pesquisadores têm direcionado seus esforços na adaptação de um modelo já existente, tipicamente, o modelo relacional, ao invés de desenvolver SGBDs inteiramente novos. O Programa de Engenharia de Sistemas da COPPE/UFRJ desenvolveu o COPPEREL - um SGBD baseado no modelo relacional - que começou a operar em 1983. Posteriormente, suas funcionalidades foram aumentadas (MATTOSO 1987) mas ainda dentro das características do modelo relacional.

Este trabalho é um esforço inicial no sentido de adaptar o COPPEREL às aplicações não convencionais, mais notadamente na área de desenvolvimento de ambientes de software, servindo como laboratório para definição de um SGBD que apoie o Projeto TABA (SOUZA 1988a).

I.2) Objetivos do Trabalho.

O objetivo fundamental é, sem dúvida, estudar as adaptações ao COPPEREL para lidar com aplicações não convencionais. Para tal, e também para definir o escopo da dissertação, foram definidos quatro objetivos mais direcionados, a saber:

1) Discutir algumas propostas existentes de sistemas/modelos mais adequados à aplicações não convencionais, considerando desde aspectos de definição de modelos até aspectos de implementação, buscando-se os requisitos de funcionalidade necessários para SGBDs não convencionais;

2) Implementar uma interface entre linguagens hospedeiras e o COPPEREL para que novas aplicações possam imediatamente usar as capacidades relacionais já existentes;

3) Implementar os algoritmos para tratamento eficiente de uma sequência arbitrariamente longa de bytes. Estes algoritmos podem ser usados para suportar atributos do tipo longo no COPPEREL ou como método de acesso de um Sistema de Gerência de Objetos;

4) Definir uma proposta para incorporação de Objetos Complexos no COPPEREL que seja possível de ser adaptada ao modelo de implementação já existente.

I.3) Organização da Tese.

Para atender aos objetivos deste trabalho o restante da tese é dividido nos seguintes capítulos.

O capítulo II discute os diversos rumos de pesquisa que, em maior ou menor grau, podem fornecer subsídios para o entendimento do problema das aplicações não convencionais, a saber: Sistemas de Gerência de Bancos de Dados Orientados a Objetos, Modelos de Dados Semânticos, Extensões em SGBDs

Relacionais e SGBDs Extensíveis. Na sequência, algumas propostas são analisadas buscando similaridades que possam levar aos requisitos de funcionalidade de SGBDs Não Convencionais. Os aspectos de controles operacionais necessários são, também, discutidos.

O capítulo III apresenta os requisitos de funcionalidade para SGBDs Não Convencionais e descreve as alterações propostas no COPPEREL.

O capítulo IV trata da implementação da interface entre o COPPEREL e linguagens hospedeiras.

O capítulo V descreve proposta para a incorporação de campos longos no COPPEREL bem como a implementação dos algoritmos e estruturas de dados desenvolvidos, acompanhados de exemplos.

O capítulo VI define a proposta para Objetos Complexos no COPPEREL, indicando soluções para a implementação.

Finalmente, o capítulo VII apresenta as conclusões deste trabalho.

CAPÍTULO II
DO MODELO À IMPLEMENTAÇÃO

II.1) As Frentes de Pesquisa.

Nesta seção, analisamos diferentes linhas de pesquisa que, de uma forma ou de outra, contribuem para o esclarecimento do problema do uso de sistemas de gerência de banco de dados em aplicações não convencionais. O estudo focaliza, principalmente, os requisitos necessários à área de engenharia em geral, mas as conclusões podem ser úteis às áreas específicas de CAD/CAM, engenharia de software e automação de escritório. O casamento das pesquisas de banco de dados com aquelas da área de inteligência artificial não é coberto nesta oportunidade.

Basicamente, o que se procura é estabelecer similaridades entre linhas de pesquisa aparentemente diferentes. Os propósitos, origens e conceitos de cada linha são muitas vezes divergentes mas, sob o ponto de vista de requisitos necessários para modelar e representar "entidades" encontradas na área de engenharia, revelam-se uma quantidade de conceitos que são fundamentalmente os mesmos.

As linhas de pesquisa mencionadas acima são:

a) Extensões realizadas em SGBD's que implementam o Modelo Relacional;

b) Modelos de Dados Semânticos;

c) O paradigma de Orientação a Objetos, oriundo da área de linguagens de programação;

d) Sistemas de Gerência de Banco de Dados Extensíveis.

Estas propostas trazem, implicitamente, uma filosofia de modelagem de dados que fornece subsídios para permitir

capturar e mapear a realidade através do uso, em maior ou menor grau, dos três aspectos abaixo:

i) Um aspecto estrutural que descreve uma moldura onde as entidades e seus relacionamentos devem ser representados;

ii) Um aspecto comportamental que, usualmente, descreve as operações aplicáveis a uma entidade;

iii) Regras que uma entidade deve obedecer para participar do modelo. Estas regras podem vir implicitamente impostas no aspecto estrutural e/ou comportamental, ou podem ser definidas explicitamente.

Sob uma outra ótica, existe o aspecto operacional que consiste em materializar os controles necessários para que um modelo seja plenamente suportado por um sistema de gerência de banco de dados completo, a saber:

- Controle de transações e de acesso concorrente;

- Armazenamento e métodos de acesso eficientes;

- Controle de versões e evolução do esquema.

Estes aspectos são rapidamente abordados na seção 2.3.

A seguir, descrevemos, brevemente, algumas características das quatro linhas de pesquisa citadas acima. Alguns dos sistemas referenciados serão estudados mais detalhadamente na seção 2.2, na qual buscaremos fugir de uma classificação rigorosa em relação às quatro linhas citadas. A razão disto, é porque alguns destes sistemas podem se encontrar em alguma fronteira. Por exemplo: o sistema POSTGRES (STONEBREAKER 1986) é uma evolução do sistema de gerência de banco de dados relacional INGRES, mas as

alterações têm sido tão profundas que ele pode ser considerado um SGBD orientado a objetos.

II.1.1) Sistemas de Gerência de Banco de Dados Orientados a Objetos.

O paradigma de "Orientação a Objetos", cuja implementação mais representativa é o SMALLTALK (GOLDBERG 1983), vem tendo cada vez mais aceitação para a definição e implementação de sistemas de computação. Ele é particularmente atraente em situações onde a flexibilidade à mudanças é importante, tais como sistemas de automação de escritório, ou quando se deseja manter a produtividade no desenvolvimento, através da re-utilização de módulos (COX 1987). O paradigma utiliza alguns conceitos, tais como o de classes, já existentes na linguagem SIMULA (DAHL 1966).

Sistemas de Gerência de Banco de Dados Orientados a Objetos (SGBDOO's) são tentativas de tornar a memória persistente e prover controle operacional para o paradigma de orientação de objetos.

Há muito foi reconhecida a adaptabilidade do paradigma de Orientação a Objetos às aplicações de banco de dados ditas não convencionais. Uma série de protótipos de SGBDOO's têm aparecido na literatura nos últimos anos: GEMSTONE (MAYER 1986), SEMBASE (KING 1986), VBASE (ANDREWS 1987), IRIS (FISHMAN 1986), PROBE (MANOLA 1986), CACTIS (HUDSON 1986), KNOS (TSICHRITZIS 1987), ORION (BANERJEE 1987a, b e c), entre outros.

A primeira vantagem dos SGBDOO's em relação aos SGBD's relacionais, por exemplo, é a possibilidade de representar um objeto do mundo real diretamente. Um banco de dados pode ser visto como uma coleção de objetos abstratos, ao invés de um conjunto de tabelas normalizadas. A segunda, é que um objeto é definido por sua estrutura, que pode ser qualquer, e pelas operações válidas sobre ele, garantindo a definição do seu comportamento. Finalmente, através de mecanismos de

generalização/especialização, a modelagem de algumas situações se torna mais natural.

Normalmente, objetos similares são agrupados em classes. Uma classe define a estrutura do objeto, as operações válidas sobre este e a implementação destas. Um procedimento que implementa uma operação é conhecido como método. Um objeto pertence (na maioria das propostas) a somente uma classe. Um objeto de uma classe é conhecido como uma instância da classe. Podemos imaginar que, ao ser criado um objeto, a definição da sua estrutura, existente na classe, é materializada (instanciada) em um novo objeto.

Toda a dinâmica de um sistema orientado a objetos acontece a partir da troca de mensagens. Um objeto envia uma mensagem a outro objeto. Este envio pode ser visto como uma forma indireta de chamada a procedimento. Um seletor na mensagem indica qual operação (ou método) a ser ativado. O objeto responde à mensagem ativando o método, que pode mudar o seu estado (alterando os valores da estrutura) e/ou enviar outras mensagens a outros objetos. Estes mecanismos garantem uma grande abstração de dados; de fato, trata-se da aplicação do conceito de tipos abstratos de dados (TAD's).

Se uma classe de objetos é manipulada seguindo-se estritamente um protocolo, isto é, um conjunto de mensagens relacionadas, é possível que classes diferentes respondam a protocolos iguais. Permitindo que um programa trate da mesma forma objetos de diferentes classes, alcança-se modularidade e reusabilidade.

Finalmente, a idéia de especialização/generalização está presente em qualquer linguagem ou SGBD que se diga orientado a objetos. Especializar uma classe significa criar uma outra classe - conhecida como subclasse da superclasse sendo especializada - cuja definição é quase a mesma da classe mãe. A subclasse herda toda estrutura e métodos definidos para a superclasse. Algumas diferenças entre subclasses e superclasses podem ser introduzidas, definindo-

se na subclasse novas variáveis (que compõem a estrutura) e novos métodos. Se for definida uma variável ou método com o mesmo nome de uma variável ou método já existentes na superclasse, vale a definição feita na subclasse, sobrepondo-se à definição original.

Uma hierarquia de classes e subclasses pode ser definida, com o mecanismo de herança valendo entre "gerações" (ou seja, um "neto" herda atributos/métodos de seu "pai", seu "avô" e assim por diante).

Alguns sistemas permitem implementar mais de uma hierarquia estrita, isto é, uma subclasse pode ter duas ou mais superclasses. Este modelo, conhecido como herança múltipla, torna necessário desenvolver um mecanismo para resolver os conflitos de nomes (por exemplo: se duas superclasses de uma subclasse tiverem o mesmo método, qual deles será herdado ?).

Se uma superclasse A tem duas subclasses B e C, várias são as restrições possíveis de se impor e várias são as soluções propostas (vide seção 2.2). Tipicamente, é possível, entre outros:

- obrigar ou não, a cobertura da superclasse pelas subclasses (isto é, todas as instâncias de A são instâncias de B U C);
- permitir que a interseção de B e C seja vazia ou não;
- fazer com que uma subclasse seja subconjunto de outra;
- criar subclasses automáticas (a partir do valor de um atributo na superclasse).

Este tipo de relacionamento entre classes, também conhecido como "is-a", está presente na maioria dos modelos semânticos de dados. O relacionamento entre SGBDOO's e modelos semânticos é evidenciado em alguns protótipos: SEMBASE é baseado no Event Model (KING 1984), IRIS é baseado no modelo semântico TAXIS (MYLOPOULOS 1980) e no modelo

semântico funcional DAPLEX (SHIPMAN 1981), no qual também PROBE é baseado.

Existem ainda alguns conceitos, não tão amplamente de consenso, que ocorrem em SGBDOO's. Por exemplo, algumas propostas, como KNO, permitem que o objeto adquira novas operações dinamicamente; outras, se preocupam com mecanismos para permitir grande flexibilidade na evolução do esquema, como no ORION.

Provavelmente, as grandes contribuições do paradigma da orientação a objetos para SGBD's não convencionais sejam os mecanismos de generalização/especialização e o uso da noção de tipos abstratos de dados. Não esquecendo estes dois pontos, na seção 2.2, analisamos o SGBDOO ORION (BANERJEE 1987a) e a linguagem LOOPS (STEFIK e BOBROW 1985), sob o ponto de vista do que as propostas oferecem de diferente em relação ao paradigma de orientação a objetos, a saber: a possibilidade de tratamento de objetos complexos.

II.1.2) Modelos de Dados Semânticos.

Os modelos de dados semânticos (ABRIAL 1974, CHEN 1976, CODD 1979, SHIPMAN 1981, HAMMER 1981, MYLOPOULOS 1980, SU 1983, entre outros) apareceram inicialmente como ferramentas para auxiliar a definição de esquemas. Mais informações sobre o significado dos dados podiam ser representadas, auxiliando em muito o próprio processo de entendimento do problema. Depois disso, o modelo levantado podia ser "podado", sendo traduzido para um modelo suportado por um SGBD, principalmente, o relacional.

A vantagem de utilizar um modelo semântico de dados é a disponibilidade de mecanismos de abstração apropriados que permitem total independência do modelo de implementação, concentrando a atenção somente no nível conceitual. Estes mecanismos não sofrem de "sobrecarga semântica", isto é, normalmente uma construção em um destes modelos tem um significado bem definido, contrastando com o relacional, que

somente com os elementos atributo e relação, tem que modelar relacionamentos, especialização/generalização, associações, agregações, e outros conceitos. Além disto, é possível trabalhar, mesmo no nível conceitual, em diversos sub-níveis de abstração, onde em cada um são escondidos detalhes do sub-nível anterior.

A interligação de modelos de dados semânticos e o paradigma de orientação a objetos é interessante. Inicialmente, a maioria de modelos de dados semânticos que surgiam focalizavam, principalmente, o aspecto estrutural das entidades e dos relacionamentos sendo modelados. Tipicamente, as restrições de integridade suportadas eram aquelas implicitamente impostas pela própria estrutura que podia ser representada.

Modelos como ACM/PCM (BRODIE 1982), NIAM (VERHEIJEN 1982) entre outros, também chamados de modelos semânticos de "segunda geração" já permitiam a definição explícita de restrições de integridade bastante complexas. Outros modelos semânticos de segunda geração como: CIM (GUSTAFSSON 1982) e IML (RICHTER 1982) já incorporavam, no começo da década de 80, a parte comportamental, porém com sérios problemas em três aspectos: i) modelagem do tempo, que alguns consideravam implicitamente (CIM) e outros modelavam por meio de atributos explícitos, ii) dificuldades com a implementação, isto é, com o mapeamento do modelo semântico para um modelo suportado por um SGBD específico e iii) dificuldades com a modelagem, uma vez que estes modelos não proviam uma capacidade de modularização ou abstração de partes do esquema sendo construídos, apesar de alguns trabalhos abordarem este problema como a integração de vistas (EL MASHI, NAVATHE 1986)

À medida que os modelos semânticos foram incorporando aspectos comportamentais, SGBDOO's apareceram baseados nestas propostas (vide ítem 2.1.1).

Segundo (HULL & KING 1987), os componentes primários dos modelos semânticos são:

a) representação explícita de objetos, atributos de objetos, e relacionamentos entre eles;

b) construtores de tipos para criar tipos complexos ;

c) relacionamentos "is-a";

d) capacidade de representar informação derivada.

Comparando-se com as características de SGBDOO's, vemos coincidências nos diversos ítens, com as seguintes ressalvas:

a) Modelos orientados a objetos também se preocupam em representar explicitamente objetos e seus atributos. Relacionamentos têm que ser simulados via atributos, mas não são reconhecidos pelo modelo. Muitos modelos semânticos (tipicamente os funcionais) usam, também, atributos para representar relacionamentos;

b) Este é, provavelmente, o ponto de maior afastamento. O paradigma da orientação a objetos não incorpora os construtores de tipos dos modelos semânticos. Em SGBDOO's, sob o aspecto estrutural, podemos definir apenas atributos simples ou multivalorados. O sistema ORION e a linguagem LOOPS, analisados na seção 2.2, são raras exceções;

c) A diferença principal é que no paradigma de orientação a objetos, além da herança de atributos, ocorre, também, a de comportamento;

d) Esta é uma preocupação oriunda da área de modelos semânticos. O SDM (HAMMER 1982) aposta bastante nesta capacidade. Do lado dos SGBDOO's, o sistema CACTIS (HUDSON 1986) apresenta um modelo capaz de tratar com generalidade o problema de dados derivados, mantendo as informações atualizadas de forma automática e eficiente.

Os modelos semânticos IFO e SAM*, apresentados na seção 2.2, são bons exemplos do espírito desta linha. O modelo SAM* foi estendido para modelar problemas na área de CAD/CAM e CIM, ganhando, assim, um construtor para modelar versões de objetos.

II.1.3) Extensões em SGBD's Relacionais.

O modelo relacional, pela sua simplicidade, embasamento teórico e experiência prática adquirida, é um forte candidato para sofrer modificações que o tornem mais adequado à nova classe de problemas. Sua principal dificuldade para lidar com aplicações não convencionais é o fato de ser orientado a registro. Isto é muito bom para as aplicações comerciais tradicionais, mas não o é, quando a entidade a ser armazenada não pode ser representada diretamente por um registro. A solução encontrada, para representar estas entidades, consiste em "esquartejar" a entidade em diversas relações. Por isto, uma entidade deste tipo é conhecida como objeto complexo. Devíamos chamar, de fato, de objeto "natural", pois é como o usuário percebe a entidade. A "complexidade" é decorrente da inabilidade dos modelos existentes em representá-lo.

Como um objeto complexo encontra-se em diversas relações, as dificuldades surgem daí. O sistema vê um conjunto de relações, mas não reconhece nelas um objeto; a integridade do objeto é difícil de ser mantida; a linguagem de manipulação não é natural; e o custo operacional (recuperação, concorrência e acesso) é muito grande.

As razões para se estender o modelo relacional, ao invés de se partir para um modelo novo, são, segundo (STONEBREAKER 1986):

- falta de consenso sobre os requisitos do novo modelo;
- simplicidade do modelo relacional;
- necessidade de compatibilizar aplicações não convencionais e convencionais.

De fato, alguns sistemas orientados a objetos se apoiam em uma base relacional. O sistema IRIS é montado em cima do Relational Storage Subsystem; (ROWE 1986) propõe a implementação de uma hierarquia de objetos compartilhados, que incorpora conceitos de orientação a objetos, a partir do sistema POSTGRES, que é um banco relacional estendido.

Uma das propostas mais tradicionais é a extensão do System R (HASKIN 1981, LORIE 1982, LORIE 1984, DITTRICH 1985) para aplicação na área de engenharia (vide ítem 2.2.1), onde é possível declarar como um objeto único, para efeito de manipulação, uma hierarquia estrita de relações. A linguagem SQL foi estendida, e foram estudados diversos aspectos necessários aos novos requisitos de controles operacionais. Outra extensão incluída foi a criação de um novo tipo de dado, conhecido como campo longo, onde é possível armazenar, recuperar e modificar uma sequência arbitrariamente longa, e não interpretada, de bytes.

Em trabalho posterior, (BEVER 1988) reconhece a necessidade de incorporar mecanismos de generalização/especialização para atender aplicações de automação de escritórios. A partir das facilidades para composição de objetos, já existentes no System R, são definidas sequências de operações em XSQL para lidar com o conceito de tipo e subtipo. O trabalho não descreve como encaixar as novas operações no sistema, mas sem dúvida, um mecanismo de TAD's permitiria o uso transparente da nova facilidade.

(STONEBREAKER 1986 e 1987) mostra a possibilidade de modificar, profundamente, as funcionalidades de um SGBD relacional, a partir de pequenas modificações no modelo. O sistema POSTGRES (vide ítem 2.2.8) implementa o conceito de TAD's em cima de atributos e relações. É possível simular objetos complexos com bastante generalidade e, também, uma hierarquia de classes com herança de atributos e operações.

II.1.4) SGBD's Extensíveis.

SGBD's extensíveis são aqueles que permitem a incorporação de novas capacidades, com o intuito de se adaptar a uma determinada classe de aplicação. Sob um certo aspecto, SGBD's que incorporam facilidades para TAD's, como POSTGRES e PROBE, podem ser vistos como sistemas desta categoria. Entendemos, porém, como sistemas extensíveis aqueles que:

a) são compostos de um pequeno núcleo, a partir do qual um SGBD completo é gerado;

b) constituem um conjunto de ferramentas apropriadas para gerar semi-automaticamente um SGBD inteiro.

Os sistemas extensíveis devem possuir generalidades tanto quanto a sua capacidade de definir novos tipos e relacionamentos entre eles (passando pela modelagem no nível lógico), como, também, quanto ao nível de implementação. Gerência de armazenamento, métodos de acesso, gerência de bloqueios, gerência de "buffers", estratégia de recuperação, e otimizadores de consultas são componentes que devem ter sua arquitetura aberta e facilmente modificável.

Para projetar um sistema extensível, é necessário então, buscar as idéias mais fundamentais entre a diversidade de conceitos e soluções existentes em banco de dados para aplicações não convencionais, o que vai de encontro aos objetivos deste trabalho.

O Data Model Compiler - DMC (MARYANSKI et al 1986) tem como função gerar, a partir de uma especificação de um modelo de dados, provida através de uma interface gráfica, alguns componentes de um SGBD, a saber:

a) uma ferramenta automatizada de projeto de esquemas no modelo de dados em questão;

b) um tradutor que transforme a LMD do modelo de dados em uma linguagem compatível com a representação física;

c) uma linguagem de consulta interativa.

O projeto do DMC entendeu que, para construir tal ferramenta, seria necessário ganhar experiência com uma série de modelos de dados orientados a objetos. A partir de um modelo básico, baseado nos modelos de dados semânticos TAXIS (MYLOPOULOS 1980) e Structural Data Model (EL-MASRI 1980) e do desenvolvimento de modelos orientados a objetos nas áreas de: aplicacões comerciais, CAD para engenharia mecânica, automação de escritório e modelagem de desempenho de software, foram encontrados os seguintes requisitos de modelagem que o DMC deve suportar:

a) relacionamentos "is-a";

b) associações genéricas representando relacionamentos N:M com suas próprias operações, restrições e propriedades;

c) associações de referências (um objeto aponta outro);

d) associações de embutimento (um objeto aponta um conjunto de outros objetos);

e) relacionamentos de composição;

f) campos longos;

g) versões;

No DMC, a definição de um relacionamento, seja de que tipo for ("is-a","part-of" e outros), pode ser feita, diretamente, através de parâmetros. Assim foi definido um conjunto de parâmetros que deve ser suficiente para representar a semântica de qualquer relacionamento. São eles:

a) Cardinalidade - indica o número de instâncias de cada entidade que pode participar do relacionamento (1:1, 1:N , N:M);

b) Herança - indica se o relacionamento envolve mecanismos de herança de tipo e/ou valores (útil para implementar relacionamentos "is-a");

c) Restrição de integridade de inserção - indica se para incluir uma instância em uma entidade, deve existir uma outra intância, na outra entidade que participa do relacionameto (útil para implementar relacionamento "part-of");

d) Restrição de integridade de eliminação - indica se há eliminação em cascata, ao se eliminar uma instância que participe do relacionamento;

e) Grau - indica quantas entidades podem participar do relacionamento (relacionamentos binários, n-ários);

f) Forma de representação interna - indica se o relacionameto é representado por atributos, por uma entidade extra ou por um construtor especial;

g) Propriedades - indica se o relacionamento pode ter suas próprias propriedades como: atributos, operações e restrições de integridade, e ainda, se pode participar de outros relacionamentos;

O sistema EXODUS (CAREY 1986a e b) tem como objetivo funcionar como uma caixa de ferramentas para a geração de SGBD's. Para tal apresenta:

a) um gerente de armazenamento de objetos;

b) uma linguagem de programação adequada para construir SGBD's;

c) um gerente de tipos generalizado;

d) uma biblioteca de métodos de acesso;

e) protocolos de gerência de bloqueios e recuperação;

f) um gerador de otimizadores de consultas baseado em regras;

g) ferramentas para construir interfaces.

O gerente de armazenamento de objetos permite uma abstração idêntica às propostas de campos longos, isto é, um objeto, para o EXODUS, é uma sequência arbitrariamente longa e não interpretada de bytes. Neste sistema, a implementação é mais eficiente que a do System R pois é baseada numa estrutura semelhante a uma árvore B+. Além disso, no System R, um campo longo é apenas um novo tipo de dado que um atributo pode assumir; no EXODUS, um campo longo é a unidade básica de armazenamento sobre a qual pode ser construída qualquer abstração de estrutura. O gerente de armazenamento de objetos também provê suporte para o controle de versões de objetos. A implementação de campos longos é abordada com detalhes no capítulo 3.

O gerente de tipos fornece facilidades para o implementador de um SGBD criar hierarquias de classes com herança múltipla, como em alguns sistemas orientados a objetos. O sistema fornece ainda tipos básicos (inteiro, real, caracter e identificador de objeto), construtores de tipos (registro, array e conjunto) e facilidades de TAD's para criar novos tipos e operações sobre estes. A existência do tipo básico "identificador de objeto" e do construtor de conjunto fornece suporte para implementar objetos complexos.

A extensibilidade do sistema Starburst (SCHWARZ et al 1986) focalizou o armazenamento de dados externos ao SGBD, gerência de armazenamento, métodos de acessos extensíveis, tipos abstratos de dados e objetos complexos.

Novamente, vemos, em uma proposta, um mecanismo de TAD's como indispensável para se obter extensibilidade. A implementação de objetos complexos pode ser feita como no System R, ou ainda, usando-se campos longos mais a flexibilidade de TAD's para implementá-los. Neste caso, porém, compartilhar sub-objetos é mais difícil.

II.2) Discussão de Algumas Propostas Existentes.

Nesta seção, analisamos algumas propostas existentes, que podem ser classificadas, de uma forma não rigorosa, de acordo com as linhas de pesquisa mencionadas na seção 2.1. O objetivo é apontar alguns conceitos úteis para a determinação dos requisitos de funcionalidade para Sistemas de Gerência de Banco de Dados para Aplicações não Convencionais, objeto do Capítulo 3.

Usamos para exemplificar, em algumas sub-seções, um objeto DFD. Este objeto representa um diagrama de análise estruturada (GANE 1978) que se deseja que seja convenientemente armazenado e controlado por um Sistema de Gerência de Banco de Dados. A descrição que se segue, é a definição informal do objeto. De acordo com as necessidades de cada sub-seção, podem ter sido incluídos atributos, entidades ou relacionamentos adicionais, necessários para realçar algum aspecto que se queira.

Um DFD é composto de diversos PROCESSOS, FLUXOS de dados, ENTIDADES EXTERNAS e DEPÓSITOS DE DADOS. Um PROCESSO, no decorrer da análise pode ser explodido em outros PROCESSOS, e assim sucessivamente. FLUXOS e DEPÓSITOS DE DADOS armazenam informações sobre DADOS, que podem ser elementares ou não. Um FLUXO pode conter vários DADOS, assim como um DADO pode aparecer em diversos FLUXOS. Um DADO normalmente aparece em somente um DEPÓSITO DE DADOS, pois a intenção é produzir depósitos de dados normalizados; mas um DEPÓSITO DE DADOS pode, naturalmente, conter diversos DADOS.
Um PROCESSO quando é explodido pode revelar novos FLUXOS, DEPÓSITOS DE DADOS, ENTIDADES EXTERNAS e DADOS.

A seguir, passamos à discussão destas propostas.

II.2.1) Extensões do System R.

Uma das propostas mais tradicionais na literatura, para implementar bancos de dados que suportem aplicações não convencionais, mais notadamente na área de engenharia, é a extensão do System R (HASKIN 1981, LORIE 1982, LORIE 1984, DITTRICH 1985). Estes trabalhos tratam dos seguintes problemas:

- definição de campos longos;
- definição de objetos complexos;
- extensões na linguagem SQL para suportar campos longos e objetos complexos;
- soluções para implementação eficiente de objetos complexos, incluindo modificações no otimizador de consultas;
- soluções para o tratamento de transações longas e acesso concorrente;
- versões de objetos.

Nesta sub-seção, trataremos dos 3 primeiros aspectos.

A necessidade de manipular, em um banco de dados, objetos de tamanho arbitrário, como: documentos, gráficos, imagens, ou mesmo voz codificada, faz com que seja necessário um novo tipo de dado capaz de armazenar estas informações. Uma solução, no modelo relacional, para este problema, seria distribuir o objeto em diversas tuplas, criando-se um atributo extra para simular uma ordenação destas tuplas. Esta solução é fraca pois exige o uso de uma estrutura de agregação (relação) para representar um conceito atômico. Seguem, em decorrência, problemas como: linguagem inadequada para manipulação, controles operacionais insuficientes, entre outros.

Os comandos FETCH, GET, INSERT, PUT e UPDATE da linguagem SQL foram estendidos para permitir manipular um campo longo em pedaços.

Basicamente, um campo longo pode ser manipulado através de um cursor, da mesma maneira que as tuplas de uma relação. Quando uma sentença FETCH é executada, uma tupla é recuperada e são abertos os cursores, um para cada campo longo que exista na relação. A partir daí, é possível recuperar, sequencial ou randomicamente o campo longo, informando-se o deslocamento e o tamanho do pedaço a ser tratado. A sintaxe para isto é:

```
GET <cursor_de_campo_longo> INTO <variável>
     STARTING AT <deslocamento> FOR <tamanho> BYTES;
```

Ao inserir uma nova tupla, os cursores de campos longos são automaticamente abertos, sendo possível construir o campo longo acrescentando-se pedaços ao final do mesmo, sucessivamente:

```
PUT <cursor_de_campo_longo> FROM <variável>
     FOR <tamanho> BYTES;
```

Na modificação de uma tupla, é possível trocar um pedaço de um campo longo por outro de tamanho qualquer:

```
PUT <cursor_de_campo_longo> FROM <variável>
     FOR <tamanho_novo> BYTES
     STARTING AT <deslocamento>
     REPLACING <tamanho_antigo> BYTES;
```

Objetos Complexos, na extensão do System R, são entidades básicas do mundo real que só podem ser representadas no modelo relacional, através da distribução dos elementos que compõem o objeto, em tuplas de relações diferentes.

Um objeto complexo é composto de uma relação raiz e outras componentes. Todas as relações (com exceção da raiz) são subordinadas à relação raiz ou a alguma outra. Uma relação só pode ser subordinada a, no máximo, uma outra relação, e uma relação não pode ser subordinada a si mesma ou a qualquer uma de suas subordinadas. Assim, um objeto complexo tem a estrutura de uma árvore.

Para declarar um objeto complexo, foram introduzidos três novos tipos de dados: IDENTIFIER, REF e COMP_OF.

O tipo IDENTIFIER funciona como um "surrogate", isto é, uma chave primária que tem seu valor gerado pelo sistema no instante de criação da tupla. O valor é único no sistema (e não só na relação). Se uma tupla for excluída, o seu IDENTIFIER jamais é reutilizado. O objetivo dos identificadores é realizar a ligação de composição entre as tuplas que constituem o objeto complexo, de forma que uma tupla de uma relação subordinada tem o valor do IDENTIFIER da tupla a qual está subordinada, em um atributo do tipo COMP_OF.

Uma vez que o sistema conhece a estrutura do objeto, através de atributos identificadores e de composição, várias restrições de integridade podem ser mantidas: uma tupla não pode ser incluída como subordinada de uma outra que não exista e a eliminação de uma tupla causa a eliminação, recursivamente, de todas as tuplas subordinadas.

Além disto, outras vantagens podem ser alcançadas:

- o objeto pode ser eficientemente armazenado e recuperado;
- o sistema pode bloquear o objeto como um todo no acesso concorrente;
- novas operações podem operar no objeto como um todo (por exemplo: copiar);
- a expressividade da linguagem SQL pode ser aumentada.

Da mesma forma que o atributo COMP_OF, o atributo REF (referência) tem como domínio valores de IDENTIFIERS, porém a semântica do relacionamento introduzido por ligações REF/IDENTIFIER é bem diferente daquela dos relacionamentos COMP_OF/IDENTIFIER. Neste último, o relacionamento é mais forte; o sistema usa este conhecimento para implementar as facilidades e restrições descritas acima.

Já o relacionamento implementado por REF/IDENTIFIER não tem a semântica de composição, não ficando, portanto, restrito à estrutura de árvore. A vantagen de usar

relacionamentos REF/IDENTIFIER é a mesma de se usar "surrogates", ou seja, se este relacionamento fosse implementado via uma chave de usuário, a integridade referencial ficaria mais difícil de ser mantida.

Em trabalhos posteriores, o tipo REF foi dividido em dois. INTERNAL_REF identifica a referência a uma tupla do mesmo objeto, enquanto EXTERNAL_REF é uma referência para uma tupla de outro objeto. Esta diferença serve para efeito de implementação (referências internas podem ser representadas gastando-se menos bytes na tabela de mapeamento). Nenhuma operação da LMD diferencia as duas formas, mas é possível distinguir os mecanismos de integridade referencial.

O objeto DFD pode ser representado de acordo com a figura 2.1. A definição parcial do mesmo é:

```
CREATE TABLE processos
                ( p_id (IDENTIFIER),
                  p_no (INTEGER),
                  descrição (CHAR(20)),
                  proc_super (EXT_REF (processos)) )

CREATE TABLE fluxos
                ( f_id (IDENTIFIER),
                  processo (COMP_OF (processos)),
                  nome_f (CHAR(25)) )

CREATE TABLE dados_em_fluxos
                ( df_id (IDENTIFIER),
                  fluxo (COMP_OF (fluxos)),
                  dados (INT_REF (dados)) )

CREATE TABLE depósito_dados
                ( d_id (IDENTIFIER),
                  nome_dep (CHAR(25)),
                  processos (COMP_OF (processos)) )

CREATE TABLE dados
                ( dados_id (IDENTIFIER),
                  nome_dado (CHAR(25)),
                  tipo_dado (CHAR(10)),
                  depósitos (COMP_OF (depósito_dados)) )
```

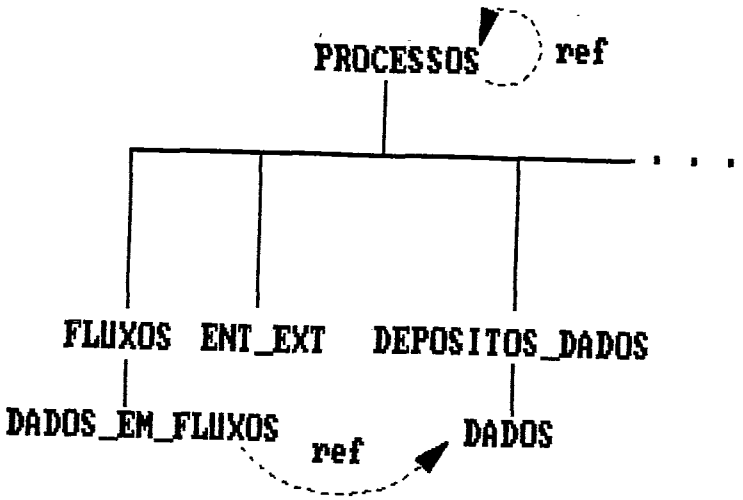


Figura 2.1 - Representação do Objeto DFD no System R

O objeto DFD mostra exemplos de referências externa e interna. Repare a necessidade de uma tabela dados_em_fluxos para modelar relacionamentos N:M entre fluxos e dados. Em um sentido (fluxos para dados_em_fluxos) podemos estabelecer uma relação de composição. No outro (dados_em_fluxos para dados) só podemos estabelecer uma relação de referência (compare com o modelo MAD, no ítem 2.2.2).

O mesmo acontece para representar o fato de um processo ser formado por outros processos. A proposta de extensão da SQL não enxerga esta modelagem.

No exemplo, consideramos um relacionamento 1:N entre depósitos de dados e dados, supondo que os diagramas tem os dados normalizados.

Para a manipulação do objeto complexo, duas facilidades são fornecidas.

Primeiro, a declaração de um cursor complexo facilita a recuperação de tuplas do objeto na SQL embutida em linguagens hospedeiras.

Segundo, a declaração de um equi-join, entre relações que participam de uma relação de composição, forma uma

junção implícita que pode ser usada em consultas subsequentes. Por exemplo, podemos declarar:

```
P_F := JOIN (processos,fluxos | processos.p_id =
                                    fluxos.processo)
```

E usar em uma consulta SQL:

```
select descrição, nome_f
from p_f
where p_no = 1.1
```

O uso de identificadores introduz a função KEY (e sua inversa ID) para recuperar a chave primária definida pelo usuário a partir de um IDENTIFIER (existe uma correspondência biunívoca entre eles). A consulta abaixo inclui, na resposta, a chave p_no. Na cláusula FROM não é necessário usar uma junção implícita entre as relações fluxos e processos.

```
select nome_f, key(processos)
from fluxos
where nome_f = "fluxo_2.3"
```

Para incluir um fluxo de dados, usamos a função inversa ID:

```
insert into fluxos values ("fluxo_2.4", ID (1.1))
```

A manipulação de objetos não fica muito elegante, pois o usuário deve ser sabedor da existência do identificador, porém, é evitado o uso de chaves concatenadas ao longo da hierarquia de objetos componentes.

II.2.2) O Sistema de Gerência de Banco de Dados PRIMA.

PRIMA (HARDER 1987) é um protótipo de um núcleo de SGBD para aplicações na área de engenharia baseado no modelo de dados Molécula-Átomo (MAD). Analisaremos neste trabalho, particularmente, as capacidades do modelo de dados para lidar com objetos complexos. MAD é um modelo de dados que pode ser considerado orientado a objetos, mas a linguagem MQL, para manipulação de suas estruturas, é, na verdade, uma extensão da álgebra relacional, pois é baseada na linguagem SQL. Os requisitos de projeto que nortearam a definição do modelo foram:

- orientação a objetos, permitindo a manipulação de objetos complexos;
- representação direta de relacionamentos N:M com capacidade de percorrer simetricamente os objetos;
- construção dinâmica de objetos complexos;
- orientação a conjuntos.

Um objeto complexo no MAD é modelado como um conjunto de átomos relacionados. Um átomo define uma unidade básica percebida pelo usuário (uma entidade, usando-se a nomenclatura do modelo ER). Como tal, possui atributos de vários tipos primitivos (como: inteiro, real, cadeia de caracteres e outros), sendo que um deles funciona como identificador. Além dos tipos primitivos, MAD suporta dois tipos especiais para representar os relacionamentos entre átomos. O tipo IDENTIFIER serve como uma chave primária única, gerada e mantida pelo sistema (um "surrogate"). Todo átomo deve possuir um identificador. O tipo REFERENCE permite que um atributo aponte para outros átomos, representando relacionamentos (como nos modelos semânticos funcionais - vide 2.2.3). Um atributo do tipo REFERENCE pode ser multivalorado, bastando para isto declarar que o atributo é um conjunto de referências, com a sintaxe :

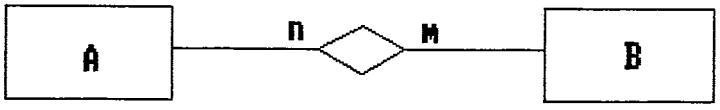
SET_OF (REF_TO (atributo))

Uma característica importante do modelo é a obrigatoriedade em se manter a simetria entre os relacionamentos entre átomos.

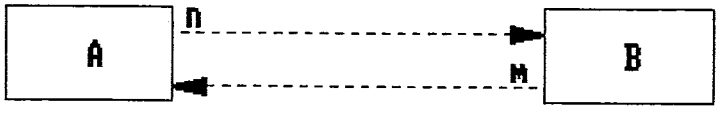
A figura 2.2 mostra um relacionamento N:M entre duas entidades (átomos) A e B, com a representação no modelo ER (a) e MAD (b). A correspondente declaração em MAD é:

```
create atom_type A
      (...
       b: SET_OF (REF_TO (B.a))
       ...)

create atom_type B
      (...
       a: SET_OF (REF_TO (A.b))
       ...)
```



(a)



(b)

Figura 2.2 - Representação de Relacinamentos N:M em MAD

Este tipo de declaração simétrica permite a modelagem de relacionamentos com qualquer cardinalidade, bem como a possibilidade de "entrar" no objeto, através da linguagem de consulta, em qualquer "sentido".

Restrições de integridade do sistema garantem que qualquer mudança no valor do atributo b de A é, automaticamente, acompanhada de uma mudança no seu inverso, isto é, no atributo a de B (e vice-versa).

A figura 2.3 representa o esquema do objeto DFD no modelo MAD. Segue a definição na Linguagem de Descrição de Dados.

```
CREATE ATOM_TYPE  processo
     (p_id:       IDENTIFIER,
      p_no:       INTEGER,
      descrição:  CHAR_VAR,
      sub:        SET_OF (REF_TO(processo.super)),
      super:      REF_TO (processo.sub),
      e_ext:      SET_OF (REF_TO(entidade_ext.processo),
      d_dados:    SET_OF (REF_TO(dep_dados.processo)),
      fluxos:     SET_OF (REF_TO(fluxos.processo)))
  KEYS_ARE (p_no)


CREATE ATOM_TYPE  entidade_ext
     (ee_id:      IDENTIFIER,
      nome:       CHAR_VAR,
      processo:   REF_TO(processo.e_ext))
  KEYS_ARE (nome)


CREATE ATOM_TYPE  dep_dados
     (dd_id:      IDENTIFIER,
      nome:       CHAR_VAR,
      processo:   REF_TO (processo.d_dados),
      dados:      SET_OF (REF_TO(dados.depositos)))
  KEYS_ARE (nome)


CREATE ATOM_TYPE  fluxos
     (fx_id:      IDENTIFIER,
      nome:       CHAR_VAR
      processo:   REF_TO (processo.fluxos),
      dados:      SET_OF (REF_TO(dados.fluxos)))
  KEYS_ARE (nome)


CREATE ATOM_TYPE  dados
     (ds_id:      IDENTIFIER,
      nome:       CHAR_VAR,
      tipo:       CHAR_VAR,
      depositos:  REF_TO (dep_dados.dados),
      fluxos:     SET_OF (REF_TO(fluxos.dados)))
  KEYS_ARE (nome)
```

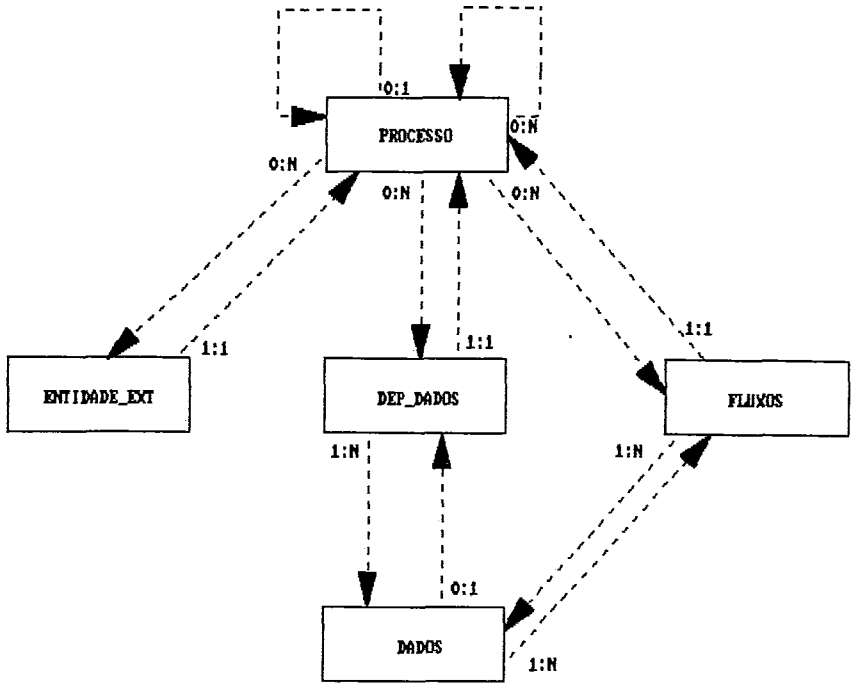


Figura 2.3 - Representação gráfica do objeto DFD em MAD.

Os átomos e suas interligações formam Moléculas. Uma molécula é uma forma de ver um grupamento de átomos, similar ao conceito de visão do modelo relacional. Moléculas podem ser pré-declaradas, correspondendo a visões normalmente tomadas pelo usuário, ou declaradas como parte de uma consulta MQL. Por exemplo, algumas moléculas pré declaradas do objeto DFD podem ser:

```
(a) DEFINE MOLECULE_TYPE composição_processos
    FROM processo.sub - processo (recursive)

(b) DEFINE MOLECULE_TYPE dfd
    FROM composição_processos - entidades_ext -
         dep_dados - fluxos - dados

(c) DEFINE MOLECULE_TYPE depósitos_&_dados
    FROM dep_dados - dados

(d) DEFINE MOLECULE_TYPE fluxos_&_dados
    FROM  fluxos - dados
```

É possível definir moléculas que são associações recursivas, como no caso (a), moléculas que são definidas a partir de outras moléculas e agrupam um objeto inteiro (b) e ainda sub-objetos compartilhados, como o átomo DADOS nas moléculas definidas em (c) e (d).

Como o sistema conhece as interligações entre os átomos, é possível simplesmente, em uma consulta, referenciar à molécula. O sistema junta as partes a partir do primeiro átomo declarado na cláusula FROM, atráves dos atributos do tipo REF_TO.

MAD define dois tipos de acessos a objetos. O acesso vertical é caracterizado por ser o acesso ao objeto como um todo, agrupando todos os seus componentes. Este acesso pode ser através de um critério de seleção que envolva um atributo que não faça parte do átomo de "entrada" do objeto. Repare que no MAD não há sentido em falarmos de raiz, pois os objetos podem ser simétricos.

O acesso horizontal permite recuperar todos os átomos de um mesmo tipo (em objetos diferentes), baseado em um critério.

Além destes, ainda distinguimos um outro tipo de acesso, muitas vezes necessário, também suportado pela MQL. Trata-se do acesso vertical, porém, apresentando na resposta somente parte do objeto complexo selecionado.

No MAD, o acesso vertical é conseguido independentemente da direção em que se olhe o objeto. Por exemplo: em MQL, podemos realizar a seguinte consulta que recupera todas as moléculas DFD (acesso vertical), ou seja, todos os átomos que constituem um DFD. O número zero, entre parêntesis, informa o nível de recursão onde deve ser procurado o processo cujo p_no é igual a 1.1.

```
SELECT all
FROM   dfd
WHERE  dfd(0).p_no = 1.1
```

Como exemplo de acesso horizontal, poderíamos recuperar todos os tipos de dados "inteiro" em qualquer instância de um dfd:

```
SELECT nome,tipo
FROM   dados
WHERE  tipo = "inteiro"
```

Finalmente, um acesso vertical com recuperação de somente parte do objeto

```
SELECT processo.p_no, entidade_ext.nome
FROM   dfd
WHERE  dfd(0).p_no > 2
```

O modelo não explicita como os resultados são retornados.

Além do aspecto estrutural e operacional mostrados até aqui, o modelo MAD mantém uma série de restrições de integridade, como, por exemplo, a restrição de simetria já mencionada. Operações especiais são definidas para conexão e desconexão de moléculas e eliminação em cascata opcional.

O modelo MAD parece ser bastante promissor em relação a manipulação de objetos complexos, principalmente por permitir recursão, modelagem de relacionamentos N:M (e não simples hierarquias de objetos complexos), simetria de acesso e manipulação a nível molecular e/ou atômico, permitindo o trabalho em diversos níveis de abstração ou em perspectivas diferentes. A linguagem MQL é bastante poderosa em termos de recuperação de objetos pelos critérios mais variados. Nos parece, porém, que uma linguagem que permita navegação mais simples através do objeto, poderia ser também muito útil, principalmente em se tratando de objetos "mais" complexos. No PRIMA, detalhes de implementação, para permitir eficiência, estão sendo cuidadosamente considerados,segundo a literatura. No primeiro protótipo, uma molécula pode conter, no máximo, um objeto com definição recursiva.

II.2.3) Modelo Semântico IFO

Enquanto a maioria dos modelos semânticos que surgiram nos últimos anos (ABRIAL 1974, CHEN 1976, CODD 1979, HAMMER 1981, KING 1982, SU 1983, BRODIE 1984) tinham como finalidade ajudar no processo de análise conceitual de um minimundo, buscando a construção de esquemas que capturassem mais informações sobre a realidade, o modelo IFO (ABITEBOUL & HULL 1987) propõe uma base matemática formal, para a investigação teórica dos tópicos relacionados com a representação do problema. Sua criação é fundamentada no amadurecimento dos conhecimentos adquiridos com a pesquisa dos modelos semânticos anteriores, e por isto foi escolhido para ser discutido neste trabalho.

O modelo IFO enfoca, principalmente, o aspecto estrutural da definição dos objetos e seus relacionamentos. Os autores argumentam que a característica formal do modelo provê uma base adequada para o desenvolvimento dos aspectos de manipulação do objeto e definição de restrições de integridade.

Quatro são os princípios básicos de modelagem semântica percebidos pelo modelo IFO :

O primeiro afirma que os dados sobre os objetos de um sistema e os relacionamentos entre eles devem ser representados de uma forma direta, sem a necessidade de usar identificadores simbólicos ou ponteiros que desviam a atenção do problema sendo modelado.

O segundo princípio básico coloca que a maioria dos relacionamentos entre objetos são de natureza funcional, como evidenciado pelo modelo funcional (KERSCHBERG 1976).

O terceiro princípio básico é a necessidade de representar relacionamentos do tipo "is-a", comuns em modelos orientados a objetos (vide 2.2.6 e 2.2.7). Estes tipos de relacionamentos introduzem uma ligação especial

entre os objetos envolvidos, diferentemente dos relacionamentos de natureza funcional, pois envolvem herança de tipos e operações entre os objetos.

Finalmente, o quarto princípio revela a necessidade de existir operadores que permitam a construção de tipos (de objetos) a partir de outros tipos já definidos, criando-se objetos estruturados. Os construtores identificados são, basicamente, o de agregação e de grupamento. Uma agregação (ou concatenação cartesiana) permite a criação de um tipo que é uma ênupla ordenada de tipos já declarados (por exemplo, uma relação no modelo relacional). A construção de grupamento permite criar um tipo que é um conjunto de elementos de outro tipo já existente.

Embora o modelo IFO considere que a maioria dos relacionamentos sejam de natureza funcional (segundo princípio básico), o próprio autor, descreve em outro trabalho (HULL 1987) que existem duas grandes abordagens filosóficas para representação de relacionamentos nos modelos semânticos, as quais geram esquemas profundamente diferentes para o mesmo problema sendo modelado.

Uma abordagem, adotada pelo IFO, é justamente a abordagem funcional, onde os relacionamentos são representados via atributos dos objetos. A outra se baseia, primariamente, no uso de um construtor de tipo (por exemplo agregação) para representar o relacionamento entre objetos.

Por exemplo: para modelar um relacionamento 1:N entre duas entidades A e B, nos modelos funcionais, deve-se criar um atributo multivalorado em A cujo domínio é B. Este atributo também é visto como uma função (embora seja multivalorado). Ao mesmo tempo, existe um atributo em B, monovalorado, cujo domínio é A, que representa a função inversa do atributo descrito anteriormente (vide figura 2.4).

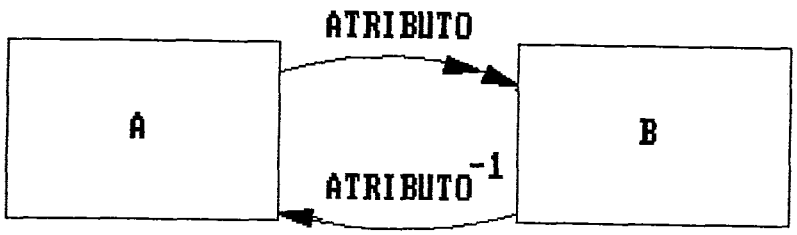


Figura 2.4 - Relacionamentos 1:N representados por funções.

A seta com duas pontas representa um atributo ("função") multivalorado, enquanto a seta com uma ponta representa um atributo monovalorado. Atributos podem ser ainda compostos de um argumento (seta com uma cauda), onde se representa um relacionamento binário entre duas entidades, ou de N argumentos (seta com N caudas) que representa o relacionamento entre um conjunto de N entidades e uma entidade (vide figura 2.5).

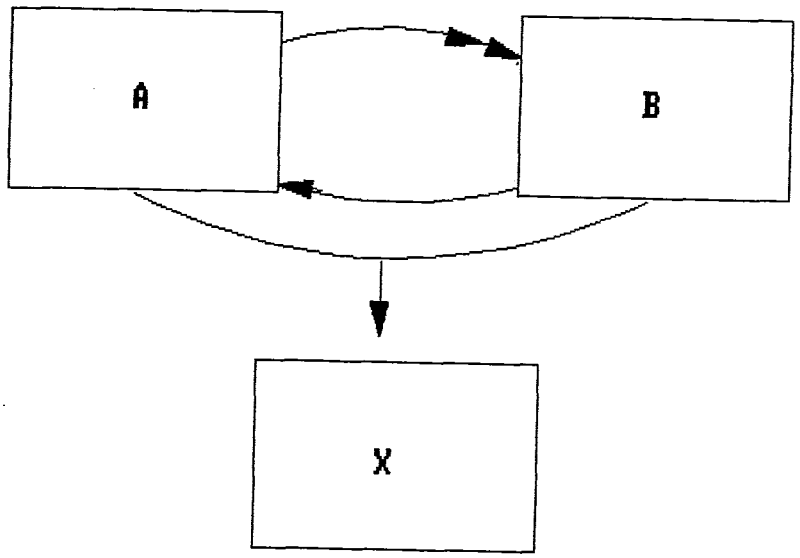


Figura 2.5 - Relacionamentos funcionais binários e n-ários.

Já em modelos que se baseiam em construtores de tipos para representar relacionamentos, como, por exemplo, o modelo de ER (CHEN 1976), o relacionamento 1:N entre as entidades A e B é representado por uma outra entidade C que é um conjunto de pares ordenados. Na figura 2.6, uma agregação é representada por um círculo com um sinal de multiplicação.

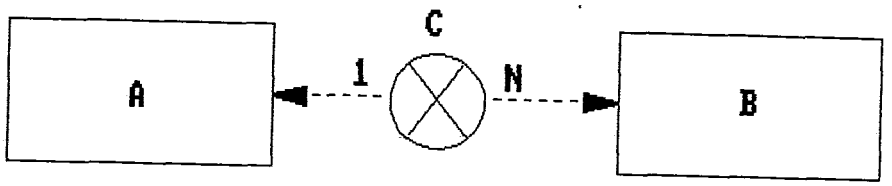


Figura 2.6 - Relacionamento 1:N representado por agregação.

Um esquema IFO pode ser representado, graficamente, por um conjunto de vértices e linhas formando um grafo. O esquema é formado de:

i) objetos que podem ter representação direta (um símbolo gráfico) ou não (quando for um objeto construído a partir de outro);

ii) fragmentos que representam diretamente relacionamentos funcionais;

iii) relacionamentos "is-a".

Cada um destes ítens é brevemente analisado a seguir.

II.2.3.1 Objetos

No esquema IFO um objeto é um tipo. Existem três tipos atômicos, a saber : apresentável (printable), abstrato e livre e ainda tipos complexos, formados a partir de dois construtores: tupla e conjunto.

Objetos atômicos representáveis são objetos pré-definidos e que servem para entrada e saída. Constituem a porção visível do banco de dados. Estes tipos dependem da aplicação e do ambiente em questão. Normalmente são formados dos tipos tradicionais de linguagem de programação : inteiro, real, cadeia de caracteres, podendo incorporar tipos não tão usuais como gráficos, ícones e outros.

Um objeto atômico abstrato corresponde a objetos percebidos, diretamente do mundo real, pelo usuário. Podem representar uma entidade concreta ou um conceito. Do ponto de vista do usuário, um tipo abstrato não tem estrutura (pelo menos no nível de abstração em que é percebido) embora possa ter atributos e subtipos. Um tipo abstrato não pode ser visualizado, mas é possível, por exemplo, exibir um de seus atributos que seja um objeto atômico apresentável. Um objeto atômico livre é aquele cuja estrutura é herdada através de um relacionamento "is-a".

Qualquer um dos três tipos atômicos pode ser usado na construção de tipos complexos. O construtor de tuplas permite a aplicação da noção de produto cartesiano, enquanto o construtor de conjunto permite formar coleções de objetos do mesmo tipo. Estes construtores podem ser aplicados, recursivamente, em qualquer ordem, porém, o modelo omite o tratamento de casos onde um objeto possa ser constituído de partes dele mesmo.

Objetos atômicos representáveis, abstratos e livres são representados respectivamente por quadrados, losangos e círculos. Um construtor de tuplas é representado por um círculo contendo um sinal de multiplicação (chamado de vértice em cruz), enquanto o construtor de conjuntos é um círculo contendo um asterisco (vértice estrela). Cada um destes símbolos que representam construtores é ligado por arcos (setas são usadas para fragmentos) aos tipos que os formam (vide figura 2.9).

II.2.3.2) Fragmentos

Um fragmento é um "sistema" constituído de objetos atômicos ou complexos inter_relacionados através de funções. Uma função é, intuitivamente, um atributo. No modelo IFO, "funções" multivaloradas não são permitidas, devendo-se usar, para modelar tal caso, uma função simples e um construtor de conjuntos. Funções são representadas por setas ligando o argumento ao domínio.

O uso de fragmentos permite representar atributos dependentes de contexto. Na figura 2.7(a) vemos uma função f que mapeia uma instância de A para uma instância de B. Na figura 2.7(b) o construtor de conjuntos faz com que uma instância de A participe de mais de uma instância de SET. Neste caso, uma instância de A mapeia, através da função f, mais de uma instância de B.

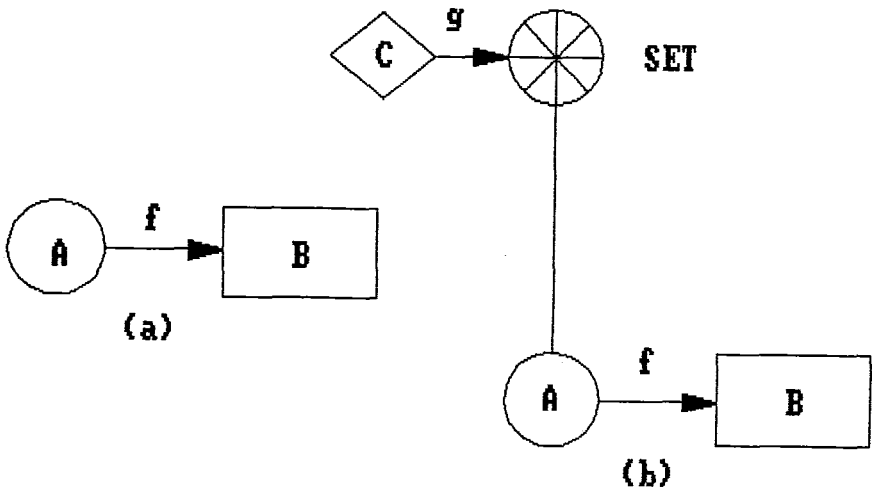


Figura 2.7 - Atributos dependentes de contexto.

Por exemplo, em 2.8(a) representamos o fato que cada estudante tem uma nota. Em 2.8(b), cada estudante tem uma nota para cada curso que participa.

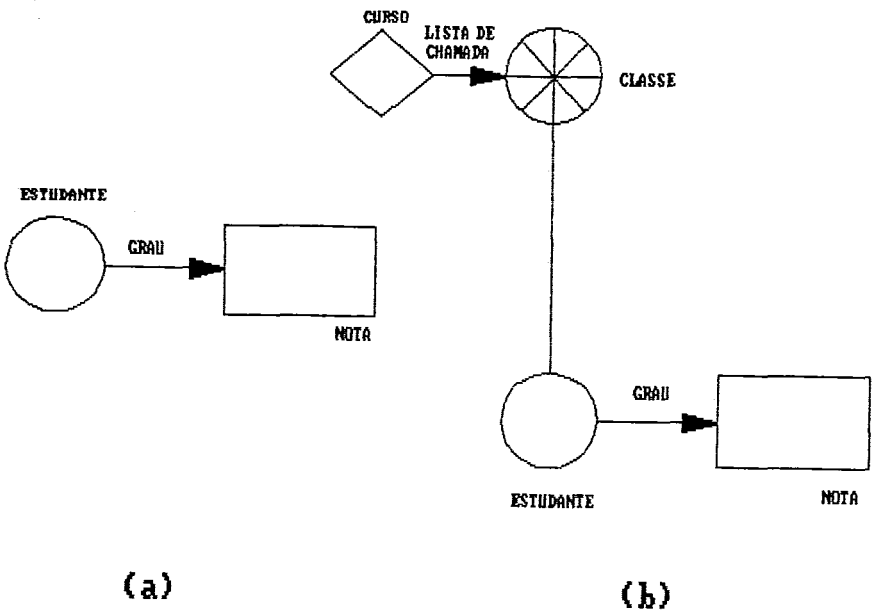


Figura 2.8 - Exemplo de atributos dependentes de contexto.

II.2.3.3) Relacionamentos "is-a"

O modelo IFO distingue dois tipos de relacionamentos "is-a". Um, chamado especialização, é representado, graficamente, por uma seta larga e não preenchida. Este é usado para modelar diversos papéis que um subtipo pode desempenhar. Por exemplo: um tipo abstrato PESSOA pode ter dois subtipos (especializações): EMPREGADO e ESTUDANTE. Uma instância (por exemplo: de ESTUDANTE) pode trocar de subtipo (virar um EMPREGADO) sem nenhum problema. Os conjuntos de instâncias, que compõem cada subtipo não precisam ser disjuntos (um ESTUDANTE pode ser, também, um EMPREGADO). Intuitivamente, o subtipo de uma especialização herda o tipo (e funções/atributos) do supertipo. Diferentemente do modelo funcional, especializações são usadas também para modelar uma restrição ao intervalo de uma função.

O outro tipo de relacionamento "is-a" é chamado de generalização, sendo representado por uma seta larga e preenchida. Neste caso, o supertipo é visto como um novo tipo formado da combinação dos seus subtipos. Por exemplo um supertipo VEÍCULOS pode ter os subtipos LANCHAS e CARROS. Não faz sentido uma instância trocar de tipo e é requerido que os conjuntos dos subtipos cubram o supertipo, ou seja, a união dos conjuntos CARROS e LANCHAS seja igual ao conjunto de VEÍCULOS. Surpreendentemente, neste caso, a herança de tipo ocorre de baixo para cima, isto é, são aplicáveis no supertipo as funções definidas para os subtipos.

Na figura 2.9, o objeto DFD é representado usando o modelo IFO.

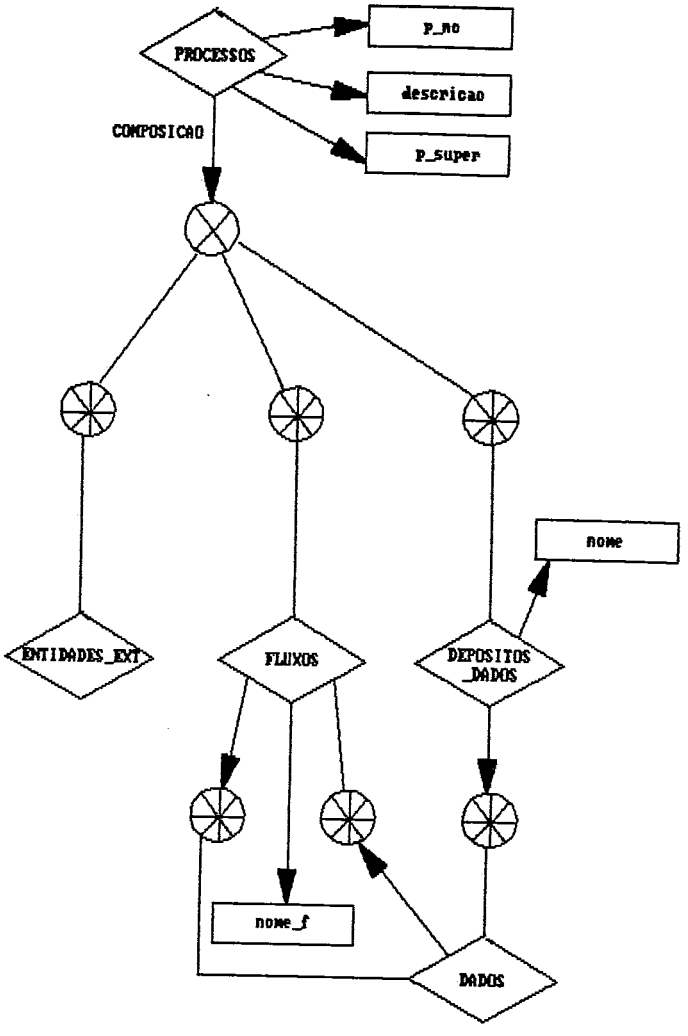


Figura 2.9 Objeto DFD no modelo IFO.

II.2.4) O Modelo Semântico SAM*

SAM* - Semantic Association Model - é um modelo de dados semântico desenvolvido, principalmente, para bancos de dados estatísticos (SU 1983) e, posteriormente, refinado para aplicações na área de CAD/CAM (SU 1986). O modelo incorpora facilidades para tratamento de:

- tipos primitivos complexos (como vetores, matrizes, tempo, conjunto e conjunto ordenado entre outros);
- relacionamentos temporais;
- hierarquias de estrutura de dados;
- definição recursiva de objetos;
- objetos complexos;
- modelagem de dados particionados e replicados;
- modelagem de versões;
- dados derivados (atributos que sumarizam categorias de objetos).

Um esquema representado em SAM* é uma rede de conceitos ligados por meio de associações. Um conceito é o que conhecemos, intuitivamente, por objeto. Existem conceitos atômicos que representam uma entidade física, abstrata ou evento não decomponível. Conceitos atômicos são representados pelos tipos comuns em linguagens de programação, tais como: inteiro, real, cadeia de caracteres e outros. Como o modelo é orientado para aplicações científicas, conjuntos, vetores e matrizes são, também, conceitos atômicos. Um conceito não atômico é um grupamento de outros conceitos (atômicos ou não), através de associacões.

Existem sete formas de associação. Cada uma tem, em si, um aspecto estrutural, operacional e de restrição semântica.

O modelo SAM* recomenda alguns conceitos atômicos (tipos de dados) que devem ser suportados por qualquer SGDB que implemente o modelo. Outros tipos de dados, úteis para uma determinada classe de aplicações (por exemplo CAD/CAM),

devem ser implementados, sendo criado operadores na linguagem de manipulação de dados para lidar com eles. Implicitamente, o que o modelo sugere, na verdade, é a incorporação de uma facilidade de tipos abstratos de dados.

Alguns destes conceitos atômicos sugeridos são:

- tradicionais: inteiro, real, cadeia de caracteres, lógico;
- estruturados: set, ordered-set, duplicate-set, vetor, matriz;
- temporal: dia, hora;
- monetários: dolar, cruzado novo;
- especiais: compute, rule.

O tipo "duplicate-set" é um "conjunto" onde são permitidos elementos duplicados. O tipo "compute" permite armazenar fórmulas ou parâmetros de entrada para um procedimento que computa o valor do atributo em tempo de execução. Estas fórmulas ou procedimentos podem usar, também, outros atributos do objeto para calcular valores. Este tipo de dado é uma simplificação dos tipos procedurais que podem ser definidos no POSTGRES (vide 2.2.8). Finalmente o tipo "rule" tem função similar ao compute, armazenando, porém, regras para sistemas especialistas.

Conceitos atômicos homogêneos são grupados através de uma associação do tipo PARTICIPAÇÃO (MEMBERSHIP) para formar domínios. Graficamente um círculo com a letra M representa esta forma de associação.

Um conceito não atômico pode ser formado através de uma associação do tipo AGREGAÇÃO. Uma agregação é representada, graficamente, por um círculo contendo um A e setas ligando este círculo a outras formas de associação (vide figura 2.10).

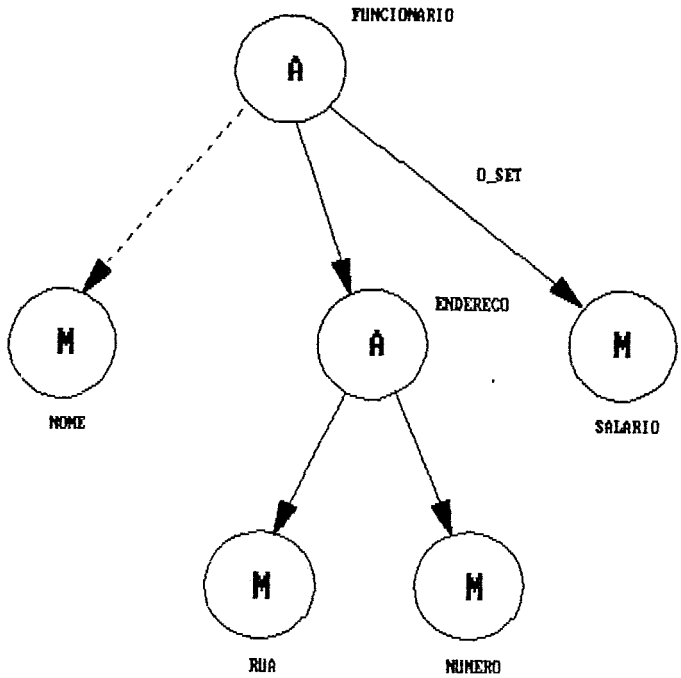


Figura 2.10 - Associações de participação e agregação.

No exemplo, NOME, RUA, NÚMERO e SALÁRIO são domínios (associações M) cujos conjuntos atômicos são, respectivamente, cadeia de caracteres, cadeia de caracteres, inteiro e cruzado. ENDEREÇO é uma agregação formado por RUA e NÚMERO. As setas que ligam ENDEREÇO a RUA e NÚMERO podem ser vistas como atributos de ENDEREÇO. Se o nome do atributo for diferente do domínio, as setas podem ser rotuladas.

No SAM* não existe uma associação especial para representar conjuntos, como em vários modelos semânticos (IFO, SDM, e outros). Assim, um atributo pode definir uma estrutura adicional sobre o domínio, como por exemplo: set, ordered-set, vetor, matriz. Na figura 2.10, um rótulo "O_SET" aparece indicando que um funcionário tem diversos salários (um histórico, por exemplo) formando um ordered-set (a ordem das intâncias é importante).

Um dos atributos de uma agregação deve ser a chave primária de acesso às instâncias da associação, como no

modelo relacional. O atributo chave é representado por uma seta tracejada.

Um objeto recursivo pode ser definido, também, usando-se atributos. Para tanto, basta construir dois atributos definidos sobre o mesmo domínio; um representa a chave e o outro, a relação de composição (vide exemplo na figura 2.11 - conceitos PROCESSO e ID_PROCESSO).

Relacionamentos genéricos entre conceitos são modelados através de uma associação de INTERAÇÃO, com o mesmo significado do uso de relacionamentos no modelo de ER (CHEN 1976). Como existe uma associação especial para este fim, a associação de AGREGAÇÃO não tem a responsabilidade (como em muitos modelos de dados semânticos) de representar relacionamentos. Assim, duas restrições de integridade são definidas para os conceitos de uma agregação: um conceito subordinado só pode ser criado atrelado ao seu pai e a eliminação de um conceito pai causa a eliminação em cascata de seus dependentes. Para a associação INTERAÇÃO existem, também, duas restrições de integridade. Uma diz respeito ao mapeamento (conhecido como cardinalidade do relacionamento no modelo ER) e pode ser grafado junto às setas que compõem a associação (1:1, 1:N e N:M). A outra garante que somente instâncias existentes podem participar de uma instância de uma associação de INTERAÇÃO (conhecido como integridade referencial no modelo relacional).

O objeto DFD, na figura 2.11, demonstra diversos aspectos do modelo discutido até aqui.

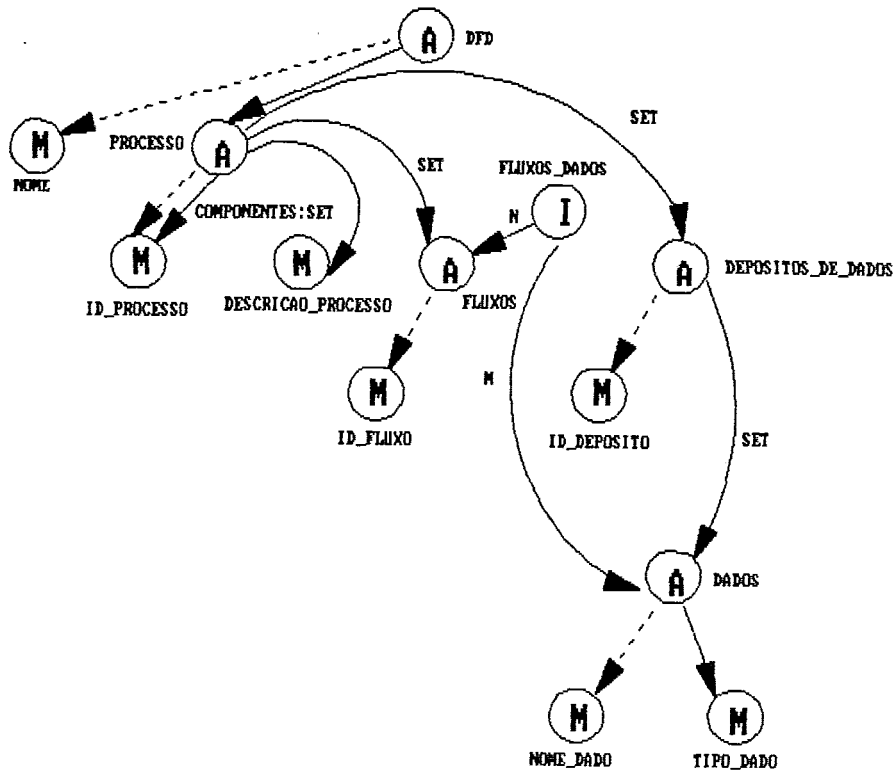


Figura 2.11 - O objeto DFD representado em SAM*.

A agregação PROCESSO está ligada por dois atributos ao domínio ID_PROCESSO. O primeiro atributo (seta), sem rótulo, indica que existe um atributo com o mesmo nome do domínio e que é chave (tracejado). O segundo atributo sobre o mesmo domínio tem o nome Componentes e é um cojunto de ID_PROCESSO, definindo, assim, o fato de que processos são formados por outros processos. Um processo contém, também, um conjunto de FLUXOS e um conjunto de DEPOSITOS_DE_DADOS. Uma associação do tipo INTERAÇÃO modela o fato de um ítem de dado (agregação DADOS) estar em vários FLUXOS; e um FLUXO possuir diversos ítens de dados (mapeamento N:M). Um DEPOSITO_DE_DADOS contém diversos ítens de dados, mas um ítem de dado só se encontra em um depósito (mapeamento 1:N), supondo-se que os dados capturados pelo DFD encontram-se normalizados (ULLMAN 1982). Neste caso, eliminar um depósito de dados causa a eliminação de todos os dados que ele contém.

A quarta forma de associação no SAM* é a GENERALIZAÇÃO, que permite modelar relacionamentos do tipo "is-a" . Uma

generalização é representada por um círculo grafado com a letra G (vide figura 2.12). No exemplo, o conceito ATIVIDADE é uma generalização dos conceitos ESTUDANTE e EMPREGADO. As instâncias de ATIVIDADE são constituídas pela união das instâncias de ESTUDANTE e EMPREGADO. Se um conceito de generalização (por exemplo ATIVIDADE) for o domínio de uma chave primária de uma associação de agregação (por exemplo PESSOA), os outros atributos da agregação (NOME e CPF) são herdados pelas associações que formam a generalização (ESTUDANTE e EMPREGADO).

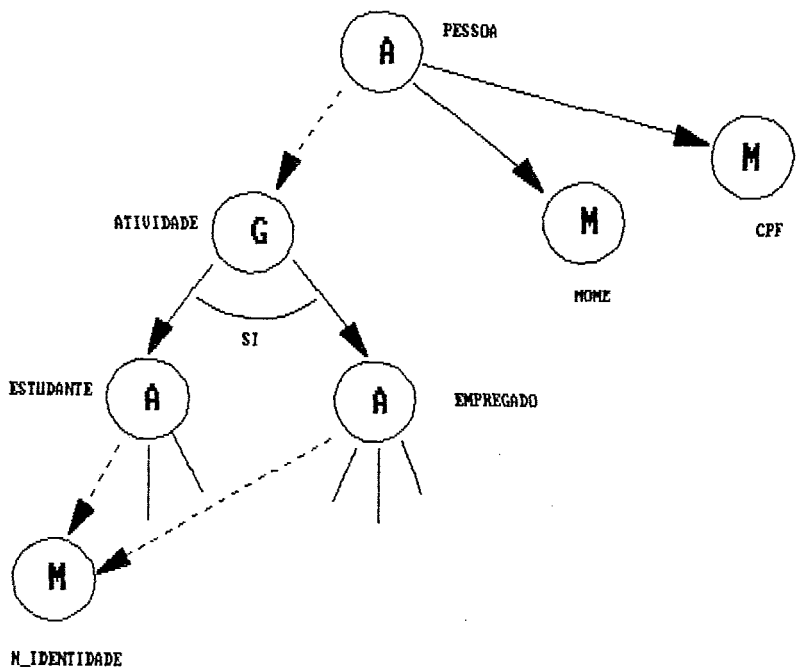


Figura 2.12 - Generalização no SAM*.

Algumas restrições de integridade são garantidas pelo modelo. Todos os conceitos que formam uma generalização devem ter o mesmo domínio como chave primária (N_IDENTIDADE). Isto sugere uma construção "bottom-up", uma vez que, em outros modelos de dados semânticos, N_IDENTIDADE seria visto como um atributo do supertipo PESSOA.

Além disto, pode ser especificado, em uma generalização, o tipo de interação entre os conceitos que a compõem. Os conjuntos de instâncias podem :

- Ser iguais (grafado SE - Set Equality);

- Possuir interseção vazia (grafado SX - Set Exclusion);

- Possuir interseção diferente de vazia (grafado SI - Set Intersection);

- Um ser o subconjunto de outro (grafado ST-SS - Set - Subset).

No exemplo da figura 2.12, um ESTUDANTE pode ser também um EMPREGADO (representado por SI).

Verificamos que numa generalização no modelo SAM* necessariamente existe a cobertura dos conjuntos que participam da associação (isto é, todas as PESSOAs são, forçosamente, ESTUDANTEs e/ou EMPREGADOs). O próprio fato da chave N_IDENTIDADE aparecer como atributo das agregações que fazem parte da generalização, e não do supertipo PESSOA, induz a este comportamento. Para modelar a possibilidade de PESSOAS não serem nem ESTUDANTES nem EMPREGADOS, seria necessário criar outra associação, sem atributos, participando da generalização ATIVIDADE. Esta forma de tratar generalizações, no modelo SAM*, difere bastante das propostas encontradas em modelos de dados semânticos e modelos de dados orientados a objetos.

A quinta forma de associação é a COMPOSIÇÃO, que cria uma entidade que pode ser nomeada e processada como um todo. Infelizmente, porém, este tipo de associação não pode ser usado para modelar objetos complexos, pois a sua definição exige que uma associação de composição contenha, em qualquer momento, somente uma única instância, que é formada por um conjunto de conjuntos, isto é, uma instância de uma associação de composição é formada pela união heterogênea de todas as instâncias que compõem os conceitos que participam da associação. Um exemplo de composição seria uma instância que representasse todos os dados (de todos os conceitos) da base de dados. Teríamos, por exemplo, um operador para copiar a base toda, aplicável na associação de composição.

Como a associação de composição aglutina conceitos heterogêneos, sua semântica foi estendida em (SU 1986) para modelar o problema de versões em banco de dados para CAD/CAM. Interessante de se notar é que não foi tratado o problema de um objeto com várias versões, mas, sim, o problema do próprio esquema mudar ao longo do tempo.

Na figura 2.13 são mostradas associações de composição para tratamento de versões do esquema. O exemplo mostra, também, a possibilidade de tratar versões embutidas. VERSOES_V1 indica a representação de 3 versões: V1, V1' e V1''. A versão V1, por sua vez, é composta, na verdade, pelas versões V1_XN e V1_XN', onde a diferença entre elas é somente o conceito XN, isto é, podemos representar variações graduais no esquema (X1 até XN-1 não se modificam) sem replicar os conceitos. Os números que aparecem como rótulos nas associações podem indicar a cronologia da criação.

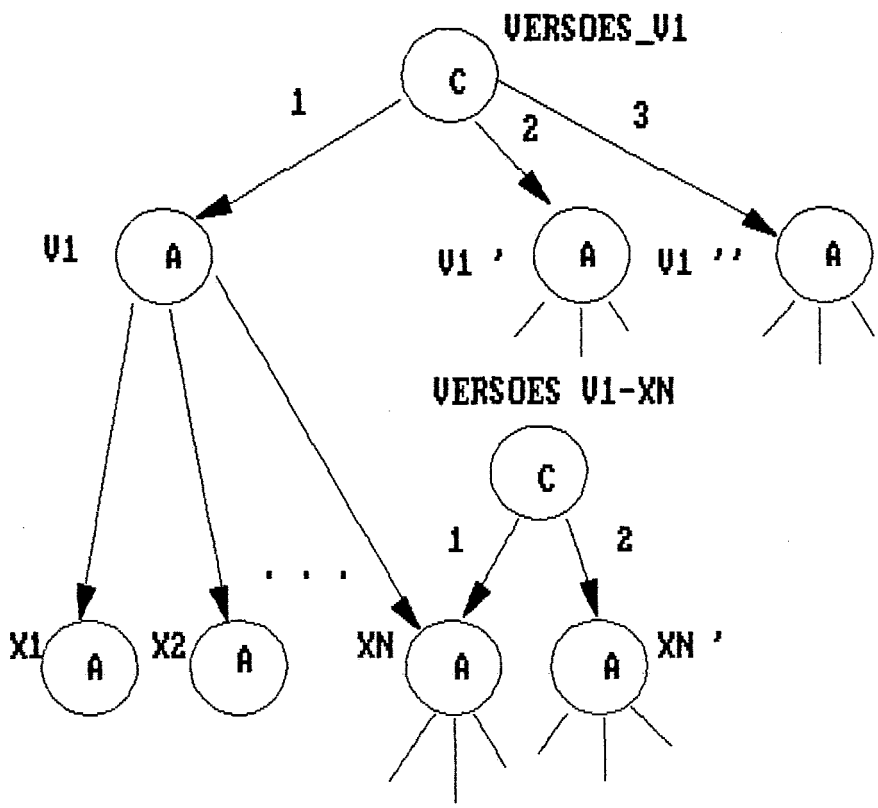


Figura 2.13 - Representação de versões no SAM*.

A sexta e sétima associações são mais usadas para bancos de dados estatísticos. A associação

PRODUTO_CARTESIANO representa uma agregação no sentido mais amplo da maioria dos modelos semânticos onde, porém, as ocorrências da associação não precisam ser representadas, pois são, na verdade, todo o produto cartesiano (e não um subconjunto) dos conceitos que compõem a associação. Finalmente, a associação de SUMARIZAÇÃO permite a composição de atributos que são funções estatísticas de algum conceito participando da associação.

Algumas das sete associações do SAM* redundam um pouco quanto ao aspecto estrutural. Operações e restrições semânticas sobre estas são, porém, bastante distintas.

O modelo permite representar algum relativismo pois é permitido que um nó seja grafado com mais de uma associação (por exemplo A,M) representando diversas visões ou diversas formas de modelar um determinado conceito.

SAM* é um modelo de dados que define, claramente, os aspectos estrutural, operacional e de integridade. As associações permitem a construção de objetos de complexidade bastante grande, inclusive recursivos. O uso do relativismo pode ser útil para a pesquisa de uma Linguagem de Manipulação de Dados, que permita manipular um objeto complexo em diversos níveis de abstração. O objeto pode ter operações que permitam manipulá-lo como um todo ou de suas partes componentes.

II.2.5) A linguagem FAD para Banco de Dados.

A linguagem FAD (BANCILHON 1987) foi incluída nesta revisão por aglutinar, no seu propósito e definição, três conceitos inter-relacionados com os aspectos que estão sendo explorados neste trabalho. Primeiro, a linguagem suporta a noção de objetos complexos; segundo, apresenta a capacidade para definição de tipos abstratos de dados; e, finalmente, a característica funcional da linguagem permite uma comparação com as idéias oriundas dos modelos de dados semânticos funcionais, como, por exemplo, FDM (KERSCHBERG 1976) e IFO (ABITEBOUL 1987), discutido no ítem 2.2.3.

O modelo computacional funcional foi escolhido por permitir o compartilhamento de objetos, uma vez que um mesmo objeto pode ser mapeado por diversas funções, e, também, por ser adequado para explorar a capacidade de processamento paralelo. Além disso, a simplicidade e elegância da linguagem podem facilitar o sucesso do modelo.

Objetos em FAD são construídos a partir de tipos de dados atômicos. Um tipo atômico é, na verdade, um tipo abstrato de dado. A implementação da estrutura do tipo e das operações sobre o mesmo são definidas pelo usuário em qualquer linguagem de programação convencional, enquanto a declaração do tipo e das operações é feita na própria linguagem FAD. Esta solução permite a estensibilidade dos tipos atômicos e a execução eficiente em uma máquina dedicada, pois a linguagem de banco de dados está "em cima" da implementação das operações, podendo estas serem armazenadas no próprio banco de dados. Outras considerações sobre extensões de um SGBD "do lado" ao invés de "embaixo", como é usual, são apreciadas em (SCHWARZ 1986).

Três conjuntos especiais são reconhecidos pelo sistema:

- O conjunto A de tipos atômicos;
- O conjunto N de nomes de atributos;
- O conjunto I de identificadores de objetos.

A partir daí, podemos definir um objeto complexo O como sendo uma tripla (identificador, tipo, valor) onde:

- O identificador pertence ao conjunto I;
- O tipo pertence a união de A com {null_type, set, tuple};
- O valor depende do tipo, podendo ser:
  . null_type - sem valor;
  . tipo atômico - o valor é um elemento do domínio definido pelo usuário;
  . set - o valor é {i1, i2, ..., in} onde os ij's são identificadores distintos de I;
  . tupla - o valor é [a1:i1, a2:i2, ..., an:in] onde os aj's são nomes de atributos distintos pertencentes a N e os ij's são identificadores, não necessariamente distintos de I.

Um objeto complexo pode ser representado por um grafo. Cada (sub)objeto é representado por um nó. Um nó do tipo atômico é representado por seu valor, enquanto um asterisco representa um set e um losângo uma tupla. Arcos ligam os nós, devendo ser rotulados com o nome do atributo quando se tratar de um arco que parte de um nó tupla.

Se o grafo for acíclico, é possível fazer uma descrição linear usando chaves para denotar conjuntos e colchetes para tuplas.

A figura 2.14 mostra a representação gráfica do objeto DFD, cuja representação linear é a seguinte:

```
PROCESSOS  =
     [p_no:          integer,
      p_descrição:   string,
      p_super:       integer,
      fluxos:        {[fnome:string, dados:{DADOS}]},
      dep_dados:     {[nome_dep:string, dados:{DADOS}]} ]

DADOS =
     [nome_dado:string, tipo_dado:string]
```

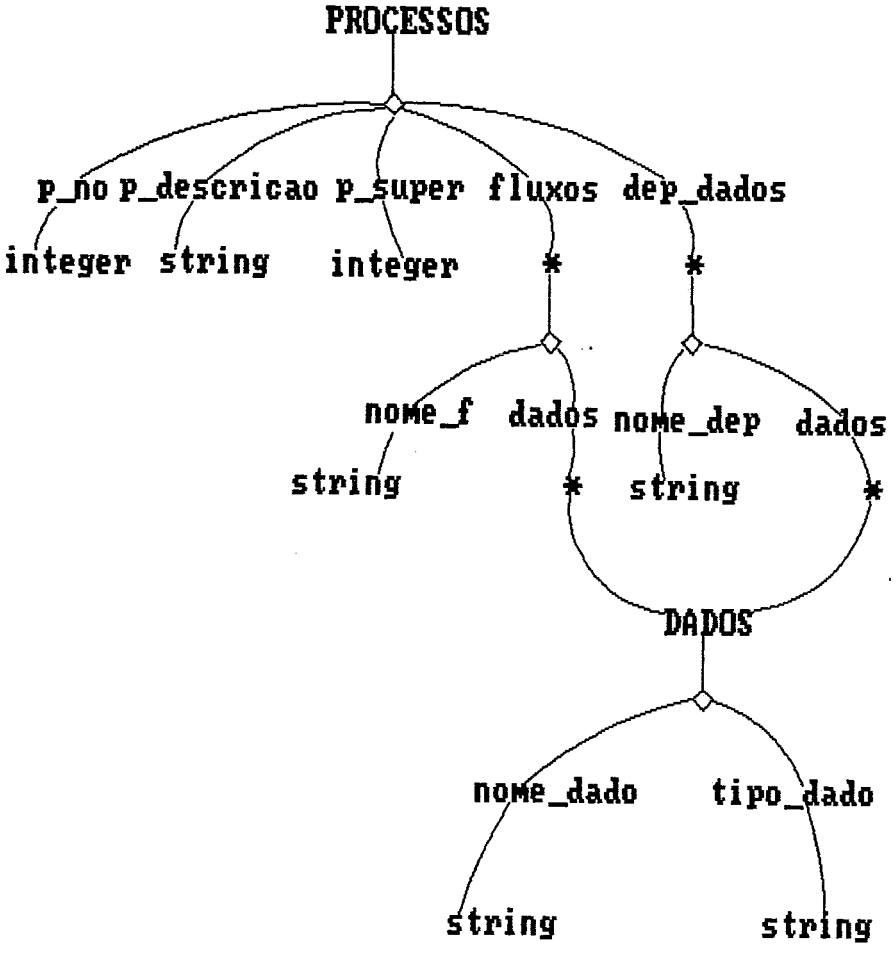


Figura 2.14 - Representação do objeto DFD em FAD.

Atualmente, o sistema está sendo estendido para suportar tipos recursivos.

Programas em FAD são operadores ou predicados. No primeiro caso, um identificador é retornado; no segundo é retornado uma palavra-chave true ou false. FAD é quase puramente uma linguagem funcional. Modificações na base de

dados são vistas como efeitos colaterais da aplicação de funções.

Alguns operadores e predicados básicos são fornecidos pelo sistema, mas a maioria deve ser implementada como funções escritas ("User Defined Functions") para os tipos abstratos de dados.

Os operadores básicos incluem funções para recuperar um valor dentro de uma tupla e para manipulação básica de conjuntos (união, interseção e diferença). Os operadores básicos para alteração da base incluem atribuição de tuplas e eliminação/inclusão de elementos em conjuntos. Alguns operadores básicos para processamento paralelo são também fornecidos.

Embora a linguagem FAD permita a modelagem de objetos complexos e o conceito de TADs, qualquer operação para manipular um objeto complexo deve ser implementada pelo usuário. O modelo funcional também permite o tratamento elegante de objetos recursivos, embora estes ainda não sejam suportados. O sistema também não suporta nenhuma restrição de integridade.

II.2.6) O Sistema ORION.

ORION é um protótipo de um sistema de gerência de banco de dados desenvolvido na Universidade do Texas (BANERJEE 1987a, b, KIM 1987). O seu modelo de dados incorpora conceitos oriundos da área de linguagens de programação orientadas a objetos tais como: classes, métodos e hierarquia de classes com herança de propriedades. Atenção especial foi dispensada no intuito de permitir facilidades adicionais para a evolução do esquema (isto é, a capacidade de modificar, dinamicamente, a definição de uma classe ou da estrutura entre classes), controle de versões de objetos (variações históricas de um mesmo objeto) e criação e manipulação de objetos compostos.

A definição de objetos compostos, no ORION, vem da necessidade de definir e manipular um conjunto de objetos como uma única entidade, e do reconhecimento da falta desta capacidade nos modelos de dados orientados a objetos mais tradicionais.

Um objeto composto - no ORION - é um objeto com uma hierarquia de sub-objetos componentes (chamados objetos dependentes) exclusivos, ou seja, um sub-objeto só pode ser possuído por um único objeto pai.

O modelo de dados distingue, claramente, dois tipos de hierarquia de classes. O primeiro tipo - "hierarquia de objetos compostos" - constitui o esquema a partir do qual um objeto composto é instanciado. Este tipo de hierarquia captura a semântica de um relacionamento do tipo "is-part-of". O segundo tipo de hierarquia, chamado simplesmente de "hierarquia de classes", representa um relacionamento do tipo "is-a", comum em modelos orientados a objetos. Um relacionamento do tipo "is-a", entre duas classes, cria o conceito de subclasse e superclasse, onde a subclasse herda propriedades (variáveis de instância e mensagens) da sua superclasse. Em particular, no ORION, uma (sub)classe pode ter mais de uma superclasse.

Sob o aspecto estrutural, um objeto composto no ORION é constituído de um objeto raiz conectado a um ou mais objetos dependentes. Um objeto dependente pode ser: um objeto atômico, uma nova raiz (permitindo indefinidos níveis de composição) ou um conjunto de objetos dependentes.

A BNF abaixo define um objeto composto no ORION. O símbolo * denota zero ou mais ocorrências.

```
<Objeto_composto> :==
     <Raiz_de_Objeto_Composto> (<Dependente_ligado>*),

<Dependente_ligado> :==
     <Variavel_de_instância> <Objeto_dependente>,

<Objeto_dependente> :==
     <Objeto_atômico>
     | <Raiz_de_Objeto_dependente> ( <Dependente_ligado>* )
     | { <Objeto_dependente>* }
```

A definição de objeto composto impõe uma hierarquia estrita na estrutura deste, materializada por variáveis de instância, especialmente chamadas de variáveis de instância de composição. Não é permitido sub-objetos compartilhados nem ciclos (um subobjeto ser composto por si próprio) diretos ou indiretos. Uma forma de superar esta limitação é o fato de um objeto poder se referenciar a qualquer outro objeto (dependente ou não), através de uma variável de instância comum. Tal rede de objetos criada é conhecida no ORION como um objeto agregado, mas o sistema, infelizmente, não reconhece, explicitamente, este tipo de estrutura, não havendo nenhuma forma de declarar, manipular ou garantir a integridade de tais objetos.

O esquema de um objeto composto é criado através da declaração de variáveis de instância de composição cujos domínios são classes de sub-objetos componentes. Estas variáveis são declaradas na classe raiz da composição e em cada classe raiz de objeto dependente.

Se existe um relacionamento de composição entre uma classe A (raiz) para uma classe B (dependente), então existe uma variável de instância de composição V em A, cujo domínio é o conjunto de instâncias da classe B. O sistema garante que somente uma instância de um objeto da classe A referencia uma determinada instância de um objeto da classe B. Esta restrição de integridade é garantida, obrigando-se que um valor não nulo só possa ser atribuído a uma variável de instância de composição, na hora da criação deste valor, ou seja: um objeto dependente "nasce" ligado ao objeto pai. Um objeto dependente não pode ter vida independente: se o seu pai é eliminado, o sistema elimina em cascata o(s) objeto(s) dependente(s). Para relaxar tal restrição tão severa, um subobjeto dependente pode trocar de pai, através de uma mensagem especial "ExchangePart".

Um conceito bastante interessante, suportado pelo ORION, é o fato de variáveis de instância de composição serem herdadas por subclasses em "hierarquias de classes" (do tipo "is-a") da mesma forma que qualquer variável, isto é, se uma variável de instância V de uma classe A faz o papel de variável de instância de composição, então, qualquer subclasse A' de A herda a variável V com a propriedade de composição. Isto significa que um subtipo de uma classe, que é a raiz de um objeto composto, é ,também, um objeto composto.

Por exemplo: o objeto composto DFD da figura 2.15 pode ser uma superclasse de uma subclasse de nome DARTS. DARTS (Design Approach for Real Time Systems) (GOMA 1984) é um método de projeto de software para sistemas de tempo real. DARTS possui muito do método de análise estruturada, acrescido de primitivas para definição de tarefas, troca de mensagem e sincronização, necessárias a sistemas desta categoria. Um objeto de nome DARTS pode ser modelado como uma especialização (que considera estas primitivas) do objeto DFD (Figura 2.16) (o objeto de nome DARTS representa um diagrama de fluxo de dados acrescido de notação para as primitivas para tempo real). Em outras palavras, as

variáveis de instância de composição que materializam a composição do objeto DARTS são herdadas da classe DFD.

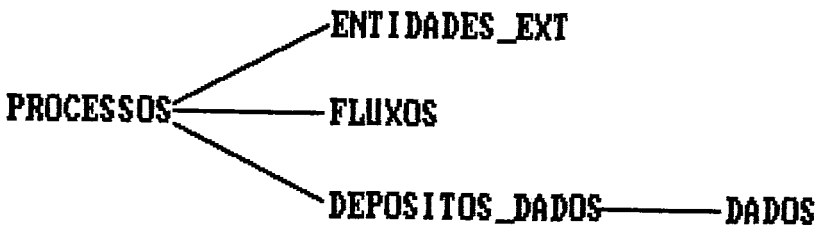


Figura 2.15 - Representação gráfica do objeto DFD no ORION.

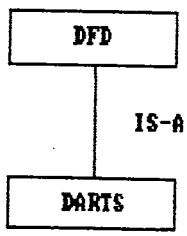


Figura 2.16 - Generalização no ORION.

Outra capacidade interessante é a de propagação de valores. Propagação de valores significa que o valor de uma variável de instância é automaticamente propagado para uma variável de instância de mesmo nome de um objeto dependente. Esta capacidade pode ser vista como o compartilhamento de um valor ao longo do objeto composto, não devendo ser confundido com o conceito de herança de variáveis entre classes. O primeiro caso trata do compartilhamento de valores entre os componentes de uma instância de um objeto composto, O segundo caso permite uma abstração do tipo generalização/especialização, onde os nomes de variáveis é que são herdados. Ao manipular um

objeto dependente, a propagação de valores evita ter que se visitar os ancestrais do objeto, para recuperação de atributos, que pertencem ao objeto como um todo, e que se encontram representados na classe raiz.

O uso de objetos compostos no ORION representa uma preocupação em modelar um tipo de relacionamento (composição) que a maioria dos modelos orientados a objetos não reconhece. Não exitem, porém, operadores pré-definidos para a manipulação de objetos compostos, devendo ser construídos dentro da filosofia geral de um modelo orientado a objetos, através de métodos programados pelo usuário. As restrições de integridade suportadas são aquelas que dizem respeito aos valores que podem ser atribuídos às variáveis de instância de composição. Um objeto dependente não tem vida própria e a estrutura representável não permite objetos "mais" complexos (recursão e sub-objetos compartilhados). Tais objetos podem ser implementados, mas o sistema não os reconhece.

II.2.7) A linguagem LOOPS.

LOOPS (BOBROW 1981, STEFIK & BOBROW 1986) é uma linguagem de programação orientada a objetos baseada no LISP que, ao contrário da maioria dos sistemas ditos orientados a objetos, permite a definição de objetos compostos. Embora LOOPS não seja exatamente um sistema de gerência de banco de dados, as idéias para tratamento de objetos complexos são perfeitamente válidas e aplicáveis a área de SGBDOO.

Um objeto composto é um objeto que contém outros objetos como partes componentes. A definição do objeto composto é feita por uma especificação de uma classe (que chamaremos de raiz, embora LOOPS não use esta nomenclatura) que descreve os sub-objetos componentes, as classes a que estes pertencem e a sua forma de interconexão, também materializada através de variáveis de instância. O objeto DFD poderia ser declarado, aproximadamente assim:

```
DFD
MetaClass Class
Supers (CompositeObject,....)
Class Variable
     Metodo "analise estruturada"
Instance Variables
     Nome NIL
     Processos NIL
          part (PROCESSOS)
     Entidades_ext NIL
          part (ENTIDADES_EXTERNAS)
     ..........
```

Instanciar um objeto composto significa instanciar a classe raiz e todas as classes componentes ao mesmo tempo. O processo de instanciação é recursivo, permitindo que um objeto composto seja parte de um outro objeto composto, porém, por default, um erro ocorrerá se, durante o processo, a descrição do objeto composto ordenar direta, ou

indiretamente, a criação de outra instância da mesma classe sendo instanciada. O problema ocorre, pois o sistema tenta instanciar o objeto composto todo, no momento da instanciação da raiz. E'possível, porém, ordenar a instanciação de sub-objetos sob demanda.

Pela mesma razão citada acima, um subobjeto componente deve ser somente uma (1) instância (e não um conjunto de instâncias) de uma classe componente. Para modelar conjuntos de objetos componentes, como é possível no ORION (vide 2.2.6), é necessário que a classe componente produza instâncias que são, em si mesmas, conjuntos, de uma maneira similar ao conceito de abstrações de classes do SDM.

Por outro lado, é permitido que um subobjeto seja referenciado por mais de um objeto, permitindo o compartilhamento de sub-objetos.

O processo de instanciação de um objeto composto pode ser visto como a criação das estruturas de dados que o representam, sendo preenchidos, automaticamente, os valores das variáveis de instância que materializam as conexões entre os objetos.

Como no ORION, uma subclasse, (relacionamento "is-a") de uma classe que é um objeto composto, é também um objeto composto, podendo este ser especializado, acrescentando-se ou redefinindo-se as partes componentes.

A implementação do processo de instanciação aproveita-se da própria elegância do modelo orientado a objetos: toda classe de objeto composto é uma subclasse (relacionamento "is-a") da classe especial CompositeObject, que contém os métodos para a manipulação (instanciação) da descrição das partes do objeto. Como LOOPS suporta herança múltipla, uma classe de objeto composto pode ser também subclasse de outras superclasses.

O sistema não prevê nenhum tipo de restrição de integridade para objetos compostos, e toda a manipulação do objeto deve ser programada através de métodos (exceto, como vimos, o processo de instanciação). A classe CompositeObject pode ter novos métodos programados para este fim.

LOOPS define, ainda, uma espécie de objeto composto chamado perspectivas. A semântica deste tipo de construção é bem diferente do que se espera de um relacionamento do tipo "is-part-of".

Perspectivas permitem a interpretação de uma entidade conceitual através de diferentes visões. Um objeto pode estar ligado a diversos sub-objetos, mas cada ligação significa uma perspectiva ou visão do objeto raiz. Podemos imaginar que cada subobjeto deste tipo de composição descreve adicionalmente (em termos de estrutura e comportamento) o objeto raiz, criando um relacionamento do tipo "as-a". Por exemplo, o objeto DFD pode estar ligado a outros sub-objetos que descrevem o seu uso como uma ferramenta para documentação, análise ou auditoria de sistemas. (figura 2.17)

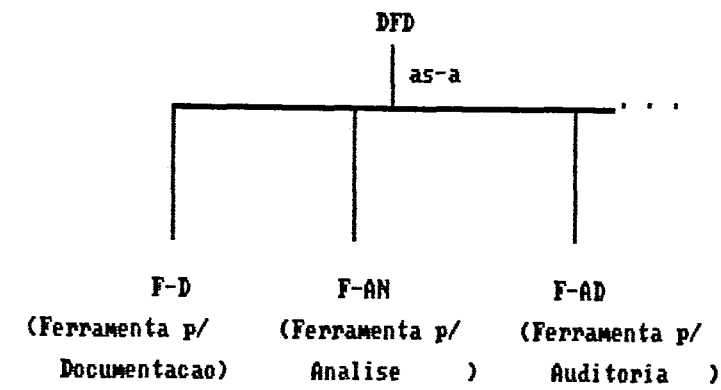


Figura 2.17 - Perspectivas em LOOPS.

Perspectivas diferentes podem ter variáveis de instância com o mesmo nome, pois serão mantidas independentemente. Ao contrário do objeto composto, perspectivas são instanciadas sob demanda, o que significa que uma determinada instância, de uma classe que contém perspectivas, pode, em um determinado instante, possuir somente algumas (instâncias de) perspectivas, e não todas

declaradas no esquema, economizando espaço de armazenamento e velocidade na recuperação do objeto.

Representar perspectivas no modelo relacional de uma forma normalizada, significa criar diversas tabelas com os atributos (variáveis de instância) inerentes a cada perspectiva. Para recuperar o objeto, devemos arcar com uma operação de junção entre a tabela principal (raiz do objeto) e cada tabela que representar uma perspectiva. Além disto, a integridade do objeto não é mantida, automaticamente, pelo modelo puro (embora algumas implementações de bancos relacionais implementem integridade referencial entre tabelas). A figura 2.18 representa o modelo relacional equivalente à figura 2.17.

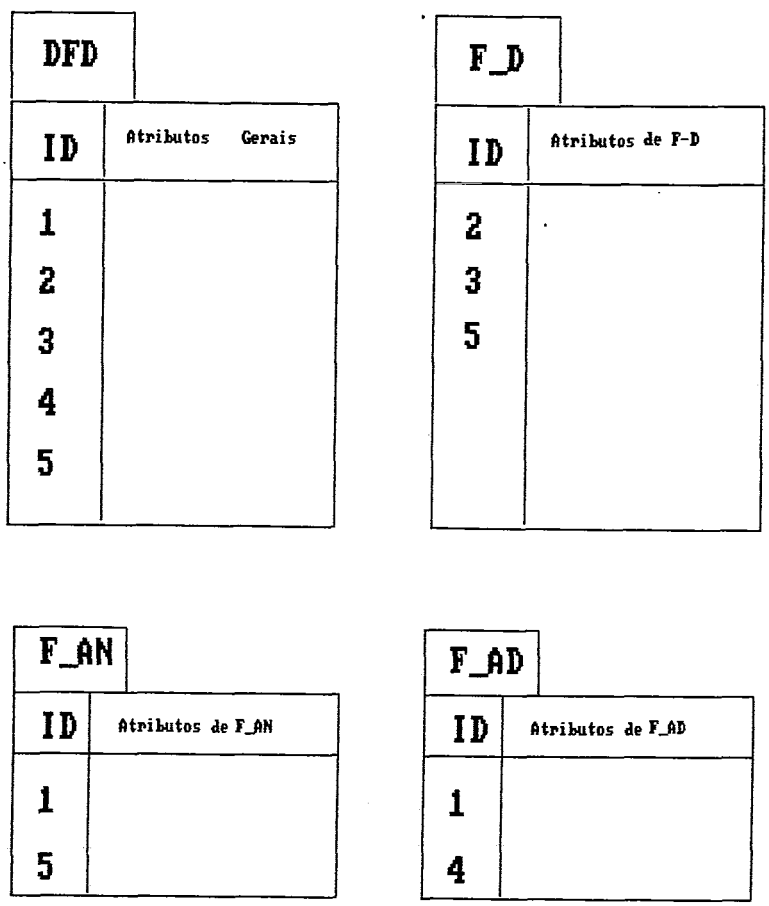


Figura 2.18 - Representação relacional de perspectivas.

Em LOOPS, uma perspectiva de um objeto é acessada, simplesmente, através do nome da perspectiva. Como regra de integridade, não podem existir "objetos perspectiva" para "objetos raiz" que não existam.

II.2.8) O Sistema de Gerência de Banco de Dados POSTGRES

POSTGRES (STONEBREAKER 1986, 1987) é uma extensão do sistema de gerência de banco de dados relacional INGRES (HELD 1975).

Inicialmente foram incluídas facilidades para suporte a tipos abstratos de dados e suporte a procedimentos como objetos do banco de dados. Com estas extensões é possível resolver vários problemas relacionados ao aspecto estrutural de objetos complexos, a saber:

- objetos complexos com muitos níveis de sub-objetos;
- objetos complexos com sub-objetos compartilhados;
- objetos complexos com estrutura imprevista.

Mais tarde (STONEBREAKER 1987), foi incluído, também, um mecanismo de herança, tornando o somatório destas extensões capaz de simular uma variedade de conceitos oriundos dos modelos de dados semânticos, como, por exemplo, generalização.

O autor argumenta que ainda não existe um conjunto mínimo de conceitos que seja de consenso geral, para suportar, genericamente, as aplicações ditas não convencionais. Muitas idéias têm surgido na última década, mas o número e diversidade das propostas é muito grande para permitir fatorá-las. Mais ainda, muitas vezes o que é adequado para uma proposta, não o é para outra.

Desta forma, o modelo relacional ainda provê uma base sólida, tanto teórica quanto prática, onde podem ser simuladas as diversas propostas. Outros argumentos a favor do modelo relacional são a sua simplicidade e a necessidade de compatibilidade, pois as aplicações não convencionais certamente conviverão em um ambiente com dados mais "bem comportados".

POSTGRES suporta um conjunto de tipos de dados atômicos a partir do qual um atributo de uma relação pode ser definido. Todos estes tipos, inclusive alguns mais tradicionais e pré-definidos como: inteiro, real, cadeia de caracteres e outros, são considerados TAD's.

Um TAD é especificado através da definição do seu nome, tamanho da representação interna, valor default e rotinas para conversão de sua representação interna para externa e vice-versa.

Vejamos um exemplo. Suponhamos o objeto DFD que é composto das relações que representam PROCESSOS, DEPÓSITOS_DE_DADOS, FLUXOS, ENTIDADES_EXTERNAS e DADOS. Um atributo de PROCESSOS pode ser a sua posição relativa a um eixo de coordenadas x-y, informando o local de sua representação gráfica na tela, em uma ferramenta gráfica para projeto de sistemas. A partir deste ponto é possível determinar a área ocupada pelo desenho, uma vez que todas as representações gráficas de processos têm o mesmo tamanho. Podemos definir um TAD específico para esta posição com a sintaxe abaixo:

```
define type posição_xy is
     (InternalLength = 8,
      InputProc = CharToPos,
      OutputProc = PosToChar,
      Default = "")
```

A representação externa de uma posição x = 10 e y = 20 é "10,20". CharToPos e PosToChar são procedimentos, escritos pelo usuário, para converter entre esta representação e a representação interna.

A relação PROCESSOS com três atributos, é definida com a sintaxe:

```
create PROCESSOS (p_no      = int4,
                  descrição = char10,
                  posição   = posição_xy)
```

Imaginemos uma interface gráfica para manipular um DFD, como visto na figura 2.19. As letras minúsculas a e b representam as coordenadas armazenadas no atributo posição da relação PROCESSOS, enquanto a letra c representa a posição do cursor em um determinado instante, por exemplo x = 25, y = 40.

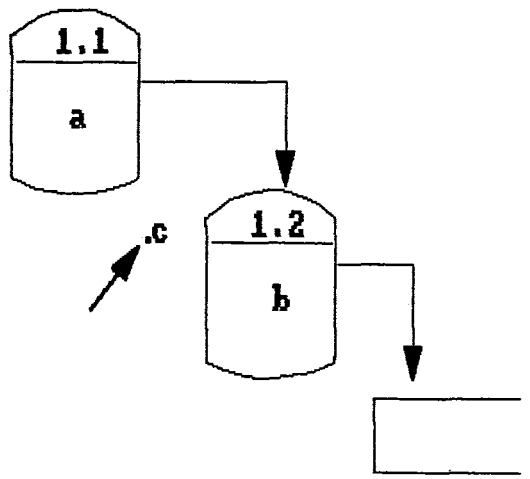


Figura 2.19 - Exemplo de uso de TAD's em POSTGRES.

Podemos definir um operador de interseção de representação de processos. Dadas duas coordenadas, o operador calcula se as representações de processos gerados a partir delas se sobrepõem.

```
define operator int_Proc (posição_xy, posição_xy)
                          returns bool is
     (Proc = Interseção_Processos,
     Precedence = 3,
     Negator = Not_int_Proc)
```

A seguinte consulta recupera da base todos os números de processos que se sobrepõem a um processo a ser desenhado na posição c.

```
retrieve (PROCESSOS.p_no)
where PROCESSOS.posição int_Proc "25,40"
```

Na definição de um operador, o sistema POSTGRES ainda permite a definição da precedência e associatividade do operador. Informações adicionais podem ser fornecidas para ajudar ao otimizador de consultas, como, por exemplo, a informação de que o operador implementado por Not_Int_Proc é a negação do operador Int_Proc, conhecimento que pode ser usado para resolver predicados do tipo not (expressão Int_Proc expressão).

A proposta de TAD's do POSTGRES não permite que o código do procedimento que implementa o TAD acesse outros atributos na mesma relação ou em relações diferentes.

Para tal, um novo tipo de dado, chamado procedimento, é reconhecido pelo sistema. Este procedimento é um sequência de comandos em POSTQUEL (linguagem de manipulação do POSTGRES, uma extensão da linguagem QUEL) que retorna uma ou mais tuplas de uma ou mais relações. Assim, o valor de um atributo do tipo procedimento é uma espécie de relação que contém tuplas de diferentes relações, conhecida como multirelação.

Existem duas formas de usar o tipo procedimento.

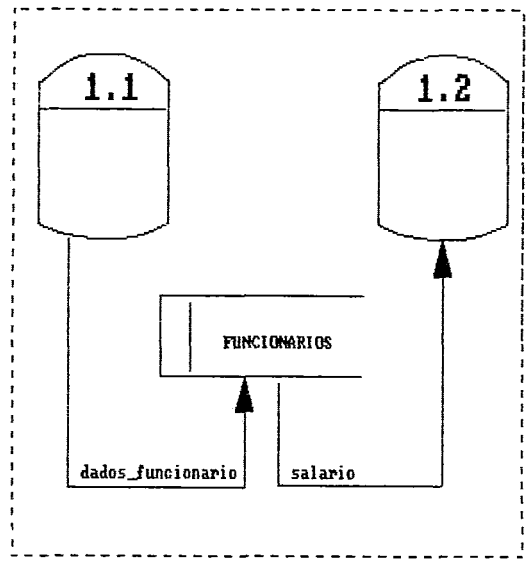
Na primeira forma, conhecida como variável, o atributo armazena, realmente, os comandos em POSTQUEL. Isto significa que tuplas diferentes de uma relação podem ter, em um atributo tipo procedimento, diferentes consultas em POSTQUEL. Isto permite uma flexibilidade enorme, podendo ser usado para implementar objetos complexos com estrutura não prevista, definida em tempo de execução.

Na segunda forma, conhecida como parametrizável, os comandos POSTQUEL não ficam armazenados no atributo e, sim, em outro lugar. Assim, todas as tuplas de uma relação tem o mesma consulta POSTQUEL em um determinado atributo procedimento. O que fica armazenado no atributo é um valor usado como parâmetro para o procedimento parametrizável.

Vejamos alguns exemplos.

A relação FLUXOS do objeto DFD representa os fluxos de dados de um DFD. Um fluxo, além de ser composto por ítens de dados, é uma entidade, em um diagrama DFD, que tem uma origem e um destino. Tanto a origem quanto o destino podem ser PROCESSOS, DEPÓSITOS_DE_DADOS, ou ENTIDADES_EXTERNAS. Dois atributos (de nomes "de" e "para"), do tipo procedimento variável, podem ser usados para ligar o objeto fluxo aos seus objetos de origem e destino. No modelo relacional, os atributos "de" e "para" não podem ser fielmente representados, pois em cada tupla, o domínio do atributo pode ser diferente, dependendo do objeto (PROCESSOS, DEPÓSITOS_DE_DADOS ou ENTIDADES_EXTERNAS) que é apontado. Integridade referencial é impossível de ser mantida.

A figura 2.20 mostra o conteúdo da tabela FLUXOS. O domínio do atributo "de" para a primeira tupla é um conjunto de tuplas da tabela PROCESSOS, enquanto para a segunda tupla, é um conjunto de tuplas na tabela DEPÓSITOS_DE_DADOS.



PROCESSOS		
p_no	descricao	. . .

FLUXOS		
NOME_F	DE	PARA
dados_ _funcionario	retrieve (PROCESSOS.ALL) where p_no = 1.1	retrieve (DEPOSITOS.ALL) where dep_nome="FUNCIONARIOS"
. . .	. . .	. . .
salario	retrieve (DEPOSITOS.ALL) where dep_nome="FUNCIONARIOS"	retrieve (PROCESSOS.ALL) where p_no = 1.2

Figura 2.20 - Procedimentos variáveis em POSTGRES.

É possível recuperar, automaticamente, atributos de tuplas pertencentes ao "valor" do atributo procedimento, usando-se a notação de pontos embutidos. A consulta abaixo recupera o nome do fluxo e o atributo descrição da relação PROCESSOS, através da execução automática de uma junção implícita.

```
retrieve (F.nome_f, F.de.descrição)
from F in FLUXOS
```

A partir deste exemplo, fica evidenciado o poder do tipo procedimentos variáveis, bem como a possibilidade de usá-lo para representar, também, objetos com estrutura imprevista.

Já um procedimento parametrizável pode ser usado para implementar o relacionamento N:M entre FLUXOS e DADOS. No modelo relacional, devemos criar uma relação FLUXOS_DADOS para representar este relacionamento.

```
create FLUXOS_DADOS
       (nome_fluxo = char25,
       nome_dado  = char25)
```

Um TAD abaixo cria um tipo procedimento parametrizável onde o símbolo $ funciona como uma variável de tupla ligada à tupla corrente, isto é, o parâmetro $nome_f deve ser substituído pelo valor do atributo nome_f da tupla corrente.

```
define type LISTA_DADOS is
     retrieve FD.nome_dado
     from FD in FLUXOS_DADOS
     where FD.nome_fluxo = $nome_f
```

A definição completa da relação FLUXOS, fica:

```
create FLUXOS
       (nome_f = char25,
        de     = postquel,
        para   = postquel,
        dados  = lista_dados).
```

A consulta abaixo recupera todos os ítens de dados que compõem o fluxo "dados_funcionário":

```
retrieve (f.nome_f, f.dados.nome_dado)
from     f in FLUXOS
where    f.nome_f = "dados_funcionário"
```

Os operadores de TAD's são implementados através de procedimentos, escritos pelo usuário, em qualquer linguagem de programação convencional, conhecidos como "Procedimentos Definidos pelo Usuário" (PDU's). Por exemplo, o procedimento Interseção_Processos que implementa o operador Int_Proc no exemplo anterior é um PDU que deve ser declarado.

O mais interessante em relação a PDU's é a capacidade destes tomarem uma tupla inteira como argumento. Desta maneira, é possível definir um procedimento que pode ser aplicado sobre uma tupla (e não sobre um atributo, como na proposta de TAD's), chegando-se mais próximo do encapsulamento de métodos em modelos orientados a objetos.

Por exemplo, uma relação poderia ser vista como uma classe, tuplas seriam instâncias de objetos e PDU's seriam métodos.

Além disto, em (STONEBREAKER 1987) o POSTGRES foi estendido para permitir criar hierarquias do tipo "is-a" entre relações, criando-se o conceito de subtipo e supertipo, onde há herança de atributos do supertipo para o subtipo. Um PDU definido sobre uma certa relação pode também ser aplicado sobre qualquer relação que seja seu subtipo, chegando-se a um mecanismo de herança de procedimentos.

Se um procedimento é aplicado sobre os atributos de uma tupla, e se um destes é do tipo procedimento em POSTQUEL, é possível, então, disparar outros procedimentos sobre outras relações. Comparando-se com as propostas dos modelos orientados a objetos, isto pode ser visto como um método que pode mudar o estado (valores dos atributos) de um objeto (tupla) e enviar mensagens para outros objetos, completando-se assim a equivalência do POSTGRES com diversas propostas de sistemas de gerência de banco de dados orientados a objetos.

De fato, (ROWE 1986) descreve a implementação de uma hierarquia de objetos compartilhados que tem conceitos tipicamente oriundos dos SGBDOO's, usando, como base, o POSTGRES.

Outra capacidade da POSTQUEL, útil para manipulação de objetos, é o comando que calcula o fecho transitivo, necessário para lidar com objetos recursivos.

Por exemplo, a relação PROCESSOS pode ter o atributo proc_super que indica o processo pai que originou, na sua explosão, o processo corrente. A definição da tabela (simplificada) é:

```
create PROCESSOS (p_no = int4,
                  descrição = char25,
                  proc_super = int4)
```

A consulta abaixo cria a relação COMPOSIÇÃO que contém todos os processos que compõem o processo de p_no = 1.1.

```
retrieve* into COMPOSIÇÃO (P.p_no, P.proc_super)
from P in PROCESSOS, C in COMPOSIÇÃO
where P.p_no = 1.1 or
      P.proc_super = C.p_no
```

A notação * faz com que a consulta seja executada recursivamente até que a relação COMPOSIÇÃO não seja mais acrescida de novas tuplas.

O SGBD POSTGRES é, sem dúvida, uma proposta bastante diferente das outras encontradas na literatura. A forma de "declarar" um objeto complexo não é elegante, fazendo com que se tenha que procurar, nas diversas relações, atributos do tipo procedimento que realizam a ligação entre tuplas. Porém, o poder de simulação de diversos problemas encontrados em modelagem de dados é muito grande. O uso de identificadores internos (não abordado até aqui) permite assegurar diversas formas de integridade. Um alvo procurado com muito afinco na implementação do sistema é a possibilidade de pré-computar os procedimentos definidos pelo usuário, de forma a obter desempenho aceitável.

II.2.9) Comparação das Propostas.

A seguir apresentamos uma comparação das propostas discutidas ao longo da seção 2.2, usando a tabela abaixo.

Um S nesta tabela significa que o sistema apresenta uma determinada característica, um N significa que não, ou a característica não foi encontrada na literatura. Um hífem aparece onde o conceito não se aplica adequadamente à proposta, e, finalmente, letras minúsculas são usadas onde um comentário adicional é necessário.

Alguma subjetividade foi necessária para a construção da mesma, pois os critérios de comparação (colunas) não são necessariamente homogêneos entre as propostas, isto é, na literatura pesquisada, algumas características de uma determinada proposta podem não ter sido apontadas explicitamente, bem como o enfoque do autor pode não ser o mesmo que o utilizado para a comparação. Assim, é possível discordar dos resultados apresentados, bem como dos próprios critérios de comparação.

A tabela também não é completa; de qualquer forma, foi bastante útil para a determinação dos requisitos de funcionalidade para sistemas de gerência de banco de dados para aplicações não convencionais, apresentados no capítulo 3.

A tabela inclui uma classificação do COPPEREL de acordo com as propostas de extensão discutidas nos capítulos 5 e 6. Alguma nomenclatura dos critérios de comparação pode ser encontrada na seção 3.1.

Tabela Comparativa das Propostas Discutidas

	OBJETOS COMPLEXOS Definição								
	ct	cc	rec	sim	relat	gran	he	vers	cl
Syst. R	S	S	N	N	N	N	S	c	S
PRIMA	S	S	S	S	N	N	N	N	N
IFO	S	S	N	b	N	N	N	N	N
SAM*	S	S	S	b	S	N	N	S	N
FAD	S	S	N	b	N	N	N	N	N
ORION	S	S	N	N	N	N	S	S	N
LOOPS	S	a	N	N	N	N	N	N	N
POSTGRES	S	S	S	N	N	N	N	d	k
COPPEREL	S	S	S	N	N	N	N	N	S

	OBJETOS COMPLEXOS Consulta		Rest. Int.		TAD's atr	reg	obj	COMP	IS-A
	an	nav	i	e					
Syst. R	S	N	S	S	N	N	N	N	N
PRIMA	S	N	S	S	N	N	N	N	N
IFO	e	e	h	h	N	–	N	N	S
SAM*	e	e	S	S	j	–	N	N	S
FAD	S	S	I	I	S	–	N	N	N
ORION	N	N	S	S	–	–	S	N	S
LOOPS	N	N	–	–	–	–	S	N	S
POSTGRES	f	f	N	N	S	S	N	N	S
COPPEREL	S	g	S	S	N	N	N	N	N

LEGENDA:

ct : construtor de tupla
cc : construtor de conjunto
rec : recursividade
sim : simetria no tratamento do objeto composto
relat: relativismo
gran : granularidade
he : grafado um s se o sistema permite somente a definição de uma hierarquia estrita
vers : tratamento de versões
cl : tipo primitivo campo longo
an : operações de manipulação de alto nível (manipulação do objeto como um todo)
nav : operações de navegação no objeto complexo
i : restrição de integridade de inclusão (um sub-objeto deve pertencer a um objeto já existente
e : Restrição de integridade de eliminação (eliminação em cascata)
atr : definição de TAD em atributo
reg : definição de TAD em registro
obj : definição de TAD no objeto
COMP : comportamento
IS-A : modelagem de generalização/especialização

NOTAS:

a - É possível simular o construtor de conjunto.
b - A manipulação por linguagens funcionais pode ser considerada como uma forma de possuir alguma capacidade de simetria.
c - É sugerido o uso de versões usando o mecanismo de objetos complexos.
d - O que é chamado de versões é, na realidade, uma tabela cópia de outra, que pode ser automaticamente atualizada, à medida que a tabela original também o for.
e - O modelo não especifica linguagem de manipulação.
f - Operações de alto nível podem ser obtidas através das operações relacionais já existentes. A notação de ponto permite uma certa navegação.
g - É possivel simular alguma navegação (5.3.2).
h - Não explicitado pelo modelo.
i - Um sistema de objetos é considerado consistente se não houver ponteiros pendentes.
j - Sugerido pelo modelo.
k - É permitido arrays de tamanho indefinido.

II.3) Controles Operacionais.

Nesta seção, analisamos os controles operacionais, que se fazem necessários para SGBD's que desejem suportar aplicações não convencionais. O controle de versões foi incluído, aqui, por entendermos que esta capacidade está mais próxima de um controle operacional, como backup, recuperação e outros, do que de um requisito de modelagem.

II.3.1) Armazenamento e Métodos de Acesso.

Uma vez que SGBD's para aplicações não convencionais devem ser mais orientados a objetos do que a registros, os tradicionais métodos de acesso e estratégias de armazenamento devem ser adaptados.

Duas soluções são usadas para melhorar o acesso a objetos complexos:

a) O grupamento físico do objeto com seus sub-objetos no banco de dados;

b) O uso de estruturas de dados que permitam acesso imediato a identificadores (ponteiros) dos objetos componentes.

Quando o objeto complexo é uma hierarquia estrita de sub-objetos e não ocorre compartilhamento de sub-objetos, o grupamento é simples de ser resolvido. Caso contrário, o grupamento é feito em favor de uma visão e em prejuízo de outras. A maioria dos sistemas realiza um grupamento estático, isto é, uma vez que um sub-objeto X foi armazenado perto de Y, aí ficará, mesmo que passe a pertencer a um outro objeto Z. Alguns sistemas, como o Object Server do ENCORE (HORNICK 1987) e o DASDBS (DEPPISCH 1986), permitem introduzir redundância controlada, replicando os objetos que devem ser agrupados de mais de uma forma. O sistema garante a atualização automática de todas as cópias,

se apenas uma for modificada. Se as aplicações forem, em sua maioria, de leitura-somente, o desempenho é bastante melhorado.

Algumas estruturas de dados podem ser criadas para agilizar o processo de recuperação de um objeto complexo. Um índice de junção (VALDURIEZ 1987) é um índice que contém entradas para todos os pares de tuplas que satisfazem um certo predicado. Uma consulta imediata ao índice permite recuperar todos os componentes de um objeto complexo.

O System R utiliza listas duplamente encadeadas para implementar objetos complexos, auxiliado por uma estrutura de mapeamento que contém os endereços de todos os registros componentes do objeto complexo. Uma vez que o objeto é acessado, o mapeamento vai para a memória, evitando ter que percorrer listas. Esta estrutura é conhecida pelo otimizador de consultas.

O DASDBS (SCHOLL 1987) utiliza, como primitiva de armazenamento, relações embutidas ou NF2 (Non-first-normal-form). Isto é possível porque hierarquias são as estruturas mais gerais que podem ser linearizadas sem introduzir redundância ou referências. O interessante da proposta é o fato que, tanto os componentes de um objeto complexo podem ser armazenados diretamente, usando as relações não normalizadas, como é possível criar índices de junção implementados sobre relações não normalizadas. Neste caso, a diferença em relação à proposta tradicional de índices de junção é o fato que eles próprios podem ser eficientemente armazenados e recuperados, combinando, assim, as soluções gerais propostas em a e b.

Sob o ponto de vista de métodos de acessos, a extensibilidade dos sistemas (vide 2.1.4) tem sido apontada como solução mais plausível. Os próprios usuários devem poder ser capazes de acoplar métodos de acesso adequados à aplicação ou ao objeto.

II.3.2) Transações e Acesso Concorrente.

Em um banco de dados convencional, uma transação é um conjunto de operações que o sistema garante que é realizado como uma unidade atômica, isto é, se houver algum problema durante a execução de uma transação, as operações realizadas até aquele ponto são logicamente desfeitas. Do mesmo modo, se uma transação chega ao seu fim e é efetivada, qualquer problema que ocorra no sistema, em qualquer tempo depois da efetivação da transação, não deve causar a perda das operações, devendo haver, para isto, mecanismos de recuperação.

Transações são unidades atômicas nas quais o usuário pode confiar; assim, estão fortemente ligadas aos problemas de acesso concorrente. Embora possam ocorrer duas (ou mais) transações simultaneamente, o efeito total é como se estas ocorressem serialmente.

O mecanismo mais usado para garantir acesso concorrente é o uso de bloqueios. Unidades (registos, objetos) "tocados" (lidos ou gravados) por uma transação permanecem bloqueados até o fim da mesma, para garantir a consistência do conjunto de operações.

Uma transação A, que porventura acesse unidades já tocadas por outra transação - digamos B - permanece bloqueada (ou não, dependendo da compatibilidade das operações) até o fim de B. Isto acontece porque no fim da transação B (seja esta efetivada ou abortada) os bloqueios são liberados, tornando as alterações eventualmente realizadas por B visíveis para A.

Uma transação pode ser abortada por vontade própria, por defeitos no sistema (hardware ou software) ou mesmo por problemas de acesso concorrente (deadlocks). Abortar uma transação significa voltar o banco de dados ao estado existente antes do início da transação.

O modelo descrito é adequado para SGBD's comerciais pois as transações duram poucos segundos (ou até menos de um segundo) e não envolvem interação com o usuário no seu

decorrer . Em aplicações não convencionais, uma transação requer interação com o usuário, podendo levar horas ou dias ,além de poder tocar muitos objetos. Os problemas que isto acarreta são:

1) Outras transações podem ficar indefinidamente bloqueadas;

2) As soluções abortivas adotadas não são satisfatórias. O usuário não gostaria de ver o trabalho de dias sendo desfeito;

3) Os mecanismos de bloqueio, normalmente implementados em memória, devem permitir bloqueios persistentes para perdurar entre sessões;

4) Em relação a objetos complexos:

a) Bloquear cada componente do objeto complexo pode ser muito custoso. Os mecanismos de granularidade de bloqueios (GRAY 1978) devem ser adaptados para objetos complexos;

b) Diversos projetistas devem poder trabalhar em partes diferentes de um objeto complexo ao mesmo tempo;

c) Sub-objetos podem ser compartilhados por mais de um objeto complexo.

Uma parte das soluções na literatura, como (HASKIN 1981), provê operações de CHECK-OUT e CHECK-IN, onde um objeto é copiado para uma área de trabalho (CHECK-OUT) deixando indicações (bloqueios persistentes) no banco de dados principal. Através de transações normais (que não interferem pois são feitas na cópia), o objeto é incrementalmente refinado. Quando o projetista se der por satisfeito, o objeto é devolvido (CHECK-IN) para o banco principal, onde, em poucos segundos, é atualizado.

Outras propostas (FISHMAN 1987) consideram o casamento de versões com acesso concorrente, pois, ao contrário de SGBD's convencionais, deve ser considerado a existência de diversos estados válidos do banco de dados.

(HORNICK 1987) propõe uso de blocos de construção onde o usuário (especializado) pode definir um modelo de transações que se adeque a uma determinada aplicação. Diversos tipos de bloqueios e formas de notificação (usando o conceito de notificar um bloqueio ao invés de esperar por ele) são fornecidos para este fim.

(BANERJEE 1987b) estuda o problema de bloqueios em objetos compostos no ORION, introduzindo novos tipos de bloqueios.

Outra solução, que parece ser de consenso, é o uso de transações embutidas. Neste esquema, uma árvore de transações é realizada. A efetivação real só ocorre se a transação raiz for efetivada. Uma sub-transação pode ser efetivada (relativamente), tornando os resultados parciais disponíveis para outros usuários. Mas se uma ancestral direta ou indireta for abortada, o sistema desfaz a sub-transação. (MOSS 1987) mostra um mecanismo baseado em métodos de "logging" para implementar recuperação em transações embutidas.

II.3.3) Controle de Versões.

Aplicações não convencionais tipicamente requerem que múltiplas cópias de um mesmo objeto sejam armazenadas no banco de dados, representando, geralmente, versões deste construídas ao longo de um projeto.

Versões são usadas mais notadamente para:

a) Representar informações passadas, como um histórico, dos estados sucessivos de um objeto. Nestes casos, conceitos

oriundos de banco de dados temporais podem ser usados, havendo, por exemplo, diferenciação entre os tempos de realização, armazenamento e validade de uma versão.

b) Modelar alternativas para um mesmo objeto. Neste caso, o mecanismo de versões deve permitir implementar uma hierarquia de versões.

Muitas são as propostas na literatura para manipulação de versões.

(HASKIN 1981) aproveita a semântica de relacionamentos de composição do System R (vide 2.1.1) para implementar versões.

(BANERJEE 1987b) e (CHOU 1986) classificam as versões em transientes e de trabalho, onde diversas restrições de integridade são aplicadas. Uma versão transiente é derivada a partir de uma de trabalho. Uma versão de trabalho não pode sofrer modificações, enquanto a transiente pode. Um objeto pode referenciar uma versão de outro objeto estática ou dinamicamente. A referência estática é uma referência a uma versão específica, enquanto a referência dinâmica é genérica, isto é, a versão específica referenciada pode variar com o tempo, podendo ser a mais recente ou qualquer outra declarada pelo usuário.

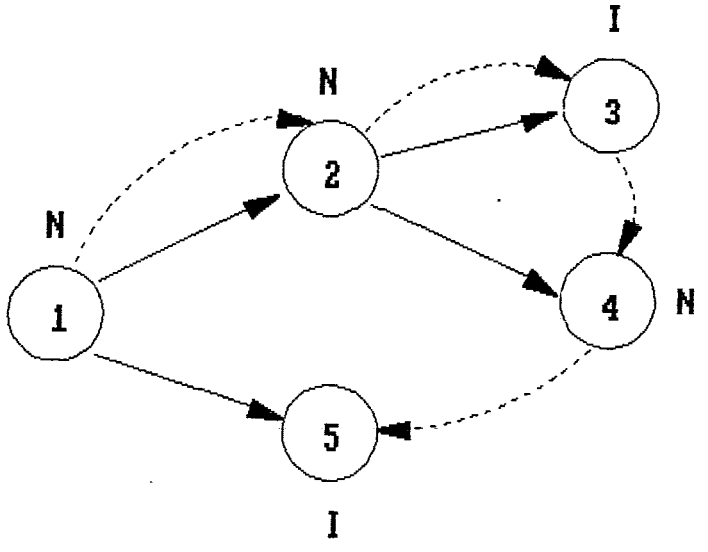
O sistema IRIS (FISHMAN 1987) planeja usar o mecanismo de versões para ajudar a obter maior concorrência.

(KLAHOLD 1987) propõe um modelo geral para o gerenciamento de versões em SGBD's que permite criar um "Ambiente de Versões" usando duas estruturas:

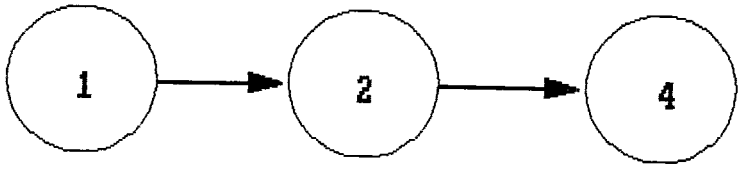
a) Ordenamento - onde diversas versões do ambiente podem participar de um ou mais ordenamentos, não necessariamente lineares;

b) Particionamento - através de propriedades, diversas versões de um ambiente podem ser agrupadas em classes, que podem determinar o conjunto de operações válidas sobre elas.

Por exemplo: o objeto DFD (vide 2.2) pode ter versões representadas de acordo com a figura 2.21 (a).



(a)



(b)

Figura 2.21 - Versões do Objeto DFD.

Na figura, são mostrados dois ordenamentos. O primeiro, representado pela seta cheia, é um grafo de derivação das versões. O segundo, representado pela seta tracejada, é uma lista em ordem cronológica da criação das versões. Os nodos grafados com I e N representam dois particionamentos. I significa DFD's que foram implementados e N, os que não foram. No exemplo, podemos imaginar que o projetista definiu o DFD1, refinou-o sucessivamente em DFD2 e DFD3, e achando que a análise estava correta, implementou o DFD3. Mais tarde, já com o sistema em produção, foi descoberto um erro de análise. A partir do DFD2 foi gerado o DFD4, onde, por alguma incoerência, foi percebido que o erro estava no

próprio DFD2. Finalmente, o DFD5 foi gerado a partir do DFD1 e implementado.

Os mecanismos descritos no trabalho são bastante poderosos. Por exemplo: um usuário poderia requisitar uma visão do "Ambiente de Versões" onde aparecesse somente DFD's não implementados, sem se importar com a ordem cronológica. A figura 2.21 (b) mostra esta visão.

As dificuldades encontradas para modelar versões são, em ordem crescente:

a) Modelagem de Versões de Instâncias de Objetos.

O sistema deve suportar, eficientemente, o mecanismo de versões, tanto no armazenamento - onde somente as partes modificadas entre versões sucessivas devem ser armazenadas - como na recuperação, devendo se evitar codificações ineficientes para representar as diferenças.

b) Modelagem de Versões do Esquema.

Naturalmente, inerente ao próprio processo de projeto, o objeto, sendo construído, pode ter seu esquema alterado entre uma versão e outra. Mudar de uma versão de uma instância para outra pode determinar, também, uma mudança na versão do esquema, o que impõe mais dificuldades no controle de versões. O sistema ORION (BANERJEE 1987b) implementa isto, fazendo uma versão de uma instância apontar para a sua versão do esquema. Em modelos onde o esquema pode ser representado usando as próprias primitivas do mesmo, como no relacional (os meta-dados podem ser representados por relações) e nos modelos orientados a objetos (classes são também objetos), é possível, para modelar e implementar versões de esquema, aplicar as mesmas soluções usadas para versões de instâncias.

c) Modelagem de Versões em Objetos Complexos.

Alguns problemas difíceis aparecem neste caso. Por exemplo: se uma versão nova de um objeto complexo teve

somente alguns sub-objetos alterados, é de se esperar que o novo objeto composto aponte, para economizar espaço de armazenamento, para os sub-objetos que não foram alterados (versão antiga). Isto causa problemas, quanto, por exemplo, à regra de eliminação em cascata dos sub-objetos de um objeto complexo. (BANERJEE 1987b) impõe restrições de integridade adicionais para lidar com este problema; (CHOU 1986) estudou a notificação de objetos que apontam para versões que foram alteradas; enquanto (KATZ 1987) propõe mecanismos para propagar as mudanças em relacionamentos de composição (objetos complexos) e de equivalência.

CAPÍTULO III

PROPOSTAS DE EXTENSÃO DO COPPEREL PARA APLICAÇÕES NÃO CONVENCIONAIS

Neste capítulo estabelecemos o escopo das extensões propostas para o COPPEREL (seção 3.2), a partir dos requisitos de funcionalidade desejáveis para SGBDs não convencionais (seção 3.1).

III.1) Requisitos de Funcionalidade para SGBD's Não Convencionais.

Neste seção analisamos, brevemente, os requisitos de funcionalidade que SGBD's para aplicações não convencionais devem suportar. Os resultados aqui descritos são apoiados nas descrições e comentários das seções 2.1 e 2.2. A seção 2.3 analisou os requisitos de controles operacionais.

III.1.1) Objetos Complexos.

Uma das facilidades imprescindíveis para tornar SGBD's capazes de lidar com aplicações não convencionais é, sem dúvida, a capacidade de definir e manipular objetos complexos. Um objeto complexo é uma entidade ou conceito do mundo real que não pode ser representado diretamente por um registro. Como já observado, a complexidade está, muitas vezes, na limitação representacional do modelo com o qual se tenta exprimir o objeto, mas o objeto em si é uma entidade atômica para o usuário.

Um sistema ideal, que suporte objetos complexos, deve apresentar as seguintes funcionalidades:

a) Definição do Objeto Complexo.

Deve ser possível declarar um objeto complexo para o SGBD. Desta forma, novas operações podem ser aplicadas e restrições de integridade especiais podem ser garantidas. Além disto, o sistema pode armazenar o objeto eficientemente, bloqueá-lo como um todo no acesso concorrente e prover suporte adequado para recuperação, em caso de falha.

Uma outra abordagem do problema permite a declaração dinâmica de objetos complexos: parte de uma intricada rede de elementos atômicos e relacionamentos pode, em um determinado instante, necessitar ser manipulada como uma entidade única.

Os construtores mínimos para criação de um objeto complexo são: o de agregação (tupla), onde um objeto é formado de um ênupla ordenada de outros objetos, e o de conjunto, onde um objeto é formado de um conjunto de outros objetos. Estes construtores devem poder ser aplicados recursivamente.

b) Linguagem de Manipulação.

Esta deve permitir a manipulação do objeto no próprio nivel de abstração dele, isto é, se o objeto for representado internamente por uma série de elementos atômicos e relacionamentos, este fato deve ser escondido do usuário. Devem ser providas operações como: copiar o objeto como um todo, recuperar o objeto a partir de um predicado que se aplique sobre qualquer uma de suas partes componentes, recuperar parte da estrutura do objeto, entre outras. Se o objeto for manipulado por uma linguagem hospedeira mais convencional (imperativa e procedural), algum mecanismo deve ser provido para recuperar o objeto registro a registro. Outras considerações sobre a LMD são traçadas no ítem d.

c) Restrições de Integridade.

Deve ser possível definir restrições de integridade intra e inter-objetos. As restrições inter-objetos são aquelas usuais em SGBD's, as intra-objetos dizem respeito a normas que os componentes de um objeto complexo devem obedecer. Os exemplos mais comuns são: a eliminação dos componentes de um objeto caso o objeto raiz seja eliminado e a obrigatoriedade ao incluir um sub-objeto de que seja parte de um objeto que já exista. Estas restrições são aplicáveis a objetos hierárquicos. Com as possibilidades introduzidas no ítem d, estas restrições podem ou não ser válidas e outras podem ser criadas. Deve ser permitido um mecanismo de definição de restrições mais generalizado.

d) Granularidade, Simetria e Relativismo.

Um objeto deve poder ser manipulado como um todo, ou somente suas partes componentes. Uma parte componente de um objeto pode, por sua vez, representar outro objeto complexo. A LDD e a LMD para objetos complexos devem ser capazes de permitir manipular um objeto em diversos níveis de granularidade. Um usuário deve poder, à sua vontade, mudar, dinamicamente, o nível de granularidade. Isto permite que, em qualquer operação, o usuário possa interagir com o objeto em um processo de refinamentos sucessivos. Algumas vezes, é necessário uma linguagem de alto nível extremamente poderosa; outras vezes, seria interessante poder navegar dentro do objeto complexo.

Simetria é a capacidade de manipular um objeto complexo a partir de qualquer ponto de partida. A maioria das propostas vêem um objeto complexo como sendo formado por uma hierarquia de componentes. Nestes casos, normalmente, o objeto raiz é a "porta de entrada" para o objeto complexo como um todo. Relacionamnetos N:M e sub-objetos compartilhados não são adequadamente modelados com esta restrição. Com uma facilidade de declaração e manipulação que leve em conta simetria, esta restrição desaparece.

Finalmente, relativismo é a capacidade de diversos usuários enxergarem, de forma diferente, um conjunto de entidades e relacionamentos que formam um objeto complexo.

Por exemplo: a partir deste conjunto, dois usuários podem perceber objetos complexos diferentes, cujos conjuntos de componentes podem possuir interseção vazia ou não, ou um ser o subconjunto do outro. Ainda: o que um vê como uma entidade, pode, para o outro, ser um relacionamento.

Granularidade, relativismo e simetria são, no fundo, variantes de um mesmo problema.

e) Recursividade.

É comum necessitar representar objetos que são componentes de partes deles mesmos. O modelo deve permitir definir estes tipos de objetos e prover mecanismos de fechamento transitivo para manipulá-los.

A recursividade deve ser permitida em mais de um componente do objeto complexo. Por exemplo: no objeto DFD, usado na seção 2.2, se considerou o fato de PROCESSOS serem compostos de outros PROCESSOS. Estes também continham DEPÓSITOS__DE_DADOS que, por sua vez, continham DADOS. A modelagem correta seria permitir que DADOS pudessem ser formados por outros DADOS (a análise estruturada considera registros e vetores).

f) Paralelismo.

Simetria e recursividade sugerem a possibilidade de processamento paralelo na manipulação do objeto. Uma LMD pode considerar a necessidade de mecanismos explícitos para explorar esta capacidade em máquinas paralelas. Linguagens funcionais, que são boas candidatas para processamento paralelo, aparecem, também, como um modelo computacional adequado para lidar com recursividade, granularidade e simetria. Além disto, alguns modelos semânticos já consideraram este modelo computacional para implementar o aspecto operacional dos objetos.

g) Outras semânticas no relacionamento intra componentes em objetos complexos.

Normalmente, a semântica do relacionamento entre componentes de um objeto complexo é do tipo "part-of". As possibilidades de manipulação sugeridas no ítem d, e um mecanismo generalizado de definição de restrições de integridade, como mencionado no ítem c, podem permitir a exploração de outras semânticas no relacionamento entre objetos componentes de um objeto complexo. Por exemplo: no LOOPS (STEFIK & BOBROW 1986) um objeto complexo pode representar perspectivas (vide item 2.2.7).

h) Comportamento.

As facilidades para objetos complexos de um SGBD devem estar atreladas a um mecanismo de tipos abstratos de dados (vide 3.1.2) de forma a permitir que operações específicas - e não somente as básicas de manipulação - possam ser definidas e aplicadas. As vantagens obtidas pelo encapsulamento de operações podem ou não impedir as facilidades de granularidade, simetria e relativismo, de acordo com a vontade do usuário.

i) Dinâmica.

Este ítem diz respeito à necessidade de mudar, dinamicamente, o esquema de um objeto complexo. Esta necessidade é inerente a problemas de projeto, onde um objeto complexo é construido, paulatinamente, ao longo do tempo. Muitas vezes o SGBD contém somente uma instância do objeto complexo. Durante o projeto, além de serem acrescidos ou modificados componentes pertencentes ao esquema já existente, e principalmente, se o problema for novo, surgirá a necessidade de refinar o esquema do objeto complexo. O SGBD deve ser capaz de suportar estas mudanças com um mínimo de custo. Esta facilidade não diz respeito unicamente à estrutura do objeto mas, também, aos aspectos de comportamento e restrições de integridade. Este ítem é abordado, novamente, onde vemos versões (2.3.3).

III.1.2) Tipos Abstratos de Dados.

Um mecanismo de TAD é um dos princípios pelo qual um SGBD pode conhecer os objetos que uma base de dados manipula, sejam estes atômicos ou complexos.

As propostas de Sistemas de Gerência de Banco de Dados Orientados a Objetos trazem, inerentemente, o mecanismo de TAD. Neste caso, o mecanismo é mais tradicional, no sentido que o encapsulamento é garantido, isto é, os objetos só podem ser acessados via operações definidas no tipo.

Já nas propostas de extensão do modelo relacional, o mecanismo de TAD é mais relaxado, permitindo, também, a manipulação dos objetos via outras operações. Basicamente, é possível definir um novo tipo de atributo e operações sobre ele. A operação pode ser aplicada na linguagem de consulta em qualquer ponto onde uma expressão possa ser usada. A operação devolve um valor, geralmente de um dos tipos já pré-definidos (como: inteiro, caracter ou booleano) que pode ser usado em um predicado. Outras propostas permitem a definição de uma operação que recebe uma tupla inteira como parâmetro, podendo esta ser aplicada somente sobre a relação para a qual foi declarada. As operações, geralmente, podem ser implementadas em linguagens de programação comuns.

Segundo (DITTRICH 1986) somente o casamento de uma facilidade para objetos complexos com uma facilidade de TAD, que não manipule somente tipos primitivos, permite atingir orientação a objetos comportamental.

Por outro lado, um mecanismo generalizado e poderoso de TAD pode ser suficiente para implementar objetos complexos. O problema é determinar a fronteira entre quais primitivas devem ser suportadas pelo sistema e quais devem ser construídas através do mecanismo de TAD. Talvez a pesquisa de SGBD's extensíveis indique um conjunto mínimo de primitivas que permita, entre outros, prover os controles operacionais básicos para que toda a facilidade de objetos

complexos seja implementada via TAD. Assim, um TAD deve permitir declarar, também, as restrições de integridade sobre o tipo e/ou procedimentos a serem automaticamente disparados na ocorrência de uma determinada condição.

III.1.3) Campos Longos.

Normalmente, qualquer sistema define um conjunto de entidades ou tipos atômicos como: inteiro, real, cadeia de caracteres e outros. Um TAD pode ser construído a partir de um tipo atômico como estes, mas o tamanho da representação interna destes tipos limita muito as possibilidades de implementação. Fica difícil representar tipos estruturados como vetores e matrizes ou informações que, por sua natureza, exigem grande espaço de armazenamento como: documentos, gráfico, imagens e outras.

Um SGBD para aplicações não convencionais deve permitir armazenar e recuperar, eficientemente, uma sequência de bytes de tamanho qualquer. Normalmente são incluídas operações para recuperar, incluir e eliminar um pedaço de tamanho qualquer em qualquer posição do campo longo.

Estas operações enxergam o campo longo como uma sequência ordenada de bytes, que são realizadas informando-se a posição relativa dentro do campo longo. Além deste tipo de manipulação, deve ser fornecida alguma forma de manipulação simbólica (vide proposta no capítulo 5).

O uso de campos longos com facilidades para TAD permite construir qualquer entidade atômica ou complexa, no que diz respeito ao aspecto estrutural.

III.1.4) Relacionamentos "IS-A".

Muitas propostas, como visto em 2.1 e 2.2, incluem mecanismos de generalização e especialização que variam

entre si em alguns detalhes (vide 2.1.1). Estes mecanismos parecem ser uma necessidade de SGBD's, em geral, e não, especificamente, de SGBD's para aplicações não convencionais.

O fato de uma subclasse significar uma nova classe que é ligeiramente diferente da superclasse que a originou, sugere alguma pesquisa no sentido de especializações serem usadas para representar novas versões de esquemas (mais sobre verões em 2.3.3)

Dentro do espírito do ítem g de 3.1.1, um mecanismo de definição de semântica de relacionamentos em objetos complexos pode fornecer suporte para, particularizando, implementar relacionamentos "is-a"; tudo isto, em um ambiente onde o próprio esquema do banco de dados pode ser visto como um grande objeto complexo.

III.2) O SGBD COPPEREL e as Alterações Propostas.

O COPPEREL (MATTOSO 1985, 1987, ZAKIMI 1982a, 1982b) é um sistema de Gerência de Banco de Dados Baseado no Modelo Relacional inteiramente desenvolvido por pesquisadores da COPPE/UFRJ.

Uma única linguagem - a LOPEREL - está disponível, englobando os comandos de descrição e manipulação de dados.

Uma base de dados criada pelo COPPEREL é, do ponto de vista do sistema operacional, um arquivo onde estão armazenadas as relações da base, criadas pelo usuário e, também, as relações do sistema, criadas automaticamente, que respondem pelo armazenamento das definições do esquema. Assim, dados e meta-dados são manipulados quase uniformemente. A única restrição quanto a manipulação das tabelas do sistema é que estas não podem sofrer inclusão de registros através da lingugem de manipulação, mas somente, indiretamente, através da linguagem de descrição de dados.

Do ponto de vista de flexibilidade e funcionalidade de descrição do esquema, o COPPEREL apresenta as seguintes vantagens:

- capacidade de definição de asserções de integridade usando os próprios comandos de manipulação da LOPEREL;
- índices baseados em técnicas de hashing;
- definição dinâmica de relações, índices, vistas, procedimentos e asserções de integridade;
- tipos primitivos: inteiro, real e alfanumérico;
- segurança contra acesso indevido.

Do ponto de vista de manipulação de dados, estão disponíveis:

- operações de cópia, união, interseção, diferença, projeção, junção, divisão, seleção de relações e ainda: inclusão, remoção e modificação de tuplas.
- macro-procedimentos sintáticos;
- controle de fluxo (seleção, interação e desvio);
- definição de variáveis;
- cálculo de expressões;

A arquitetura do sistema é baseada em um analisador/tradutor que se comunica com a máquina virtual.

O analisador/tradutor é responsável por receber os comandos da LOPEREL, realizar as análises léxica e sintática e, utilizando-se de chamadas à maquina virtual, realizar, também, testes semânticos. A razão disto é que a tabela de símbolos do analisador/tradutor é composta das relações do sistema. O analisador/tradutor gera instruções em código intermediário para serem executadas na máquina virtual COPPEREL.

A máquina virtual COPPEREL é a responsável pela execução de todos os comandos. O Executor implementa as funções da máquina virtual através de dois conjuntos de módulos: um que se compõe dos módulos que respondem pelas

primitivas de acesso e outro que se compõe dos módulos que respondem pelas instruções da máquina.

Dado o atual estágio do COPPEREL e os requisitos de funcionalidade desejáveis para SGBDs não convecionais (3.1), optou-se por trabalhar, nas extensões do COPPEREL, nos seguintes caminhos:

i) Construção de uma interface para linguagem hospedeira, que permita a utilização do COPPEREL atráves de uma liguagem de programação de uso geral, que possa ser mais adequada para a construção de aplicações específicas;

ii) Definição de um novo tipo de dado - o campo longo - que pode, também, ser útil na construção de um Gerente Geral de Armazenamento de Objetos, e a implementação de seus algoritmos.

iii) Definição de uma proposta para incorporação de objetos complexos, sem ferir a arquitetura geral do sistema e tratando a princípio do enfoque estrutural (DITTRICH 1986);

Os capítulos IV, V e VI tratam, respectivamente, destas extensões.

CAPÍTULO IV
UMA INTERFACE ENTRE O COPPEREL E LINGUAGENS HOSPEDEIRAS.

O Banco de Dados COPPEREL possui uma única janela para manipulação, definição e gerência de dados que é a linguagem LOPEREL. Esta linguagem é baseada na álgebra relacional, possuindo, também, características de linguagens convencionais como: procedimentos parametrizáveis, contrôle de fluxo, definição de variáveis e comandos de entrada e saída (MATTOSO 1982). Embora possa ser utilizada em lotes, o seu objetivo principal é o uso interativo. Por isto, LOPEREL apresenta-se como uma linguagem autocontida. Estando independente de qualquer outra linguagem, o trabalho do usuário casual é facilitado.

Com o interesse voltado para bancos de dados não convencionais, o COPPEREL sofrerá uma série de modificações que o tornará mais flexível e adequado para este fim. Torna-se imprescindível uma interface entre o SGBD e uma linguagem de alto nível que permita a construção de novas aplicações que não poderiam ser implementadas puramente em LOPEREL. No caso específico do projeto TABA (SOUZA 1988a), ferramentas para ambientes de desenvolvimento de software serão implementadas usando o COPPEREL como SGBD subjacente, como por exemplo FEGRES (TRAVASSOS 1988).

Como parte do trabalho desta tese, uma interface entre o COPPEREL e linguagens hospedeiras foi implementada. Neste capítulo, descrevemos as premissas de projeto, a definição de sua funcionalidade e seu modo de operação.

IV.1) Premissas.

Existem diversas maneiras de implementar a interface entre o COPPEREL e linguagens hospedeiras:

i) O programa hospedeiro pode chamar a máquina virtual do COPPEREL ;

ii) O programa hospedeiro pode passar comandos de mais alto nível (como a própria LOPEREL);

iii) A interface pode ser geral ou especializada por operação;

iv) O programa hospedeiro pode ser ligado diretamente com o COPPEREL

v) O programa hospedeiro pode usar alguma forma de troca de mensagens (em um ambiente multi-tarefa).

Estas formas de implementação oferecem diversos graus de dificuldade de construção, desempenho geral e facilidade de uso pelo programador da aplicação. Adotamos, então, algumas premissas básicas que nos guiaram na escolha de características da interface. São elas:

1) O COPPEREL deve poder ser acessado a partir de qualquer linguagem de alto nível. O implementador de uma aplicação deve ser livre para escolher a linguagem de programação que mais se adeque à construção desta, e não à manipulação da base de dados;

2) O implementador de uma aplicação deve ser encorajado a usar o COPPEREL e não desestimulado; por isto a linguagem de manipulação de dados, a linguagem de definição de dados e a própria interface devem ser simples de serem usadas;

3) O objetivo final é testar a utilidade das extensões do COPPEREL e não despender muito esforço, criando pré-compiladores ou interfaces complexas porém eficientes. Além disto, provavelmente, a interface será redefinida para suportar estas novas extensões, como, por exemplo, manipulação de objetos de tamanho variável.

Desta forma, a interface:

1) Foi constituída de uma subrotina responsável pela comunicação entre a aplicação e o SGBD;

2) A aplicação envia comandos em LOPEREL para a interface, evitando que o programador tenha que aprender os comandos da máquina virtual e garantindo que os testes semânticos providos pela LOPEREL sejam executados;

3) A chamada é interpretada. O desempenho é menor mas evita a construção de pré-compiladores. Uma alternativa, que oferece um desempenho maior e cuja complexidade de implementação não é tão grande, será esboçada adiante nesta mesmo capítulo;

4) Inicialmente, o programa de aplicação, a interface e o COPPEREL estão ligados como um único bloco executável. Mais tarde, pode ser desenvolvido um mecanismo para troca de mensagens entre a interface e o SGBD que é adequado para ambientes multi-tarefa. Neste caso, o programa de aplicação forma, junto com a interface, um único bloco executável. Esta alteração é transparente para o programador.

IV.2) A Definição da Interface.

O projeto lógico da interface constou de:

1) Definir a forma de chamada da interface, incluindo definições de parâmetros de entrada e saída;

2) Definir o mecanismo de troca de registros entre o programa principal e a interface;

3) Definir o uso de variáveis da linguagem hospedeira onde em LOPEREL é permitido o uso de constantes ou variáveis LOPEREL;

4) Definir o mecanismo de sinalização de erros encontrados pelo COPPEREL para o programa principal;

5) Definir uma operação nova da LOPEREL de nome "PROXIMO_REGISTRO" que permitirá o tratamento de um registro por vez. Embora a LOPEREL permita a definição e manipulação de cursores, estes devem ser usados dentro de um bloco autocontido, isto é, um registro pode ser acessado por vez, somente para ser manipulado pela própria LOPEREL;

O restante desta seção trata da definição da interface.

A interface ao COPPEREL foi implementada como uma rotina única (generalizada) de nome CPREL. A razão de ser uma rotina somente é facilitar o uso e aprendizado da interface, uma vez que um único protocolo de acesso pode ser usado. A chamada será feita como subrotina (disponível na maioria das linguagens), determinando que a troca de dados seja feita através de parâmetros. O fato de linguagens diferentes tratarem áreas de memória comuns de formas diferentes afasta a possibilidade de troca de informações por este meio.

A forma da chamada é:

```
CALL CPREL (codigo_retorno, comando, buffer,
            variável_hosp, variável_hosp,....)
```

Os parâmetros a serem passados para a interface CPREL são, por definição:

Parâmetro : CODIGO_RETORNO
Tipo : inteiro, 4 posições
E/S : saída
Uso : obrigatório
Significado: Este parâmetro indica se a execução da interface foi bem sucedida ou não. Seus possíveis valores podem indicar erros léxicos, sintáticos, semânticos ou de execução do COPPEREL, em uma correspondência biunívoca com as mensagens de erro descritas no Apêndice D do Manual de Referência da LOPEREL. Podem ainda indicar erros no uso da interface, como passagem errada de parâmetros. Como boa técnica de programação, após cada chamada à interface CPREL, este código de retorno deve ser testado; em caso de erro, o programa principal pode tomar uma atitude apropriada. Uma outra rotina auxiliar de nome "OBTEM_MENSAGEM" será fornecida. Seu objetivo é obter o texto da mensagem de erro a partir do código de erro.

Parâmetro :COMANDO
Tipo : cadeia de caracteres de tamanho variável
E/S : entrada
Uso : obrigatório
Significado: Este parâmetro contém a(s) sentença(s) (ou "transações", usando a nomenclatura definida no Manual de Referência da LOPEREL) a serem enviadas para execução pelo COPPEREL. Uma transação (delimitada por uma sentença para abrir ou criar a base de dados e uma sentença para fechar a base de dados) é recheada por "gerências" ou "comandos".

Todos os elementos da LOPEREL estão disponíveis e podem ser usados através da linguagem hospedeira, porém, alguns detalhes devem ser observados:

1) Devem ser passados uma ou mais "gerências" ou "comandos" completos (delimitados por ";");

2) O elemento "instrução" é composto por um "comando", "desvio" ou "instrução condicional". "Instruções" pertencem a um corpo que define uma asserção, vista ou bloco e podem ser rotuladas. "Instruções" podem ser usadas (mesmo rotuladas) se todo o corpo da definição (da asserção, vista ou bloco) for passado de uma só vez para a interface.

3) Os "comandos" de seleção, modificar_registro e remover_registro envolvem "condições". Uma condição é uma expressão lógica simplificada onde os primários são nomes de variáveis LOPEREL, nomes de atributos ou constantes. É necessário que se possa incluir, também, variáveis da linguagem hospedeira. A interface CPREL avalia o valor da variável e envia uma constante para o COPPEREL.

Um marcador especial (&), embutido na condição, indica que, ali, é o lugar de uma variável da linguagem que será passada como um parâmetro adicional. O seguinte exemplo em FORTRAN ilustra esta situação:

```
      READ (100, 5) NOME
  100 FORMAT ('Entre com o nome do editor', 10A)
      COMANDO = 'SELECIONAR CARACTERISTICAS TAL QUE
                EDITOR = & EM $AUX'
      CALL CPREL (COD_RET, COMANDO, NOME)
```

Além disso, o "comando" modificar_registro envolve expressões que são atribuidas aos campos que estão sendo modificados. Estas expressões também podem ser substituidas pelo marcador "&".

4) Os "comandos" inserir_registros e proximo_registro envolvem o uso de um buffer para enviar ou receber registros da base de dados.

Parâmetro :BUFFER
Tipo : cadeia de caracteres de tamanho variável
E/S : entrada ou saída
Uso : opcional

Significado: Se o "comando" passado para a interface for inserir_registros (opção INSERIR REGS SEGS) ou proximo_registro, então a interface CPREL interpretará o terceiro parâmetro como sendo o buffer de onde ou para onde serão transferidos os registros da base de dados. Atualmente, o COPPEREL suporta 3 tipos de dados: real, inteiro e alfanumérico.

Em LOPEREL, a declaração dos tipos inteiro e real é acompanhada do formato de leitura e gravação (o formato interno depende do computador hospedeiro), enquanto no tipo alfanumérico acompanha o tamanho do campo que indica tanto a representação interna quanto a externa. Como a linguagem hospedeira pode ser qualquer, os formatos internos dos tipos das variáveis destas podem não coincidir com os formatos internos dos dados armazenados. Assim, o formato a ser passado no buffer deve ser o formato externo, em caracteres ASCII.

Como exemplo, suponhamos o seguinte arquivo COPPEREL:

```
CRIAR ARQ TESTE COM 1000 REGS DE 3 ATRS:
      VAR_INTEIRA : I(6)
      VAR_REAL    : R(8:2)
      VAR_ALFA    : A(10)
COM CHAVE VAR_INTEIRA;
```

E o trecho de programa FORTRAN:

```
INTEGER*4 V_INT
REAL*8 V_REAL
CHARACTER*5 V_ALFA
CHARACTER*30 BUFFER
V_INT = 3
V_REAL = 12.3
V_ALFA = 'OI'
CALL PREPARA_BUFFER (V_INT,V_REAL,V_ALFA,BUFFER)
COMANDO = 'INSERIR REGS SEGS EM TESTE'
CALL CPREL (EST,COMANDO,BUFFER)
```

Após a chamada a rotina PREPARA_BUFFER, a variável BUFFER deve ter o conteúdo:

```
000003/0000001230/OIbbbbbbbb///
 I(6)    R(8:2)   A(10)
```

O buffer passado para a interface pode ter tamanho maior que o necessário, porém as rotinas para conversão de tipos internos da linguagem (inteiros e reais) em caracteres ASCII devem ser escritas pelo próprio programador. Observe que os caracteres "/" exigidos pela LOPEREL não devem ser omitidos. A ordem dos atributos no buffer deve ser a mesma destes no arquivo, sendo de responsabilidade do programador obedecer a esta regra.

O trecho de programa abaixo lê este mesmo registro do banco de dados:

```
V_INT = 3
V_ALFA = CHAR (V_INT)
COMANDO = 'SELECIONAR TESTE TAL QUE VAR_INTEIRA = &
          EM $AUX'
CALL CPREL (COD, COMANDO, V_ALFA)
CALL CPREL (COD,'PROXIMO_REGISTRO $AUX',BUFFER)
```

Na leitura do registro, o programa encontrará o buffer preenchido como na figura abaixo. Neste caso, a responsabilidade de fornecê-lo assim foi da interface.

```
00000300000012300Ibbbbbbbb
```

A passagem dos dados em ASCII, facilita a manipulação externa (por exemplo: enviar os dados para a tela) pela linguagem hospedeira, porém esta é responsável por converter os dados em uma representação favorável a sua manipulação para outros fins.

Uma outra opção seria, ao invés de usar o parâmetro BUFFER, permitir que o programador passe cada uma das variáveis correspondendo aos atributos do arquivo COPPEREL. Isto, porém, não livraria de fazer as conversões para ASCII e poderia complicar a clareza quando o arquivo tiver muitos atributos.

A principal vantagem da solução adotada é, sem dúvida, a sua independência da linguagem hospedeira e a sua generalidade, que será útil para manipular novos tipos que venham ser incorporados pelo COPPEREL. Isto indica, também, a necessidade de se definir, sempre, uma representação externa para estes novos tipos.

Parâmetro : VARIÁVEL_HOSPEDEIRA
Tipo : cadeia de caracteres de tamanho variável
E/S : entrada
Uso : opcional

Significado: Quando a transação enviada para a interface for um "comando" seleção, modificar_registro ou remover_registro, este pode vir entremeado com caracteres "&" onde houver condições ou expressões a serem atribuídas a campos no comando modificar_registro. Cada caracter "&" representa uma variável da linguagem que é passada como um parâmetro deste tipo. O número de parâmetros do tipo VARIÁVEL_HOSPEDEIRA é igual ao número de caracteres "&" encontrados no parâmetro COMANDO. Se houver menos variáveis hospedeiras do que caracteres "&", ou se a ordem destas não

corresponder à ordem no COMANDO, os resultados serão imprevisíveis (porém sem comprometer a segurança e a integridade dos dados).

Uma VARIÁVEL_HOSPEDEIRA deve ser do tipo cadeia de caracteres.

Veja o exemplo:

```
V_INT = 3
ALFA1 = CHAR (V_INT)
V_ALFA = 'OI'
COMANDO = 'SELECIONAR TESTE TAL QUE VAR_INTEIRA = & E
           VAR_ALFA = & EM $AUX'
CALL CPREL (EST, COMANDO, ALFA1, V_ALFA)
```

Concatenado ao caracter "&" podem vir mais alguns caracteres para efeito de documentação. Por exemplo:

```
COMANDO = 'SELECIONAR TESTE TAL QUE VAR_INTEIRA = &ALFA1
           E VAR_ALFA = &V_ALFA EM $AUX'
```

IV.3) O Comando PROXIMO_REGISTRO.

Este comando permite o acesso de um registro por vez de uma determinada relação. Esta funcionalidade é necessária quando se trabalha a partir de uma linguagem hospedeira, e por isto, não existia antes na LOPEREL. Sua sintaxe e semântica são definidas a seguir. Os símbolos usados para definir a sintaxe são os mesmos usados no Manual de Referência da LOPEREL.

Sintaxe:

{PROXIMO} [REGISTRO] nome_arquivo [:nome_atributo,,,]

Semântica:

Este comando recupera um registro de cada vez do nome_arquivo especificado. A ordem em que os registros são recuperados é a ordem em que se encontram no arquivo. A primeira vez que este comando é emitido para um nome_arquivo é recuperado o primeiro registro. Em cada vez subsequente, é recuperado o próximo registro na ordem interna do nome_arquivo, até que todos os registros sejam percorridos (quando uma mensagem de erro é retornada ao usuário) ou a transação acabe. Se uma nova transação começar, a emissão deste comando causará a recuperação, novamente, do primeiro registro.

Este comando pode ser emitido, intercaladamente, para arquivos diferentes. O COPPEREL retorna o próximo registro na sequência interna.

A implementação deste comando teve, como objetivo, atender à necessidade do uso da LOPEREL através de linguagens hospedeiras, embora também possa ser utilizado interativamente. Assim, não apresenta nenhuma flexibilidade em relação ao formato de saída do registro (ao contrário dos comandos MOSTRAR, IMPRIMIR e GRAVAR). Os atributos especificados em nome_atributo são transformados em sua representação ASCII, concatenados e enviados para a tela (LOPEREL interativa) ou para o programa principal (LOPEREL com linguagem hospedeira). Se nenhum nome_atributo for especificado, então são enviados todos os atributos do arquivo, na ordem em que aparecem neste.

IV.4) Modificações no COPPEREL para suportar a interface.

Para o funcionamento do COPPEREL com a interface desenvolvida, alguns módulos do COPPEREL tiveram de ser

modificados. Estes módulos são basicamente aqueles relacionados com a Entrada/Saída, seja devido a LOPEREL interativa ou relacionados a exibição/inclusão de registros.

Basicamente o que foi alterado foi a entrada do léxico, o sintático, os módulos de inserção de registros, de tratamento de erros, e as rotinas do comando PRÓXIMO_REGISTRO, que são baseadas no comando MOSTRAR.

Assim, atualmente, existem 2 versões do COPPEREL: a primeira corresponde ao COPPEREL interativo, já existente; a segunda é composta de um conjunto de bibliotecas de módulos objeto para serem ligadas à interface CPREL e ao programa principal, escrito na linguagem hospedeira.

Uma breve descrição das modificações efetuadas em cada módulo pode ser encontrada no Apêndice B. A descrição da função destes módulos pode ser encontrada em (MATTOSO 1987).

IV.5) Desenvolvimento Futuro.

IV.5.1) Otimização.

Uma otimização possível na interface CPREL é a possibilidade de diminuir-se o número de intervenções da LOPEREL em uma mesma chamada que é executada diversas vezes. Isto pode ser feito de uma forma transparente ou não, para o programador.

A primeira vez que uma determinada chamada for realizada por um programa, além da execução do comando propriamente dito, é gerada urepresentação interna - comandos da máquina virtual COPPEREL - correspondente a este. Nas próximas chamadas, o tempo de execução é menor, uma vez que a interface pode fazer chamadas imediatas à máquina virtual.

A forma de trabalho não transparente inclui um identificador para o comando "compilado", trocado entre a interface e o programa principal. Aplicações que necessitem

de alto desempenho em pontos críticos poderiam forçar a "compilação" dos comandos em pontos de calmaria, para uso posterior. A forma transparente implica em uma certa perda de flexibilidade. Um determinado ponto de chamada no programa principal não poderia emitir, durante a execução do mesmo, comandos diferentes.

IV.5.2) Tratamento de Erros.

Como todo o controle entre o programa de aplicação e o COPPEREL é feito através da interface, o mecanismo de identificar erros por um código de retorno pode ser trocado ou complementado por um mecanismo de sinalização de exceções.

Usando uma outra subrotina denominada DECLARA_TRATADOR, o programa de aplicação pode associar a cada erro específico o nome de uma rotina para tratamento e recuperação. Quando um erro for detectado pela interface, ao invés de retornar um código de estado, a tabela criada por DECLARA_TRATADOR é pesquisada, procurando-se um tratador associado. Se houver, este é chamado pela interface.

Se o tratador não abortar a execução do programa, o controle volta para a interface que o devolve para o programa principal. O tratador não pode fazer chamadas à interface.

A implementação desta facilidade pode significar clareza no código do programa principal, evitando-se os constantes testes após as chamadas à interface. Seu uso não invalida o comando da LOPEREL "EM CASO DE ERRO"; este pode ter precedência sobre o tratamento de erros associado ao programa hospedeiro.

IV.5.3) Conversão de Tipos.

Com o objetivo de facilitar o trabalho do programador, a interface pode ser estendida para entender os tipos de dados da linguagem hospedeira.

A vantagém é eliminar o parâmetro BUFFER e o trabalho necessário para montá-lo ou interpretá-lo. O exemplo do ítem anterior ficaria:

```
INTEGER*4 V_INT
REAL*8 V_REAL
CHARACTER*5 V_ALFA
V_INT = 3
V_REAL = 12.3
V_ALFA = 'OI'
COMANDO = 'INSERIR REGS SEGS EM TESTE'
CALL CPREL (EST, COMANDO, V_INT, V_REAL, V_ALFA)
```

Onde os parâmetros devem ser passados na ordem que ocorrem na definição do arquivo TESTE.

Para a leitura a vantagem seria a mesma, eliminando a necessidade de converter as variáveis hospedeiras. Por exemplo:

```
V_INT = 3
COMANDO = 'SELECIONAR TESTE TAL QUE VAR_INTEIRA = &
             EM $AUX'
CALL CPREL (EST, COMANDO, V_INT)
COMANDO = 'PROXIMO_REGISTRO $AUX'
CALL CPREL (EST, COMANDO, V_INT, V_REAL, V_ALFA)
```

Para isto, é necessário que a inteface conheça a linguagem que a chama, ou sejam criadas interfaces ajustadas para cada linguagem.

Para facilitar a implementação, devem ser fixados os tipos internos que podem representar um tipo do COPPEREL. Por exemplo: o tipo real do COPPEREL será representado em FORTRAN como REAL*8.

Além de conhecer a representação interna dos tipos da linguagem, a interface deve consultar os meta-dados para saber como interpretar uma determinada variável_hospedeira.

CAPÍTULO V

CAMPOS LONGOS.

Outra extensão indispensável, para fazer o COPPEREL capaz de lidar com aplicações não convencionais, é a inclusão de um novo tipo de dado constituído de uma sequência de bytes arbitrariamente longa e não interpretada. Além dos tipos inteiro, alfanumérico e real, já existentes, um campo de uma relação no COPPEREL pode ser do tipo longo.

Alguns autores como (CAREY 1986a) e (SCHWARZ 1986) apontam o uso de campos longos não somente como mais um tipo de dado, mas, sim, como um gerente geral de armazenamento de objetos. Neste sentido, a implementação de campos longos no COPPEREL visou a acumular experiências para o projeto de um gerente de objetos a ser usado em futuras implementações.

Neste capítulo apresentamos a sintaxe e semântica das novas operações para manipular campos longos, bem como a descrição dos algoritmos implementados. A próxima seção apresenta as premissas de projeto para a incorporação de campos longos que deram origem ao modelo de implementação descrito nas sub-seções seguintes.

V.1) Premissas.

As propostas na literatura para incorporação de campos longos em SGBD's foram apresentadas na seção 2.2. No COPPEREL, buscamos os seguintes objetivos:

1) Um campo longo pode ser usado para qualquer propósito, logo o sistema não interpreta seu conteúdo. O significado do conteúdo de um campo deste tipo é conhecido somente pela aplicação que o manipula;

2) Um campo longo deve ser tratado de forma extremamente eficiente;

3) Um campo longo deve poder ser manipulado inserido no contexto das operações relacionais inerentes ao modelo e também de forma interativa, pedaço a pedaço;

4) Pedaços de um campo longo devem poder ser referenciados, também, de forma simbólica, obtendo maior independência do aspecto físico.

V.2) Sintaxe e Semântica da LMD para Campos Longos.

Para o usuário, um campo longo é um atributo, como outro qualquer, declarado na criação da relação que o contém.

O campo longo é visto como uma sequência de bytes que pode ser endereçada informando-se o byte inicial e final. A notação Atr (inicio:fim) exprime o intervalo de bytes compreendidos entre a posição inicio e fim do atributo de nome Atr de uma determinada relação.

Toda a manipulação do campo longo é feita através de pedaços. Duas operações fundamentais estão disponíveis: recuperação de um pedaço, simplesmente usando a notação acima ou a troca de um pedaço por outro de tamanho qualquer. As operações de inclusão e exclusão são realizadas através da operação de troca. Para simplesmente incluir, indica-se o byte inicial igual ao final (inserção após este valor). Para simplesmente eliminar, realiza-se a troca pelo pedaço nulo.

O usuário pode trabalhar com o campo longo em duas modalidades. Na modalidade básica, a manipulação do campo longo é muito similar a de um campo alfanumérico, e é realizada dentro das operações normais da álgebra relacional.

Na outra modalidade, chamada interativa, os comandos emitidos para o campo longo referem-se ao último registro acessado ou a todos os registros de uma relação.

A sintaxe e semântica propostas para cada um destes casos são:

V.2.1) Declaração do Campo Longo.

Um campo longo é declarado simplesmente estendendo-se a sintaxe do não-terminal tipificação (apêndice A):

```
tipificação ::=

      { DE TIPO }  { REAL (inteiro:inteiro) }
      {   :     }  { INTEIRO (inteiro)      }
                   { ALFANUMERICO (inteiro) }
                   { LONGO                  }
```

Em negrito aparece a extensão necessária em relação a sintaxe já existente. Não pode ser criado índice de acesso para campos do tipo longo.

V.2.2) Manipulação na Modalidade Básica.

Nesta modalidade, o campo longo é tratado como um campo do tipo alfanumérico, porém, nas operações onde deve ser especificado explicitamente parte do campo, é usada a notação Atr (inicio:fim). Ao invés de apresentarmos a sintaxe detalhada de cada opção, mostramos uma série de exemplos comentados a seguir.

```
1)   CRIAR ARQUIVO PROCESSOS
     COM 100 REGISTROS DE 3 ATRIBUTOS:
          p_no       : INTEIRO (3),
          descricao  : ALFANUMERICO (20),
          repr_graf  : LONGO
     COM CHAVE: p_no
     RAIZ RECURSIVA


2)   INSERIR PARTES SEGUINTES EM PROCESSOS
          COMO COMPONENTE DE p_no = 1 EM PROCESSOS

     1.1/"subprocesso"/"w3333p4444w5555"//


3)   MODIFICAR PROCESSOS
     TAL QUE p_no = 1.1
          (repr_graf (6:10) PARA "w0000p9999")
```

```
4)   SELECIONAR PROCESSOS
     TAL QUE repr_graf (6:10) = "w0000"
     EM $AUX

5)   MOSTRAR $AUX COM p_no, repr_graf (1:18)

     1.1  w3333w0000p9999w55
```

Observações sobre os exemplos:

1) Uma relação de nome PROCESSOS, raiz de um objeto complexo (vide capítulo 6) é criada com 3 atributos. O atributo repr_graf é do tipo longo e contém comandos gráficos que armazenam as operações para exibir o desenho do processo.

2) Neste comando é inserido um registro que contém um campo longo, que é tratado como um campo alfanumérico nesta operação. A partir daí as estruturas para armazenar o campo longo são criadas, sendo estabelecido o seu tamanho inicial.

3) Um exemplo de modificação do campo longo: todas as operações são feitas à base de troca. Assim, o pedaço da posição 6 até 10, que contém "p4444", está sendo trocado por um pedaço maior, ficando o campo longo, neste registro, com o conteúdo: "w3333w0000p9999w5555"..

4) Neste comando, um pedaço do campo longo entra em um predicado de seleção de registros, de uma forma muito similar à manipulação de tipos alfanuméricos. O registro de p_no = 1.1 é recuperado em $AUX.

5) Finalmente, um campo longo é exibido na tela, restringindo-se o seu intervalo. Se $AUX tivesse mais de um registro, todos eles teriam exibido somente o pedaço selecionado do campo longo. Se houver caracteres não ASCII, provavelmente algo estranho acontecerá no terminal de vídeo. O sistema não interpreta o conteúdo do campo longo. Futuramente, podem ser desenvolvidas opções para exibir um campo longo obedecendo a um determinado formato, que poderá

ser sofisticado (por exemplo: exibição gráfica, sonora) ou simples (por exemplo: exibição em hexadecimal).

V.2.3) Manipulação na Modalidade Interativa.

Esta modalidade surgiu pela limitação de certas linguagens hospedeiras de manipular sequências de bytes muito longos e, também, para fornecer mais flexibilidade.

Nesta modalidade, os comandos emitidos para o campo longo referem-se ao último registro acessado em uma relação (através do comando PROXIMO_REGISTRO, definido em 4.3) ou a todos os registros desta, se nenhum registro em particular ainda tiver sido recuperado (se não foi emitido nenhum comando PROXIMO_REGISTRO, para a relação, desde o início da transação) .

A sintaxe e semântica propostas são:

1) Recuperação:

Sintaxe:

MOSTRAR-CAMPO
  nome-relação , nome-campo (inicio:fim)

Semântica:

Recupera o pedaço compreendido entre os bytes início e fim do campo nome-campo da última ou de todas as tuplas acessadas de nome-relação. Se o comando for emitido da LOPEREL interativa, o pedaço é exibido na tela, com as restrições indicadas na observação 5 do ítem acima. Se for emitido através da interface para linguagem hospedeira, o(s) pedaço(s) recuperado(s) volta(m) no buffer de comunicação (vide 4.1)

2) Troca:

Sintaxe:

MODIFICAR-CAMPO

nome-relação , nome-campo (inicio:fim) PARA expressão

Semântica:

Troca o pedaço compreendido entre os bytes início e fim do campo nome-campo da última ou de todas as tuplas acessadas de nome-relação pelos bytes devolvidos pela expressão, que deve ser do tipo expressão-alfanumérica (Manual de Utilização da LOPEREL). Para simplesmente incluir, indica-se o byte inicial igual ao final (inserção após este valor). Para simplesmente eliminar, a expressão deve retornar nulo (ou o literal "").

Outras operações como pesquisa e troca de substring serão implementadas posteriormente.

V.3) Implementação de Campos Longos.

Nesta seção descrevemos a estrutura de armazenamento e os algoritmos de busca, inclusão e exclusão de bytes de um campo longo. Embora toda a manipulação do campo seja feita a partir da troca de pedaços, os algoritmos básicos são os citados. A troca de um pedaço por outro de mesmo tamanho é imediata, sendo o algoritmo similar ao de busca, com a diferença que os bytes, ao invés de serem recuperados, são substituídos; por esta razão, não o descreveremos. Operações de troca por pedaços de tamanho diferente, envolvem este algoritmo e mais o uso do algoritmo de inclusão ou exclusão.

Os algoritmos apresentados, em pseudo-código, embora manipulem uma estrutura de dados cuja definição é tipicamente recursiva, usam interações como forma de obter repetições. A razão é porque os algoritmos tinham, como objetivo, a posterior implementação em FORTRAN.

O Apêndice C mostra o exemplo de uma sessão utilizando a implementação destes algoritmos.

V.3.1) Estrutura de Armazenamento.

A implementação de campos longos é baseada no conceito de relação ordenada, proposta por (STONEBREAKER 1983) e no gerente de armazenamento de objetos do sistema EXODUS (CAREY 1986b).

Fisicamente, um campo longo é representado por um conjunto de páginas (ou folhas) que guarda o conteúdo do campo e por um índice do tipo árvore B+, que é usado para localizar um determinado byte lógico (deslocamento dentro do campo longo) em uma folha. Na relação que contém o campo longo é encontrado, na verdade, o endereço da árvore que controla o campo.

Cada nó da árvore é formado por um conjunto de pares de ponteiro e contador. O ponteiro contém o endereço do nó raiz de uma sub-árvore filha e o contador representa o número do último byte armazenado nela, de forma que o contador do último par do nó raiz do objeto corresponde ao tamanho atual do campo longo. Cada nó interno é idêntico ao nó raiz, mas os contadores obtidos são relativos, isto é, seus valores são absolutos em relação à sub-árvore, mas relativos em relação a árvore como um todo. Para se obter o valor absoluto de um contador é necessário somar o número de bytes da sub-árvore vizinha, à esquerda.

As folhas contêm a sequência de bytes do campo longo e cada uma pode estar entre 50% e 100% preenchida. O último nível da árvore é encadeado formando uma lista, agilizando o processo de busca.

A figura 5.1 representa uma árvore B de um campo longo com 66 bytes de tamanho. Os números embaixo das folhas representam o intervalo lógico de bytes que a folha abriga, e não são armazenados (seu propósito é apenas clarear o esquema). Cada nó desta árvore pode abrigar até 3 pares (p,c) (ponteiro, contador), e as folhas podem abrigar até 10 bytes.

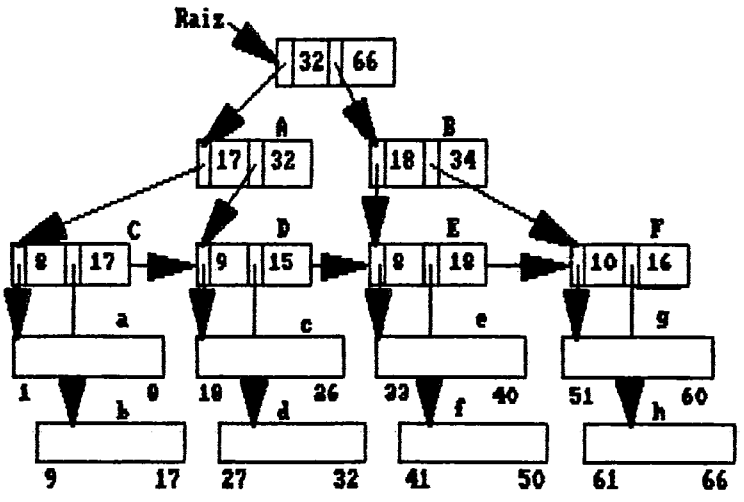


Figura 5.1 - Árvore de Armazenamento de um Campo Longo.

V.3.2) Operação de Busca.

Para recuperar uma sequência de bytes dentro do campo longo, o byte inicial é procurado através da árvore, iniciado-se no nó raiz. Uma vez recuperado o nó raiz, uma pesquisa binária é realizada entre os contadores, buscando-se o menor contador, tal que o seu valor seja maior ou igual que o byte procurado. O ponteiro correspondente contém o caminho da sub-árvore a ser descida. O byte a ser procurado nesta sub-árvore é a posição anterior decrementada do número de bytes da(s) sub-árvore(s) à esquerda. A pesquisa segue até encontrarmos uma folha. Se toda a quantidade de bytes a ser recuperada não estiver nesta folha, as folhas vizinhas são acessadas usando-se informações no nó pai, ou no próximo vizinho à direita do nó pai, através da lista encadeada existente no último nível da árvore.

Algoritmo.

```
Busca (objid, início, fim, buffer-saída)

/* Recupera em buffer-saída os bytes entre início e fim do
campo longo cuja raiz da árvore é objid */

. Recupera em Nó o nó apontado por objid
. Realiza pesquisa binária em Nó procurando o par p(i) e
     c(i) tal que c(i-1) < início <= c(i-1) (PESQ_B)
. Enquanto não atingir uma folha
     . início <- início - c(i-1)
     . Recupera em Nó o nó apontado por p(i)
     . PESQ_B
. Fim-Enquanto
. Recupera em FOLHA a folha apontada por p(i)
. Se o pedaço a mover está todo nesta folha
     então
          . move para buffer-saída
     senão
          . recupera através da lista encadeada as outras
            folhas, movendo-as para buffer-saída
. Fim-se

Fim-Busca
```

Exemplo:

Recuperar os bytes 42 até 54 na árvore de figura 5.1.

. Obtem o nó R (nó RAIZ).

. Pesquisa byte 42 em R, obtendo:
  i=2, c(i)=66, c(i-1)=32, p(i)=B.

. Pesquisa o byte 42 - c(i-1) = 10 em B, obtendo:
  i=1, c(i)=18, c(i-1)=0, p(i)=E.

. Pesquisa o byte 10 - c(i-1) = 10 em E, obtendo:
  i=2, c(i)=18, c(i-1)=8, p(i)=f.

. Logo, o byte 42 está na posição 10 - c(i-1) = 2 da folha f, que tem, ao todo c(i) - c(i-1) = 10 bytes

. Recupera de e, os bytes de 2 a 10. Ficam faltando 4 bytes

. Como i+1 não exite em E, recupera a folha p(1) em F que é g.

. Recupera os 4 bytes restantes de g.

V.3.3) Operação de Inclusão.

Se uma operação de troca envolver inclusão de bytes após uma posição P, o algoritmo de inserção é implicitamente acionado. Este algoritmo diferencia-se do algoritmo tradicional de inserção de chaves em árvore B, pois é capaz de inserir um número arbitrário de bytes em uma única operação, evitando inúmeras atualizações da árvore (e possíveis explosões de seus nós) que seriam necessárias no outro caso, tornando o custo de múltiplas inserções intolerável.

Para inserir N bytes após a posição P, a árvore é pesquisada da mesma forma que no caso da busca, porém os contadores de cada nó visitado são atualizados para refletir a inserção realizada. Ao chegar a folha onde se encontra o byte P, se houver espaço para inserir os N bytes nesta folha, isto é feito e a operação termina. Se não houver, são alocadas tantas folhas quantas forem necessárias para, junto com o espaço restante na folha alcançada, acomodarem os N bytes.

Uma vez que as folhas sejam alocadas e preenchidas, o pai da folha do byte P é atualizado para refletir os novos pares de ponteiros e contadores associados às novas folhas. Novamente, se não houverem pares livres suficientes no nó pai, serão alocados tantos nós quanto necessários para acomodar todos os pares criados. Esta operação continua recursivamente até que todos os nós, ao longo do caminho para P, tenham sido atualizados. Esta operação pode em uma única execução, aumentar em mais de um o número de níveis da árvore.

Nós e folhas serão alocados pelo gerente de espaço existente na máquina virtual COPPEREL, sendo colocados fisicamente próximos. A bufferização é controlada pelos mecanismos existentes anteriormente.

Algoritmo.

Inserir (objid, P, N, buffer-entrada)

/* Insere no campo longo cuja raiz da árvore é objid, após a posição P, N bytes oriundos do buffer-entrada */

. Recupera em Nó o nó apontado por objid
. Realiza pesquisa binária em Nó procurando o par p(i) e c(i) tal que c(i-1) < P <= c(i-1) (PESQ_B)
. Incrementa o valor N em todos os c(j), da posição i até o fim do nó
. Grava Nó no endereço objid
. empilha objid na pilha_caminho
. Enquanto não atingir uma folha
    . P <- P - c(i-1)
    . Recupera em Nó o nó apontado por p(i)
    . PESQ_B
    . Incrementa o valor N em todos os c(j), da posição i até o fim do nó
    . Grava Nó no endereço p(i)
    . empilha p(i) na pilha_caminho
. Fim-Enquanto
. Recupera em FOLHA a folha apontada por p(i)
. Se ainda couberem N bytes na folha apontada por p(i)
    então
        . move de buffer-entrada para esta folha
        . grava a folha
    senão
        . calcula e aloca as folhas necessárias para inserir os N bytes, colocando os seus endereços e quantidades de bytes na pilha1
        . Distribui os bytes de buffer-entrada nas folhas cujos endereços estão na pilha1
        . Rearranjar-Árvore
. Fim-se

Fim-Inserir

```
Rearranjar-Árvore

/* Esta rotina é chamada pela rotina de inserir quando novas
folhas foram alocadas. Se o nó do último nível não tiver
espaço para abrigar as folhas alocadas na pilha1, uma
inserção de nós em massa é tratada. */

. "deu overflow"
. Enquanto der overflow
  . Se deu overflow no raiz
     então
       . aloca novo raiz
     senão
       . Lê e desempilha  o Nó no topo da pilha_caminho
       . Se couber todos os endereços da pilha1
          neste Nó
          então
            . insere elementos da pilha1 neste Nó
            . "fim de overflow"
          senão
            . calcula e aloca os  nós necessários para
              apontar para todas as entradas da pilha1,
              colocando seus endereços e quantidades na
              pilha2
            . distribui os pares (p,c) da pilha1 nos Nós
              alocados na pilha2.
            . Se for o último nível da árvore, encadeá-los
            . pilha2 <- pilha1
            . "deu overflow"
       . Fim-Se
  . Fim-Se
. Fim-Enquanto

Fim-Rearranjar-Árvore
```

Exemplo:

Inserir 41 bytes após o byte 42 da figura 5.1. A figura 5.2 mostra o resultado da operação. Repare que somente os nós que participam do caminho para o byte 8 têm que ser alterados, permanecendo o resto da árvore intacta.

. Caminha-se para o byte 42 como no exemplo acima. Os nós também são incrementados, mas esta alteração acabará sendo desprezada por que haverá overflow e novos valores de contadores serão calculados pelo algoritmo.

. Chegando à folha f, esta já se encontra totalmente ocupada. É necessário alocar novas folhas - i, j, k, l, m - para junto com a folha f, guardarem os 10 bytes desta mais os 41 bytes do buffer de entrada. A nova distribuição é calculada: f, i e j ficam com 9 bytes; k, l, m ficam com 8 bytes. Os endereços i, j, k, l, m bem como suas quantidades ficam na pilha1.

. Seguindo a pilha_caminho, as novas folhas devem ser apontadas pelo nó E já existente. Como este não tem espaço suficiente, novos nós G e H são alocados e colocados na pilha2. A distribução fica assim:

nó E: 3 entradas: folhas e, f, i
nó G: 2 entradas: folhas j, k
nó H: 2 entradas: folhas l, m

Após esta operação, a pilha1 recebe a pilha2, ou seja os nós G e H. Overflow continua sinalizado.

. Seguindo a pilha_caminho, os novos nós devem ser apontados pelo nó B, já existente. Como este não tem espaço suficiente, um novo nó I é alocado, sendo colocado na pilha2. A distribuição fica assim:

nó B: 2 entradas: nós E, G
nó I: 2 entradas: nós H, F

Após esta operação, a pilha1 recebe a pilha2, ou seja o nó I. Overflow continua sinalizado.

. Seguindo a pilha_caminho, o novo nó deve ser apontado pelo nó R, já existente. Como este tem espaço, o ponteiro é inserido e overflow não é sinalizado, terminando a operação.

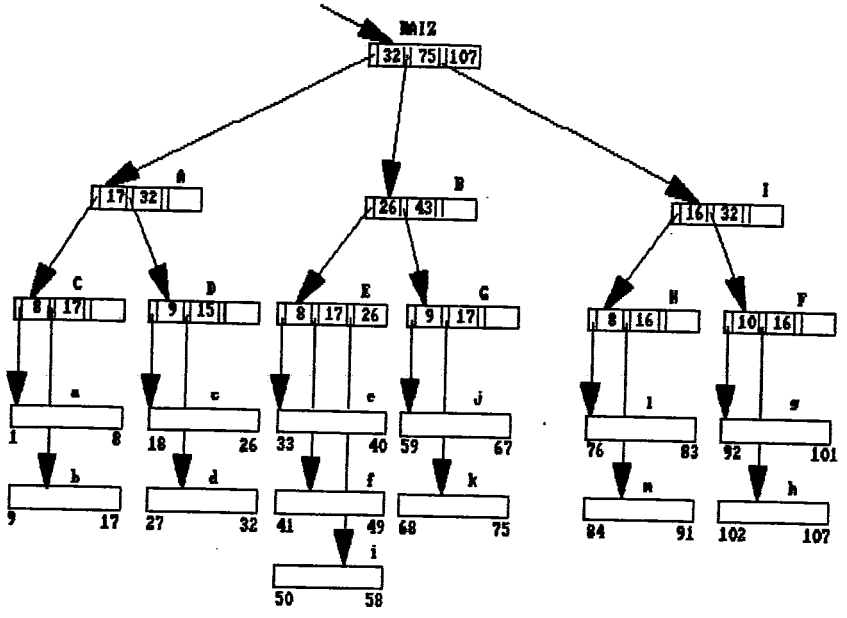


Figura 5.2 - Inserção em um Campo Longo.

V.3.4) Operação de Eliminação.

Para eliminar um intervalo de bytes, um algoritmo capaz de lidar com exclusão em massa de chaves em árvore B também é requerido.

A árvore é percorrida procurando-se, como no algoritmo de pesquisa, os bytes inicial e final do intervalo a ser eliminado. Todas as sub-árvores compreendidas entre o caminho à esquerda (byte inicial) e à direita (byte final) são eliminadas e os nós pertencentes a estes caminhos são atualizados.

Após a eliminação a árvore pode se encontrar desbalanceada. O rebalanceamento é feito na árvore, de cima para baixo, em uma única passada. No momento da eliminação é criada uma pilha em memória que contém informações sobre os caminhos percorridos e o número de entradas (ou bytes) restantes em cada nó (ou folha). Com estas informações a

pilha é percorrida buscando-se nós que estejam sub-utilizados ou em perigo de se tornar sub-utilizados, isto é, quando o número de entradas é tal que o nó não pode correr risco de ter um de seus filhos eliminados sob pena de ficar ele próprio com poucas entradas. Cada nó ou folha nesta situação é consertado, ganhando entradas ou se fundindo com o irmão. Uma raiz sub-utilizada força o reajustamento de seus filhos diretos. Se ficar com uma única entrada, a árvore implode um nível. Novamente, em uma única operação de eliminação, uma árvore pode implodir mais de uma vez.

Algoritmo.

Eliminar (objid, P, N)

/* Elimina no campo longo cuja raiz da árvore é objid N bytes da posição P (inclusive) em diante. Uma pilha, de nome pilha_caminho, é montada à medida que o algoritmo é executado. Seu objetivo é ser usada para rebalancear a árvore. As informações que vão para esta pilha são: o endereço do nó/folha, a quantidade de entradas/bytes que sobrou no nó/folha e a posição i do par (p,c) do próximo nó/folha na descida. */

. Recupera em Nó o nó apontado por objid
. Enquanto não atingir uma folha E o caminho à direita for igual ao caminho à esquerda
    . Decrementa o valor N em todos os c(j), da posição i até o fim do nó
    . Grava Nó
    . Empilha informações na pilha_caminho
    . Recupera o próximo nó/folha no caminho, ainda comum, de descida
. Fim-Enquanto
. Se chegou a uma folha
  então
    . Decrementa o valor N em todos os c(j), da posição i até o fim do nó pai desta folha
    . Empilha informações na pilha_caminho (sobre o pai)
    . Grava o pai
    . Lê a folha
    . Desloca os bytes da folha, eliminando os bytes
    . Empilha informações na pilha_caminho (sobre a folha)

```
    senão
      . O último nó acessado é o UAC (último antecessor
          comum)
      . Salva o contexto dos caminhos à direita e à esquerda
      . desaloca as sub-árvores existentes no UAC entre os
          caminhos à dir e a esq
      . desloca as entradas do UAC, retirando os pares (p,c)
          das sub-árvores de desalocadas
      . Empilha informações na pilha_caminho (sobre o UAC)
      . Grava o UAC
      . Desce-Sub-árvore-à-Esquerda
      . Desce-Sub-árcore-à-Direita
      . Reencadeia o último nível da árvore
. Fim-Se
. Rebalancear_Árvore

Fim-Eliminar


Desce-Sub-árvore-à-Esquerda

. Restaura o contexo do caminha à esquerda
. Enquanto não chegar a uma folha
      . Recupera o próximo nó
      . desaloca as sub-arvores apontadas pelos p(j)'s que
        estão à direita do par (p, c) do caminho de descida
      . Atualiza o nó
      . Grava o nó
      . Empilha informações na pilha_caminho (sobre o nó)
. Fim- Enquanto
. Empilha informações na pilha_caminho (sobre a folha)

Fim-Desce-Sub-árvore-à-Esquerda

Desce-Sub-árvore-à-Direita

. Restaura o contexo do caminha à direita
. Enquanto não chegar a uma folha
      . Recupera o próximo nó
      . desaloca as sub-arvores apontadas pelos p(j)'s que
        estão à esquerda do par (p, c) do caminho de descida
      . Atualiza o nó, deslocando os pares (p, c) para a
          esquerda
      . Grava o nó
      . Empilha informações na pilha_caminho (sobre o nó)
. Fim- Enquanto
. Lê a folha
. Desloca os bytes desta para a esquerda
. Grava a folha
. Empilha informações na pilha_caminho (sobre a folha)

Fim-Desce-Sub-árvore-à-Direita
```

```
Rebalancear_Árvore

/*  A  subrotina  "Eliminar"  pode  deixar  a  árvore
desbalanceada. Esta  rotina rebalanceia a árvore fundindo ou
redistribuindo as  entradas/bytes de  nós/folhas que estejam
em perigo.  Estar em perigo significa ou já ter implodido ou
não poder correr o risco de perder um filho sequer, sob pena
de implodir.  A pilha_caminho criada na subrotina "Eliminar"
é usada para costurar a árvore em um único passo */

. Calcula  para toda a pilha_caminho  se os nós/folhas estão
em perigo,  criando uma  estrutura de  dados  "perigo",  que
contém verdadeiro ou falso para cada nó/folha do caminho
. Enquanto a raiz estiver em perigo
  . Caso
     a raiz tenha um filho único, do tipo folha
          . árvore mínima, já rebalanceada (retorna)
     a raiz tenha um filho único, do tipo nó
          . este é a nova raiz (árvore implode)
          . atualiza pilha_caminho
     a raiz tenha 2 ou 3 filhos
          . Funde_Redistribui estes filhos
          . Atualiza perigo em relação a estes filhos
  . Fim-Caso
  . Atualiza perigo em relação a raiz
. Fim-Enquanto
. Enquanto não chegar ao fim de pilha_caminho
  . Enquanto o filho à esquerda estiver em perigo
     (máximo de 2 iterações)
     . Funde_Redistribui nó/folha esquerdo do topo da
                                          pilha_caminho
     . Atualiza perigo em relação a este nó/folha
  . Fim-Enquanto
  . Se o caminho à direita é diferente do à esquerda
     então
          . Se o filho à direita estiver em perigo
             então
               . Funde_Redistribui nó/folha direito do topo
                         da pilha_caminho
               . Este nó/folha sai de perigo
          . Fim-Se
   . Fim-Se
. Fim-Enquanto

Fim-Rebalancear-Árvore
```

Observações:

1) No algoritmo, observa-se que para o tratamento do caminho à esquerda existe a necessidade de uma interação, que pode ocorrer, no máximo, duas vezes, para tirar o nó/folha de perigo. Esta situação só ocorre quando o nó/folha for filho do UAC, e for fundido com seu irmão à direita, que, por fazer obrigatoriamente parte do caminho de eliminação à direita, pode estar com muito poucas entradas. Este é o único caso que pode causar o "nascimento" de um nó fundido ainda em perigo, necessitando, assim, mais uma operação de fusão/redistribuição. Nos outros casos, por definição, somente 1 operação de fusão/redistribuição é suficiente para tirar um nó/folha de perigo. Este problema não ocorre no caminho à direita simplesmente porque o caminho à esquerda é tratado primeiro.

2) A condição de perigo é calculada pelos seguintes parâmetros:

2.a) Folhas.

Uma folha está em perigo se tiver menos que nfmin bytes. Normalmente, nfmin é o maior inteiro, maior ou igual que o tamanho da folha / 2.

2.b) Nós.

Um nó está em perigo quando:

2.b.1) Raiz.

- tem 1 entrada somente ou;
- tem 2 entradas e um de seus filhos (necessariamente o do caminho) está em perigo ou;
- tem 3 entradas, é o UAC e dois filhos (necessariamante o caminho à esquerda e à direita) estão em perigo.

2.b.2) Outros nós.

- tem menos que nnmin entradas ou;
- tem nnmin entradas e o filho (na pilha_caminho) está em perigo (se for o UAC, basta 1 filho estar em perigo) ou;
- nnmin + 1 entradas, é o UAC e ambos os filhos estão em perigo.

Normalmente, nnmin é o menor inteiro, menor ou igual que o número de entradas no nó / 2.

Nnmin e nfmin podem ser alterados, conseguindo-se assim um ajuste na taxa de ocupação de um nó/folha.

Funde-Redistribui

```
/* Esta subrotina funde ou redistribui um nó/folha (o que
estiver no topo da pilha_caminho) em perigo com um de seus
irmãos. O algoritmo trabalha com 3 entidades: o pai, o fil1
e o fil2. Por definição fil1 está sempre à esquerda de fil2.
Normalmente, o nó/folha que se deseja fundir/redistribuir
fica como fil1, a menos que este nó já seja o extremo
direita do pai */

. Atribui os papéis de pai, fil1 e fil2
. Se a entidade for folha
    então
      . Se a soma da quantidade de bytes de fil1 e fil2
                  couber em uma só folha
          então
            . Funde fil2 em fil1
          senão
            . Se fil1 é a folha em perigo
                então
                    . Redistribui bytes de fil2 para fil1
                então
                    . Redistribui bytes de fil1 para fil2
            . Fim-Se
    senão
      . Se a soma da quantidade de entradas de fil1 e fil2
                  couber em uma só nó
          então
            . Funde fil2 em fil1
          senão
            . Se fil1 é o nó em perigo
                então
                    . Redistribui entradas de fil2 para fil1
                então
                    . Redistribui entradas de fil1 para fil2
            . Fim-Se
. Fim-Se
. Atualiza pilha caminho
```

Fim-Funde-Redistribui

Exemplo:

Tomando-se por base o exemplo na figura 5.2, eliminar 41 bytes a partir do byte 39 (ou seja, até o byte 79 inclusive).

. O nó R é recuperado. Este nó é o UAC, pois o caminho para o byte 39 é através do nó B (caminho à esquerda) enquanto o caminho para o byte 79 é através do nó I (caminho à direita).

. Desce-Sub-árvore à Esquerda:

. O nó B é recuperado. O primeiro par (p,c) fica com os valores (E, 6) enquanto o segundo par é logicamente eliminado, desalocando-se a sub-árvore que inicia no nó G (isto é: o nó G e as folhas j e k).

. O nó E é recuperado. O primeiro par (p,c) fica com os valores (c, 6) enquanto o segundo e terceiro pares são logicamente eliminados, desalocando-se as folhas f e i.

. Desce-Sub-árvore à Direita:

. O nó I é recuperado. Os contadores dos pares (p,c) são atualizados para refletir a eliminação. Não há necessidade de desalocar sub-árvores, nem deslocar pares (p, c) pois, o caminho é, por acaso, o par da extrema esquerda do nó

. O nó H é recuperado. Os contadores dos pares (p,c) são atualizados para refletir a eliminação. Não há necessidade de desalocar folhas, nem deslocar pares (p, c) pois, o caminho é, por acaso, o par da extrema esquerda do nó

. A folha l é recuperada, sofrendo um deslocamento para esquerda (e consequente eliminação) de 4 bytes.

. Após estas operações os bytes foram eliminados. A árvore está em um estado válido (isto é, neste ponto, qualquer operação que se realizar funcionará), porém desbalanceada. A figura 5.3 ilustra este estado.

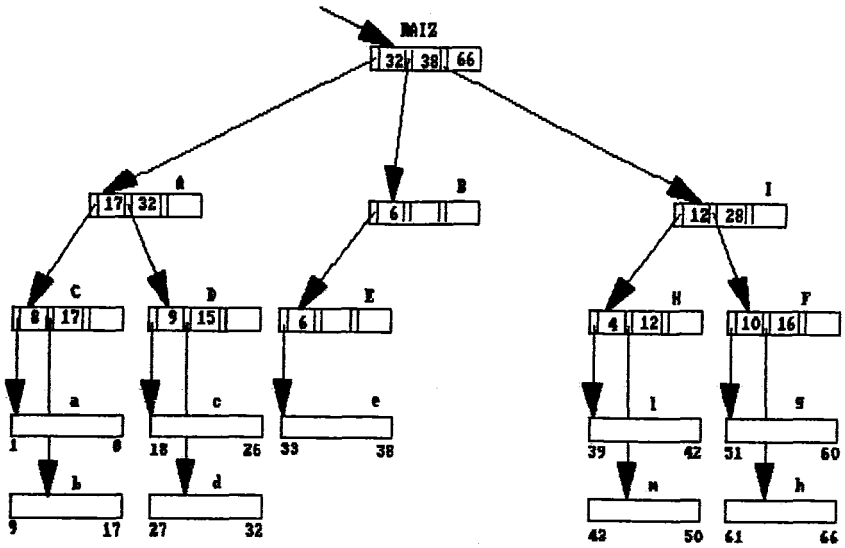


Figura 5.3 - Árvore Desbalanceada após Eliminação.

. Rebalancear-Árvore:

. Calcula nós em perigo, supondo nfmin = 5 e nnmin = 2:

. Caminho à esquerda:
. folha e fora de perigo pois seu número de bytes >= nfmin
. Nó E em perigo pois seu número de entradas < nmin
. Nó B em perigo pois seu número de entradas < nmin

. Caminho à direita:
. folha l em perigo pois seu número de bytes < nfmin
. Nó H em perigo pois seu número de entradas = nnmin e seu filho (folha l) também em perigo
. Nó I em perigo pois seu número de entradas = nnmin e seu filho (nó H) também em perigo

. Raiz:
. Raiz R em perigo pois é o UAC, número de entradas = 3 e seus dois filhos (nós B e I) estão em perigo

. Rebalanceamento do Raiz:
. A raiz R força a fusão dos nós B e I. O nó I é desalocado
. A figura 5.4 ilustra este estado

. Rebalanceamento do Resto da Árvore
. Nó E em perigo, funde-se com o nó H. O nó H é desalocado
. A figura 5.5 ilustra este estado
. Folha l em perigo se redistribui com m
. A figura 5.6 ilustra o estado final da árvore

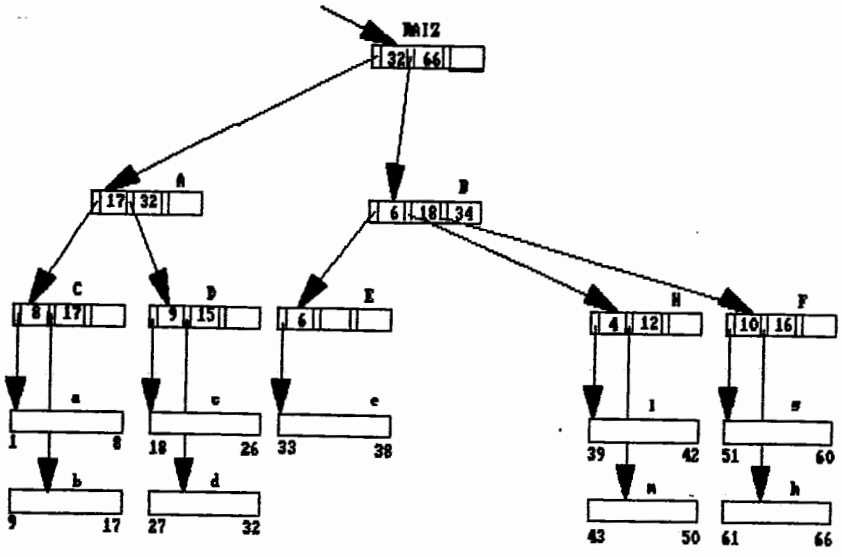


Figura 5.4 - Rebalanceamento da Árvore (1).

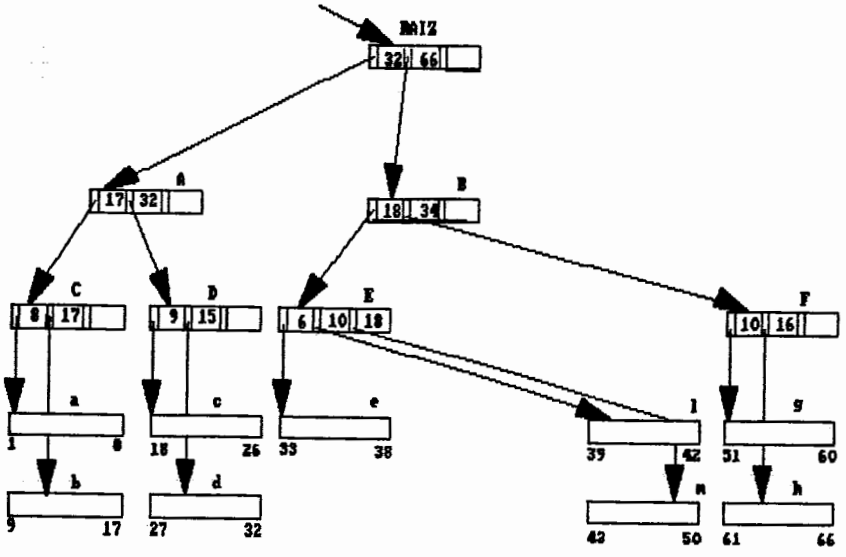


Figura 5.5 - Rebalanceamento da Árvore (2).

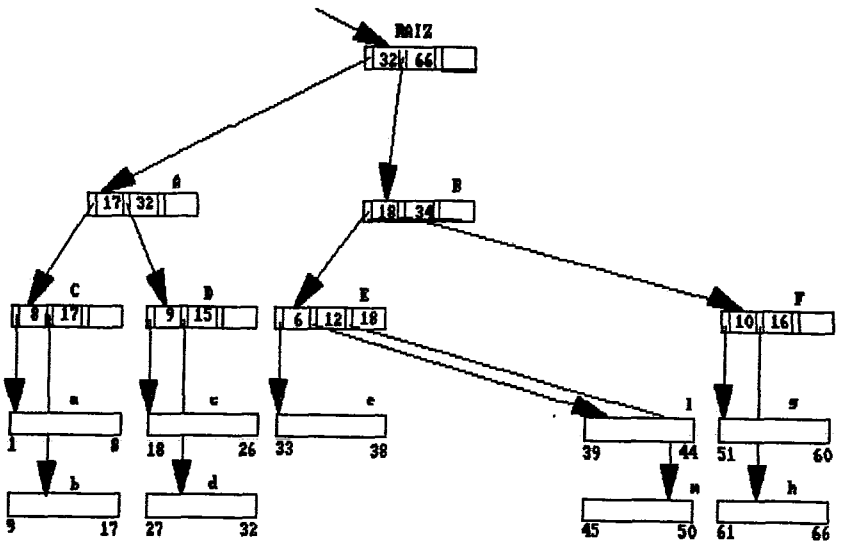


Figura 5.6 - Aspecto Final da Árvore.

V.4) Rótulos.

Embora a implementação de campos longos proveja uma abstração de uma sequência ordenada de bytes, onde um pedaço de qualquer tamanho e em qualquer posição pode ser trocado por outro pedaço, permitindo flexibilidade, com eficiência, no armazenamento de diversos tipos de informação, algumas vezes é necessário a manipulação destas informações de uma forma mais simbólica e menos sujeita a alterações no próprio campo.

Por exemplo: suponhamos um campo longo que esteja armazenando operações gráficas, representando um esquema de um DFD, aonde na posição 10.000 começam uma série de operações que correspondem à representação de uma figura componente do DFD, a um depósito de dados chamado NOTA_FISCAL. Se por acaso o campo longo sofrer modificações que alteram o seu tamanho antes do byte 10.000, a representação deste será deslocada da sua posição original.

Para resolver este problema, uma solução, por nós introduzida, permite associar um rótulo "NOTA_FISCAL" ao

endereço 10.000, introduzindo-se um certo nível de independência física.

Pode ser usado um rótulo, ao invés do número do byte, na manipulação do campo longo, via modalidade básica (vide 5.2.2) ou interativa (vide 5.2.3).

Por razões de consistência lógica, todos os rótulos existentes em um intervalo de bytes eliminado são, também, eliminados.

Novas operações LOPPEREL foram propostas para manipulação de rótulos.

V.3.4.1) Associação de Rótulos

Sintaxe:

CRIAR ROTULO nome-rótulo EM
nome-relação nome-campo (posição)
[TAL QUE condição]

Semântica:

Associa nome-rótulo à posição do campo longo nome-campo nos registros de nome-relação que satisfaçam à condição. Se a cláusula TAL QUE for omitida, o rótulo é associado ao último registro acessado em nome-relação através do comando PROXIMO REGISTRO (vide 4.3). Se nenhum comando PROXIMO REGISTRO foi emitido para nome-relação desde o início da transação, então o rótulo é associado para todos os registros de nome_relação.

O nome_rótulo é um identificador, e é único para um determinado campo longo de um registro. Posição é um inteiro. Um erro é retornado se a posição for maior que o tamanho do campo longo.

V.3.4.2) Eliminação de Rótulos

Sintaxe:

ELIMINAR ROTULO nome-rótulo EM
nome-relação nome-campo
[TAL QUE condicão]

Semântica:

Elimina nome-rótulo do campo longo nome-campo dos registros que satisfaçam à condição. Se a cláusula TAL QUE for omitida, o rótulo é eliminado do último registro acessado em nome-relação através do comando PROXIMO REGISTRO (vide 4.3). Se nenhum comando PROXIMO REGISTRO foi emitido para nome-relação desde o início da transação, então o rótulo é eliminado de todos os registros de nome_relação. Um erro é retornado se nome_rótulo não existir.

Exemplo:

```
CRIAR ROTULO nota_fiscal EM
processos rep_graf (6)
TAL QUE p_no = 1.1

CRIAR ROTULO fim_nf EM
processos rep_graf (10)
TAL QUE p_no = 1.1

SELECIONAR processos
TAL QUE repr_graf (nota_fiscal:fim_nf) = "w0000"
EM $AUX

MOSTRAR $AUX COM p_no, repr_graf (1:18)

1.1  w3333w0000p9999w55

MODIFICAR PROCESSOS
TAL QUE p_no = 1.1
     (repr_graf (1:5) PARA "w1111p2222")
```

```
SELECIONAR processos
TAL QUE repr_graf (nota_fiscal:fim_nf) = "w0000"
EM $AUX

MOSTRAR $AUX COM p_no, repr_graf (1:18)

1.1  w1111p2222w0000p9999w55
```

V.3.4.3) Indicações Para Implementação.

Para a implementação eficiente do sistema de rótulos torna-se necessário uma modificação nas estruturas de dados internas.

O COPPEREL manterá uma nova relação no esquema conceitual (TAB_RÓTULOS) que contém, basicamente, 3 campos:

- o primeiro identifica, univocamente, o rótulo em relação aos campos longos de toda a base, contendo a concatenação do nome da relação, do nome do campo e do valor da chave primária;

- o segundo contém o endereço físico da folha onde se encontra o byte associado ao rótulo;

- o terceiro contém o deslocamento do rótulo em relação ao início da folha.

Quando o usuário realiza uma operação de associação de rótulo, o sistema computa o endereço físico da folha correspondente ao byte lógico e o valor de seu deslocamento na folha, armazenando mais um registro em TAB_RÓTULOS.

Através do armazenamento do endereço físico e não do byte lógico ao qual o rótulo foi associado, a manutenção do mecanismo de rótulos fica livre do custo de atualizar todos os rótulos que existam depois de um byte B quando o campo sofre uma inserção ou eliminação antes de B.

A relação TAB_RÓTULOS é indexada tanto pelo campo que identifica o rótulo quanto pelo campo que contém o endereço físico da folha. Assim, embora inserções ou eliminações de bytes possam influenciar os rótulos existentes na folha que sofreu a operação, estes podem ser localizados e ter seu deslocamento atualizado de forma bastante eficiente.

Para que possam ser feitas referências a intervalos de bytes entre rótulos é necessário que cada nó ou folha da árvore B aponte para o seu pai. Operações de busca de intervalos de bytes não precisam subir até a raiz da árvore, uma vez que existe uma lista encadeada no último nível de nós.

CAPÍTULO VI

EXTENSÕES NO COPPEREL PARA MANIPULAR OBJETOS COMPLEXOS.

Neste capítulo, descrevemos uma proposta para incorporação de objetos complexos no SGBD COPPEREL. São apresentados o modelo e a funcionalidade alcançados, bem como a sintaxe detalhada dos novos comandos a serem adicionados a LOPEREL para a manipulação adequada destes objetos. O exemplo do objeto DFD, usado no capítulo dois, é desenvolvido ao longo deste capítulo para melhor ilustrar a aplicação dos novos comandos.

VI.1) Premissas.

Existe uma variedade de propostas na literatura para a manipulação e implementação de objetos complexos (vide seção 2.2). Na seção 3.1, analisamos alguns requisitos de funcionalidade desejáveis para SGBD's que devam dar suporte a aplicações não convencionais. Entre eles, o suporte a objetos complexos foi estudado, apontando-se uma série de aspectos que podem, em maior ou menor grau, serem adaptados a uma determinada proposta. Os requisitos de funcionalidade para objetos complexos, no COPPEREL, são um subconjunto destes. Nesta seção, algumas premissas básicas são levantadas, justificando-se a proposta desenvolvida ao longo das próximas subseções.

a) Objetos Complexos como Extensão do Modelo Relacional.

O SGBD COPPEREL deve ser adaptado sem perder sua característica de ser relacional. Neste sentido, algumas decisões foram tomadas:

- Um objeto complexo é, estruturalmente, um conjunto de tuplas em relações diferentes, que podem ser manipuladas

como relações normais nas operações de consulta. As operações usuais de inclusão e exclusão de registros não podem ser usadas, para estas relações componentes, pois o usuário deve ser sabedor dos efeitos colaterais (como eliminação em cascata) que operações desta natureza podem causar;

- A linguagem de manipulação do objeto complexo deve ser uma extensão da LOPEREL, sendo fornecidas operações de alto nível. Granularidade variável (vide 3.1) e navegação no objeto também podem ser obtidas (vide 6.3.3);

- O usuário continua usando o conceito de chave primária em qualquer relação componente do objeto. O uso de identificadores internos, necessários para implementação, é transparente para ele;

- A estrutura, operações e restrições de integridade de um objeto complexo são controladas através de mecanismos genéricos (enfoque estrutural). O enfoque operacional, usualmente implementado via uma facilidade para tipos abstratos de dados, não é tratado nesta proposta, não sendo, consequentemente, obtido o enfoque comportamental.

b) Recursividade.

Um mecanismo, ainda que simples, para representar objetos que são compostos por partes do mesmo tipo deles mesmos, deve ser fornecido pelo sistema, uma vez que tais casos não são raros na prática.

c) Sub-objetos Compartilhados.

O sistema deve permitir que um sub-objeto (representado por uma tupla em uma relação componente) seja possuído por mais de um objeto pai, isto é, o relacionamento entre objeto-pai e objeto-componente deve poder ter cardinalidade N:M, ao invés de, mais tradicionalmente, 1:N. Mais ainda, dois (ou mais) pais de um sub-objeto compartilhado, podem ou não, pertencer ao mesmo objeto complexo.

d) Estrutura do Objeto Complexo é um Grafo Acíclico.

A maioria das propostas para objetos complexos permite a representação de uma hierarquia estrita. Na extensão do COPPEREL, um sub-objeto pode ser filho de mais de um objeto (relações diferentes), formando um grafo que não pode ter ciclos.

A premissa a constitui, na verdade, uma limitação imposta pelo modelo, enquanto as premissas b, c e d são as funcionalidades que a extensão deseja alcançar. Estas funcionalidades foram escolhidas a partir de uma série de necessidades de modelagem normalmente encontradas no mundo real, contrabalanceadas com as dificuldades de implementação. A intenção é obter uma proposta que seja, ao mesmo tempo, útil e implementável.

VI.2) Declaração do Objeto Complexo.

Um objeto complexo é declarado a partir das suas relações componentes. O comando LOPEREL para criar relações foi estendido de forma que se possa declarar, opcionalmente, para uma relação sendo criada, que esta é a raiz ou um componente de um objeto complexo. A razão disto é que as extensões sintáticas para declarar objetos complexos são mínimas. Não é necessário criar atributos em relações componentes que apontem para a relação pai. As ligações de composição do objeto complexo são transparentes para o usuário. Os atributos de todas as relações que compõem o objeto complexo devem ter nomes únicos entre eles.

A sintaxe para criar relações é:

```
criar_arquivo :==

                CRIAR ARQUIVO [$]nome_arquivo cabeçalho

cabeçalho :==

     COM inteiro REGISTROS DE inteiro ATRIBUTOS:
     {nome_atributo tipificação},,,
     COM CHAVE: {nome_atributo_chave},,,
     [E COM INDICE SOBRE: {nome_atr_indice},,,]
     [{ RAIZ                                          }]
     [{ RAIZ RECURSIVA [COMPARTILHADA]                }]
     [{ {COMPONENTE [COMPARTILHADO] DE
           {nome_arquivo_pai},,,},,,                  }]
```

As cláusulas em negrito correspondem às modificações propostas na sintaxe para declarar objetos complexos.

Analisamos cada uma das três cláusulas opcionais a seguir:

a) RAIZ

Uma relação, declarada como raiz, funciona como a raiz do objeto complexo, a partir de onde uma relacao componente pode ser, subsequentemente, declarada. Nesta opção, um objeto não pode ser formado de partes do mesmo tipo, sendo proibido aplicar a operação de seleção recursiva definida em 6.3.3.

b) RAIZ RECURSIVA [COMPARTILHADA]

Neste caso, o objeto complexo pode ser formado de partes dele mesmo, podendo ser aplicada a operação de seleção recursiva. Se a palavra COMPARTILHADA for omitida, então um sub-objeto pode ter, no máximo, um pai.

c) [COMPONENTE [COMPARTILHADO] DE
{nome_arquivo_pai},,,},,,

Esta cláusula declara uma relação como componente de uma ou mais relações, permitindo a criação de um grafo. O relacionamento de composição entre a relação sendo criada e

qualquer um de seus pais pode ser COMPARTILHADO ou não. Se for, significa que uma instância da relação, sendo declarada, pode ser sub-objeto de mais de uma instância na relação pai. A sintaxe permite que o compartilhamento exista em relação a qualquer um dos pais, independentemente um do outro.

Para evitar ciclos no grafo, o nome_arquivo_pai deve ser uma relação previamente criada e que, também, seja uma relação raiz ou dependente.

Se uma relação for componente de mais de um nome_arquivo_pai, estes devem ser dependentes (direta ou indiretamente) da mesma relação raiz. A razão disto é evitar maior complexidade para garantir a integridade de cada esquema de objeto complexo, nas operações de inclusão e/ou eliminação de registros em relacões componentes.

Vejamos um exemplo completo da criação do objeto DFD. Sua representação gráfica se encontra na figura 6.1. A raiz e as partes componentes são representadas por relações. Setas direcionadas de pai para filho representam relacionamentos de composição. Um C, grafado ao lado de uma seta, indica que o componente filho é compartilhado.

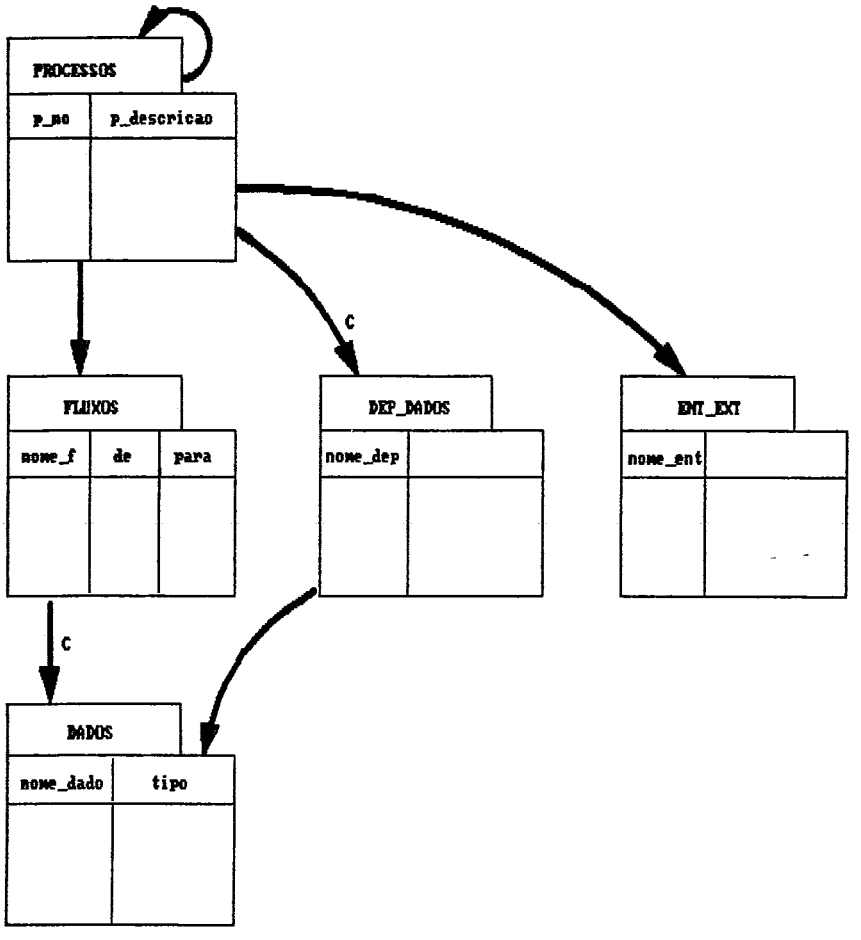


Figura 6.1 - Objeto DFD no COPPEREL Estendido.

A declaração deste objeto fica:

```
CRIAR ARQUIVO PROCESSOS COM ...
COM CHAVE: P_NO
RAIZ RECURSIVA

CRIAR ARQUIVO FLUXOS COM ...
COM CHAVE: NOME_F
COMPONENTE DE PROCESSOS

CRIAR ARQUIVO DEP_DADOS COM ...
COM CHAVE: NOME_DEP
COMPONENTE COMPARTILHADO DE PROCESSOS

CRIAR ARQUIVO ENT_EXT COM ...
COM CHAVE: NOME_ENT
COMPONENTE DE PROCESSOS

CRIAR ARQUIVO DADOS COM ...
COMO CHAVE: NOME_DADO
COMPONENTE DE DEP_DADOS,
COMPONENTE COMPARTILHADO DE FLUXOS
```

No exemplo acima, a raiz é recursiva, indicando que um processo pode ser formado por outros processos, mas um processo filho não pode pertencer a mais de um pai.

Um DEP_DADOS pode ser componente de mais de um PROCESSO. Como este é a raiz do objeto, temos aqui um caso de sub-objetos compartilhados, necessariamente, por objetos diferentes. Por outro lado, um DADO pode aparecer em mais de um fluxo. Como FLUXOS não é uma relação raiz, podemos ter compartilhamento entre objetos diferentes ou dentro do mesmo objeto. O sistema não distingue entre estes dois tipos de compartilhamento como um passo na direção de ser obter granularidade e relativismo: sob o ponto de vista de FLUXOS (e não de PROCESSOS), um DADO compartilhado é sempre um caso de sub-objeto compartilhado por objetos diferentes.

A necessidade de não fazer distinção entre os dois casos de compartilhamento é mais notada quando acontece na recursividade de uma relação raiz. A figura 6.2 ilustra esta situação. Um curso é composto (pré-requisito) por outros cursos. Um curso pode ser pré-requisito de mais de um curso.

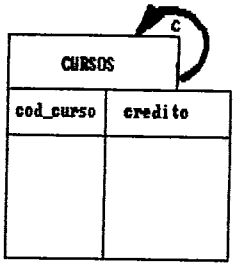


Figura 6.2 - Objeto COMPARTILHADO com RECURSIVIDADE.

A figura 6.3 ilustra a representação de algumas instâncias de cursos. Sob o ponto de vista da instância FIS III, o curso CALC I é um sub-objeto compartilhado internamente, enquanto sob o ponto de vista de FIS II (ou CALC II), é um sub-objeto compartilhado entre objetos diferentes.

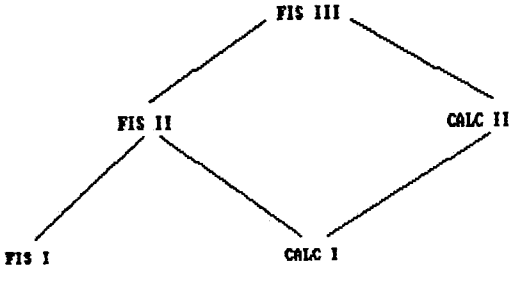


Figura 6.3 - Instâncias do objeto CURSOS.

VI.3) Manipulação do Objeto Complexo.

VI.3.1) Construção de Instâncias.

Para a construção/reconstrução de um objeto complexo estão disponíveis operações de inserção e eliminação de registros, similares às já existentes na LOPEREL. Como o modelo permite sub-objetos compartilhados e sub-objetos com

mais de uma relação pai, são providas novas operações para conectar e desconectar componentes. A sintaxe e a semântica destas operações são as seguintes (A sintaxe da LOPEREL completa, incluindo extensões, está no apêndice A).

a) Inserção de registros em relações componentes.

Sintaxe:

```
                  { DE nome_arq }
INSERIR PARTES    { SEGUINTES   } EM nome_arq_componente
                  { GRAVADOS    }

[COMO COMPONENTE DE {condição EM nome_arq_pai},,,]
```

Semântica:

Insere registros oriundos de nome_arq, do terminal (ou da interface para linguagem hospedeira) ou de um arquivo externo no nome_arq_componente. Este pode ser uma relação componente ou raiz com recursividade de um objeto complexo. Nome_arq_pai deve ser um dos pais declarados na criação de nome_arq_componente.

A condição resolve para um ou mais registros do nome_arq_pai. Os registros inseridos são ligados como componentes destes. A condição deve obedecer às seguintes regras:

- a condição deve envolver somente atributos do nome_arq_pai, não necessariamente o(s) atributo(s) que compõe(m) a chave primária deste;
- a condição não pode resolver para zero registros, caso contrário, um erro é sinalizado;
- se nome_arq_componente for um componente não compartilhado de nome_arq_pai, a condição deve resolver para somente um registro, caso contrário, um erro é sinalizado.

Não é obrigatório que existam cláusulas para conexão com todos os nome_arq_pai de um nome_arq_componente; mas pelo menos uma é obrigatória. A cláusula só pode ser totalmente omitida se nome_arq_componente for uma relação raiz. Neste caso, o objeto que está sendo inserido não é subordinado a nenhum outro.

Exemplos:

```
INSERIR PARTES SEGUINTES EM PROCESSOS

/1/"processo principal"//

INSERIR PARTES SEGUINTES EM PROCESSOS
COMO COMPONENTE DE p_no = 1 EM PROCESSOS

/1.1/"subprocesso"//

INSERIR PARTES SEGUINTES EM DADOS
COMO COMPONENTE DE
          nome_f = "F1" OU nome_f = "F2" EM FLUXOS,
          nome_dep = "FUNC"              EM DEP_DADOS

/"SALARIO"/"INTEIRO"//
```

b) Eliminação de registros em relações componentes.

Sintaxe:

```
RETIRAR PARTES nome_arq_componente TAL QUE condição
```

Semântica:

O comando é similar ao comando RETIRAR comum da LOPEREL. A diferença é o efeito cascata de eliminação de componentes que ocorre. Nome_arq_componente pode ser o nome da relação raiz ou de qualquer um de seus componentes. A eliminação em cascata termina quando um sub-objeto é compartilhado ou tem mais de um pai em outra relação que não foi eliminado.

Para melhor entendermos o efeito de propagação da eliminação, principalmente nos casos citados acima, fornecemos o algoritmo abaixo:

ALGORITMO Propagação De Eliminação

Problema: Dado uma relação F, componente de relacões P1, P2,... Pn, eliminar, pelo efeito de propagação, as instâncias f's de F, quando ocorrer eliminação das intâncias p's de Pj.

Algoritmo:

. Eliminar a materialização da composição entre p's e f's, formando uma lista L de filhos f's desconectados

. Se F é compartilhado em relação a Pj
  então
    . Eliminar f's duplicatas de L
    . Eliminar f's de L que tiverem conexão para outros elementos de Pj não pertencentes a p's

. L(Pj-F) <- L

. Se todos os Pi já foram tratados
  então
    . Lfinal <- L(p1-F) ∩ L(P2-F) ∩ ... ∩ L(Pn-F)
    . Elimina de F os f's pertencentes a Lfinal (chamada recursiva)

c) Conexão de Componentes.

Sintaxe:

CONECTAR [REGISTROS DE] nome_arq_componente

TAL QUE condição1

[COMO] COMPONENTE [DE] {condição2 EM nome_arq_pai},,,

Semântica:

Os registros de nome_arq_componente que satisfazem à condição1 são ligados como componentes dos registros de nome_arq_pai que satisfazem à condição2. As restrições da condição2 são as mesmas do ítem a nesta mesma sub-seção.

Um erro é retornado se algum dos registros selecionados pela condição1 já estiverem conectados e o relacionamento de nome_arq_componente e nome_arq_pai não for do tipo compartilhado.

Exemplo:

```
CONECTAR DADOS
TAL QUE nome_dado = "SALARIO"
COMO COMPONENTE DE nome_f = "F3" EM FLUXOS
```

A possibilidade de registros, já incluídos como componentes de um objeto complexo, virem a ser componentes de outros sub-objetos, mais tarde, pode causar um ciclo nas instâncias de um objeto complexo, quando houver raiz com recursividade.

Exemplo: Após as duas primeiras inclusões do exemplo no ítem a desta mesma sub-seção, o seguinte comando causaria este problema.

```
CONECTAR PROCESSOS
TAL QUE p_no = 1
COMO COMPONENTE DE p_no = 1.1 EM PROCESSOS
```

O sistema rejeita este tipo de conexão.

d) Desconexão de Componentes.

Sintaxe:

```
DESCONECTAR [REGISTROS DE] nome_arq_componente

TAL QUE condição1

FORA [DE] {[condição2] [EM] nome_arq_pai},,,
```

Semântica:

Os registros de nome_arq_componente que satisfazem à condição1 deixam de ser componentes dos registros que satisfazem à condição2 em nome_arq_pai. A condição2 deve ser omitida quando o relacionamento entre nome_arq_pai e nome_arq_componente não for do tipo COMPARTILHADO.

Podem ocorrer erros em duas situações, com a operação sendo abortada:

- Se a condição2 resolver para algum registro que não for pai dos registros que satisfazem à condição1;

- Se a desconexão causar que algum registro de nome_arq_componente fique sem nenhum pai (na relacão nome_arq_pai ou em qualquer outra da qual nome_arq_componente seja dependente).

Exemplo:

```
DESCONECTAR DADOS

TAL QUE nome_dado = "SALARIO"

FORA DE nome_f = "F2" EM FLUXOS,
                         DEP_DADOS
```

VI.3.2) Seleção no Objeto Complexo.

Para selecionar objetos, são providas novas operações que permitem a manipulação do mesmo em alto nível. O modo de operação do comando segue o modelo da operação SELECIONAR já existente na LOPEREL, ou seja, são geradas novas relações (temporárias ou não) com os registros que compõem o resultado dos objetos selecionados. Estas operações estão grupadas em um único comando, descrito a seguir.

Sintaxe:

```
SELECIONAR OBJETO [*] nome_arq1

TAL QUE condição

EM [$]prefixo

[[ COMPLETO                                  ]]
[[ [ADICIONANDO] PARTES {nome_arq2},,,       ]]
```

Semântica:

O comando SELECIONAR OBJETO fornece um meio poderoso de selecionar objetos complexos, pois é possível abstrair-se de sua estrutura. É possível gerar, na saída, parte da estrutura do objeto.

Nome_arq1 é o nome de uma relação qualquer que pertença a um objeto complexo, podendo ser a raiz ou qualquer componente. O objeto complexo considerado para resolver a consulta é aquele composto por nome_arq1 e seus descendentes. Desta forma, a condição pode envolver qualsquer atributos que apareçam em nome_arq1 ou nos seus dependentes. Como é obrigatória unicidade de nomes de

atributos dentro de um objeto complexo, nenhuma notação especial é necessária para referenciar a estes nomes.

O sistema realiza, internamente, uma operação equivalente a uma operação de junção entre as relações necessárias para resolver a condição de seleção. Estas relações formam um conjunto R composto, necessariamente, de:

1) Relação nome_arq1;

2) Relações que possuam atributos usados na condição de seleção;

3) Relações necessárias para formar todos os caminhos entre a relação nome_arq1 e as relações obtidas em 2.

A saída gerada é, também, um objeto complexo composto de um conjunto S de relações que dependem da última cláusula do comando SELECIONAR OBJETO:

a) Caso a última cláusula seja omitida, S = R;

b) Caso seja especificado COMPLETO, S é o conjunto de todas as relações que compõem o objeto complexo, isto é, nome_arq1 e seus descendentes;

c) Com a cláusula PARTES, S = P, onde P é o conjunto de relações especificadas em nome_arq2, junto com todas as relações necessárias para formar todos os caminhos entre nome_arq1 e os diversos nome_arq2;

d) Com a cláusula ADICIONANDO PARTES, S = R U P.

Os nomes das relações em S mantêm uma correspondência biunívoca com os nomes das relações originais, acrescidos do prefixo. O sinal $, opcional, já existente na LOPEREL, indica que as relações criadas devem ser temporárias, sendo automaticamente eliminadas ao final da sessão.

As instâncias que participam do conjunto S, são em cada caso, as seguintes:

a) A junção implícita entre as relações em R é realizada, satisfazendo a condição de seleção, criando uma relação virtual J. Esta relação é transformada "de volta" no conjunto de relações S, possivelmente realizando o equivalente a uma operação relacional de projeção, para evitar registros duplicados em qualquer relação de S.

b,c e d) É realizada uma operação similar ao caso 1, obtendo-se a relação de nome prefixo_nome_arq1, que contém um registro raiz para cada objeto complexo selecionado. As instâncias que participam das outras relações do conjunto S são todas as que pertencem a cada objeto complexo selecionado. No primeiro caso, estas instâncias são somente a projeção daquelas que aparecem na relação virtual J.

A operação 1 pode ser vista como uma operação relacional simples de junção, onde, simplesmente, não foi necessário nem fornecer as relações que participam da junção, nem predicados para a operação (equivalente a um equi-join). As outras podem ser vistas como operações de recuperação de objetos - porque todas as instâncias que os compõem são coletadas - mas permitindo a geração de somente parte da estrutura dos objetos. Isto é muito útil pois permite navegação no objeto com bom desempenho (vide 6.3.3).

Vejamos alguns exemplos, a partir da figura 6.4.

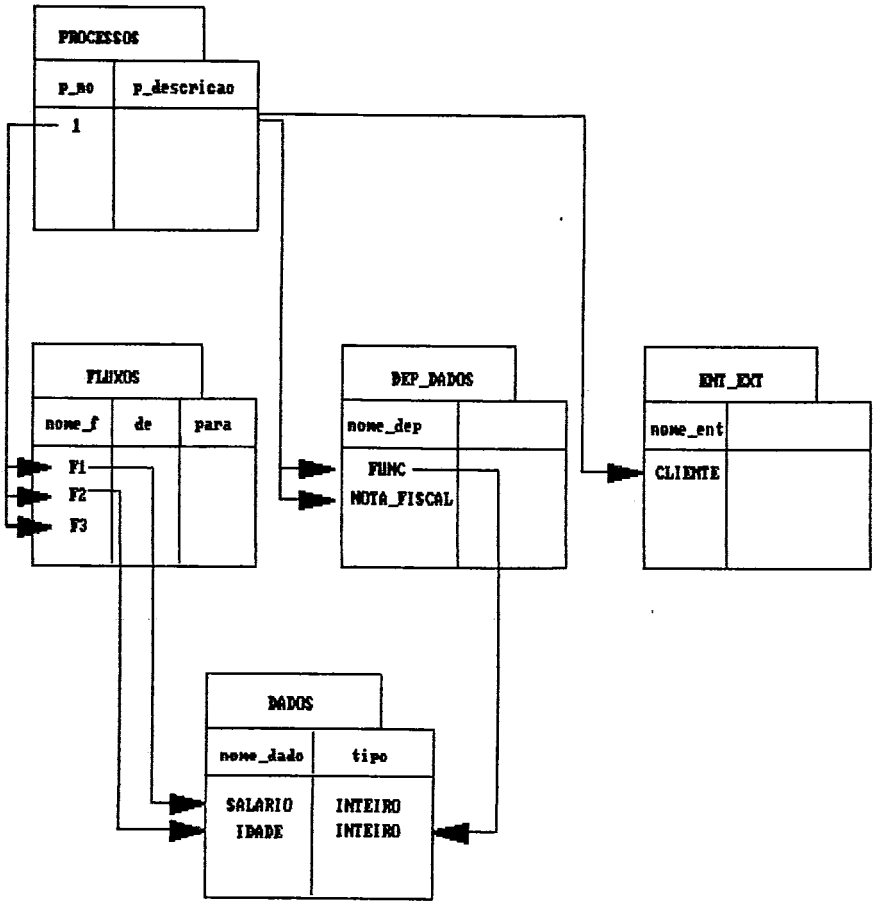


Figura 6.4 - Objeto DFD instanciado.

1) SELECIONAR OBJETO PROCESSOS
TAL QUE tipo = "INTEIRO"
EM $AUX1

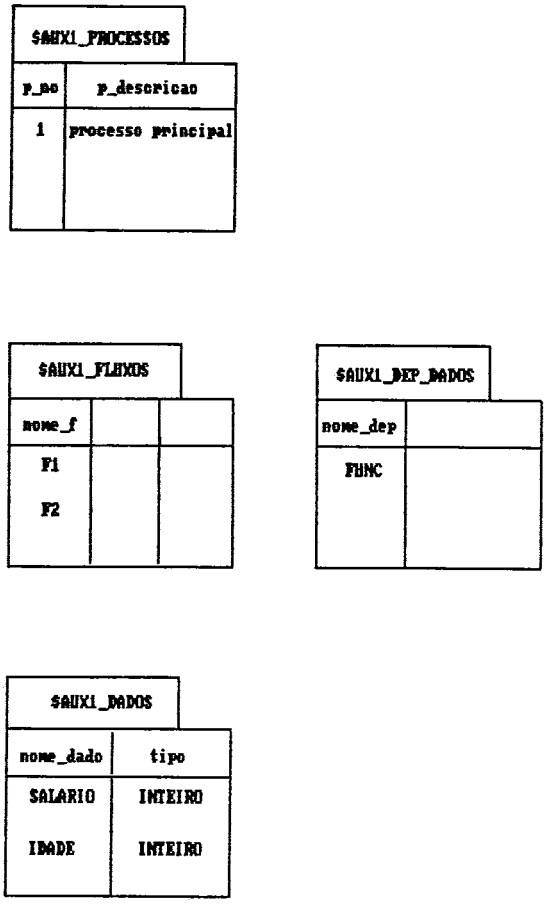
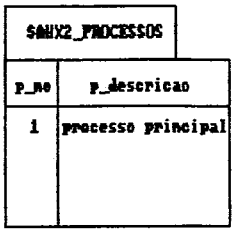
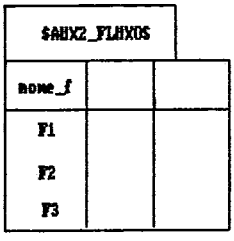


Figura 6.5 - Consulta 1

2) SELECIONAR OBJETO PROCESSOS
   TAL QUE nome_ent = "CLIENTE"
   EM $AUX2
   PARTES FLUXOS



$AUX2_PROCESSOS	
p_no	p_descricao
1	processo principal



$AUX2_FLUXOS		
nome_f		
F1		
F2		
F3		

Figura 6.6 - Consulta 2

```
3) SELECIONAR OBJETO PROCESSOS
   TAL QUE tipo = "INTEIRO"
   EM $AUX3
   COMPLETO
```

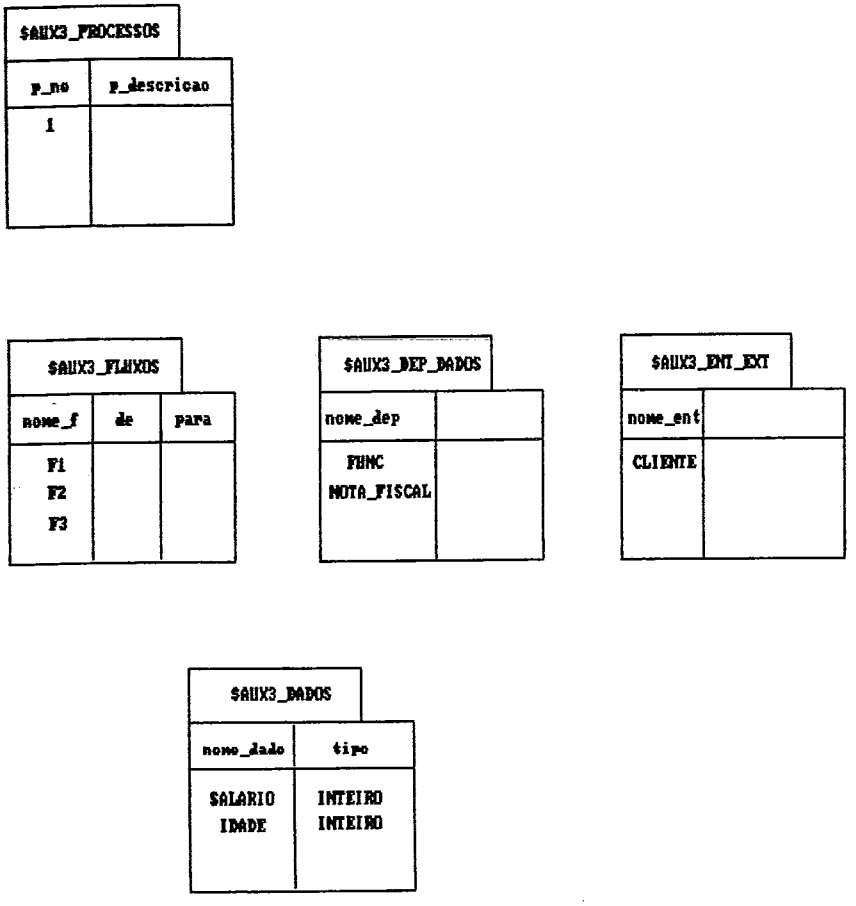


Figura 6.7 - Consulta 3

O operador * permite que cada componente recursivo da raiz seja considerado como um objeto para seleção de registros, realizando o fechamento transitivo. Como os componentes recursivos podem ser compartilhados, cuidado especial deve ser tomado para não considerar o mesmo sub-objeto mais de uma vez. Neste caso, nome_arq1 é, necessariamente, o nome de uma relação raiz declarada como recursiva (COMPARTILHADA ou não).

Um atributo, que aparece na condição, pode ser acompanhado de informação indicando, entre parêntesis, o nível de recursão em que o predicado deve ser avaliado. Se a

informação for omitida, é considerado que o predicado deve ser avaliado sem aplicar o operador de recursão.

O nível de recursão pode ser qualificado por:

1) Uma constante:

```
SELECIONAR OBJETO * PROCESSOS
TAL QUE nome_f(2) = "F9"
EM $AUX
```

2) Uma expressão com variáveis ligadas:

```
SELECIONAR OBJETO * PROCESSOS
TAL QUE nome_f(n) = "F1" E
        nome_f(n+1) = "F8"
EM $AUX
```

3) Variáveis ligadas a um predicado

```
SELECIONAR OBJETO * PROCESSOS
TAL QUE nome_f(n) = "F1" E
        nome_f(m) = "F8" E n < m
EM $AUX
```

4) Quantificador Universal

```
SELECIONAR OBJETO * PROCESSOS
TAL QUE nome_dado(T) = "SALARIO"
EM $AUX
```

5) Quantificador Existencial

```
SELECIONAR OBJETO * PROCESSOS
TAL QUE nome_dado(E) = "IDADE"
EM $AUX
```

Em relação à figura 6.8, as consultas 1, 2 e 3 acima retornariam, na relação $AUX_PROCESSOS, somente o processo de p_no = 1. A consulta 4 retornaria os processos 1.1 e 1.1.1; a consulta 5, os processos 1 e 1.1.

Os componentes da raiz que tiveram que ser recursivamente recuperados para calcular o fecho transitivo não aparecem na saída, a menos que a condição envolva um predicado com eles. Assim, a consulta 6, abaixo, recupera todos os componentes do processo 1, mostrando em $AUX_PROCESSOS os processos 1, 1.1 e 1.1.1.

```
6) SELECIONAR OBJETO * PROCESSOS
   TAL QUE p_no = 1 E p_no(E) <> 0
   EM $AUX
```

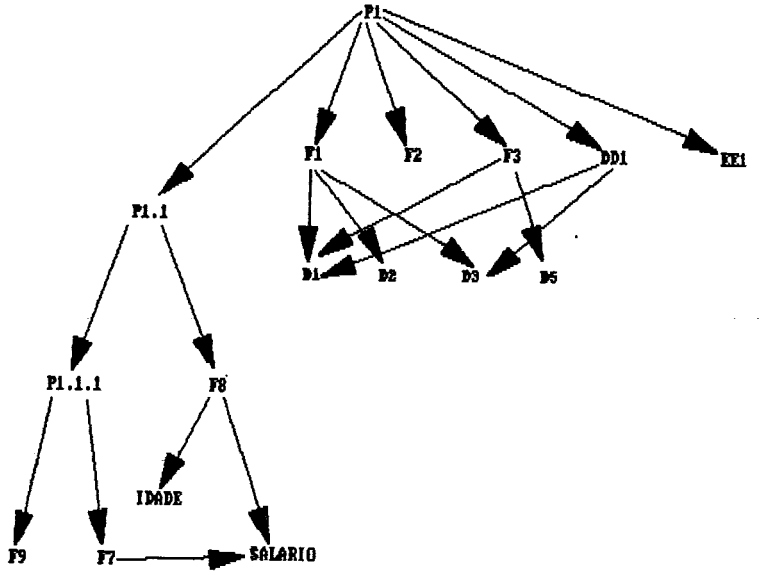


Figura 6.8 - Instâncias de um objeto com recursão.

VI.3.3) Exibição do Objeto Complexo.

Os registros que compõem um objeto complexo podem ser exibidos na tela ou enviados à interface para linguagem hospedeira, nas seguintes formas (a nomencaltura é a mesma de HARDER 1987):

1) Acesso Horizontal

Através do comando MOSTRAR, já existente na LOPEREL, é possível recuperar todos os registros de uma relação componente de um objeto complexo. Todos os registros são exibidos, independentes do objeto a que pertençam. Do mesmo modo, é possível aplicar um comando simples de seleção a uma relação componente já existente, gerando uma relação que não pertence a nenhum objeto complexo.

Exemplo:

```
SELECIONAR FLUXOS
TAL QUE nome_f > "F"
EM $AUX

MOSTRAR $AUX
```

2) Acesso Vertical

Através do comando MOSTRAR OBJETO, obtem-se uma listagem compreensiva dos componentes de um objeto complexo.

Sua sintaxe é:

MOSTRAR OBJETO nome_arq

Onde nome_arq pode ser qualquer relação dependente ou raiz. O exemplo abaixo mostra o formato desta listagem, para um conjunto de instâncias da figura 6.8.

```
SELECIONAR OBJETO PROCESSOS
TAL QUE p_no = 1
EM $AUX
COMPLETO

MOSTRAR OBJETO $AUX_PROCESSO
```

```
Esquema PROCESSOS

     PROCESSOS (p_no, descrição)          recursivo, raiz
       FLUXOS (nome_f, ........)                    , 1 super.
          DADOS (nome_dado, tipo_dado)     comp.   , 2 super.
       DEP_DADOS (nome_dep, ...)           comp.   , 1 super.
          DADOS (nome_dado, tipo_dado)             , 2 super.
       ENT_EXT (nome_ent, .....)                   , 1 super.

Instâncias $AUX_PROCESSOS

     PROCESSO..... 1 processo principal

     sub_PROCESSOS neste objeto:

     1.1
         1.1.1

     FLUXOS........ F1
     DADOS..... D1 INTEIRO _____________(2), 2/2
     DADOS..... D2 INTEIRO _____________(1), 1/2
     DADOS..... D3 REAL    _____________(1), 2/2
     FLUXO..... F2
     FLUXO..... F3
     DADOS..... D1 INTEIRO _____________(2), 2/2
     DADOS..... D5 INTEIRO _____________(1), 2/2
     DEP_DADOS..... DD1 _________________________(1)
     DADOS..... D1 INTEIRO _____________    , 2/2
     DADOS..... D3 REAL    _____________    , 2/2
     ENT_EXT....... EE1
```

Primeiro, algumas informações sobre o esquema do objeto complexo são exibidas. A partir da identação e das indicações ao lado do nome de cada relação, é possível ter uma idéia de todo o esquema do objeto: que relações o compõem, que atributos existem em cada uma, quantos superiores uma relação tem e se o relacionamento com ele é compartilhado ou não. A partir daí vem a listagem dos registros. A coluna da direita indica a relação a qual o registro pertence. A identação representa o relacionamento de composição. Os números, entre parêntesis, denotam a quantidade de instâncias superiores que um registro tem em um relacionamento compartilhado. Por exemplo: o dado D1 tem (2) fluxos superiores. O mesmo dado D1 quando listado sob o contexto de DEP_DADOS aparece sem o indicador, pois o relacionamento entre DADOS e DEP_DADOS não é do tipo compartilhado. Os números separados por barra indicam, respectivamente, o número de ligações com pais de entidades diferentes e o número máximo possível, limitado pelo

esquema. Por exemplo: os dados D1 e D3 são ligados ao fluxo F1 e ao depósito DD1. Já o dado D5 encontra-se ligado ao fluxo F3 e a algum depósito de dados não pertencente ao processo de p_no = 1. Para saber que depósito é este, poderiam ser emitidos os comandos:

```
SELECIONAR OBJETO DEP_DADOS
TAL QUE nome_dado = "D5"
EM $AUX2

MOSTRAR $AUX2_DEP_DADOS
     DD6
```

Ou ainda:

```
MOSTRAR OBJETO $AUX2_DEP_DADOS
```

```
Esquema DEP_DADOS

       DEP_DADOS (nome_dep, ...)                    , 1 super.
          DADOS (nome_dado, tipo_dado)              , 2 super.
Instâncias $AUX2__DEP_DADOS

     DEP_DADOS..... DD6
     DADOS..... D5 INTEIRO _____________   , 2/2
     DADOS..... D9 REAL    _____________   , 1/2
```

E ainda, se quisermos saber que processo compartilha o dado D5 com o processo de p_no = 1, faríamos:

```
SELECIONAR OBJETO PROCESSOS
TAL QUE nome_dado = "D5"
EM $AUX3

MOSTRAR $AUX3_PROCESSOS
     1 processo principal
     7 inclusao de func.
```

3) Navegação.

O comando de seleção de objeto complexo mostrado no ítem 6.3.2, acompanhado do comando LOPEREL MOSTRAR, já existente anterirormente, fornece um meio eficiente de navegação no objeto complexo. Estes comandos podem ser usados através da interface para linguagens hospedeiras (capítulo 4), permitindo a construção de uma interface amigável com o usuário.

Uma parte de um objeto complexo pode ser vista como uma relação R com atributos Ai (i=1,i) que pode ser dependente de um conjunto de pais Pj (j=1,n) e, também, possuir dependentes Fk (k=1,m).

Dado um registro r de R que obedeça a uma condição envolvendo um atributo Ai, para navegar para obter o conjunto de instâncias que compõem um de seus pais Pj, emitem-se os comandos:

```
SELECIONAR OBJETO PJ
TAL QUE Ai = expressão
EM $AUX

MOSTRAR $AUX_PJ
```

Para navegar para um de seus filhos Fk, emitem-se os comandos:

```
SELECIONAR OBJETO R
TAL QUE Ai = expressão
EM $AUX
PARTES FK

MOSTRAR $AUX_FK
```

Os comandos podem ser realizados eficientemente, uma vez que somente os registros que interessam são recuperados.

VI.4) Restrições de Integridade.

As restrições de integridade suportadas pela extensão do COPPEREL, para objetos complexos, estão ligadas à própria estrutura possível de ser representada, sendo garantidas, de uma forma implícita, na hora da aplicação de alguma operação da LMD estendida. Estas restrições já foram mencionadas quando apresentamos a semântica dos novos comandos. Nesta seção, simplesmente organizamos as idéias de outra forma: as restrições são enunciadas, listando-se, logo após, os comandos responsáveis por garanti-las. São elas:

1) Uma instância de uma relação dependente deve estar ligada a pelo menos uma instância de pelo menos uma relação pai. Os comandos abaixo a garantem:

1.a) RETIRAR PARTES - Eliminar um registro pai causa a eliminação recursiva de seus dependentes, a menos que estes estejam ligados a outro(s) pai(s) (na mesma ou em outra relação).

1.b) DESCONECTAR - O sistema não permite desconectar o último vínculo de um registro dependente com algum pai.

1.c) INSERIR PARTES - Uma nova instância de uma relação dependente é ligada, automaticamente, a pelo menos uma instância de uma relação pai.

1.d) INSERIR - O comando já existente na LOPEREL não pode ser usado para inserir registros em relações que participem da composição de um objeto complexo.

1.e) RETIRAR - Idem, acima.

2) Não é possível ter instâncias dependentes ligadas a instâncias pai que não existam (integridade referencial). Os comandos seguintes a garantem.

2.a) INSERIR PARTES - A condição que seleciona o(s) pai(s) de um registro dependente não pode retornar um conjunto vazio.

2.b) CONECTAR - Idem, acima.

3) Não é permitido formar ciclos nas instâncias de um objeto raiz recursivo. Os comandos abaixo a garantem.

3.a) INSERIR PARTES - Não é possível inserir um dependente para um pai que não exista (restrição 2).

3.b) CONECTAR - O comando rejeita a possível formação de um ciclo.

Muitas outras restrições de integridade poderiam fazer parte do modelo. Como apontado na seção 3.1, um mecanismo generalizado de restrições de integridade pode ser bastante útil. Listamos, abaixo, algumas que poderiam ser opcionalmente incorporadas.

1) Para uma relação dependente de mais de uma relação pai, poderia ser indicada, para cada relação pai em separado, a obrigatoriedade ou não de um registro dependente participar de uma ligação com ele (como "membership class" do modelo CODASYL).

2) Não permitir o compartilhamento de objetos fora do mesmo objeto (isto impediria, de saída, a declaração de dependentes compartilhados diretamente da relação raiz). Por exemplo: se o dado D1 é compartilhado entre os fluxos F1 e F2, então F1 e F2 devem ser dependentes do mesmo objeto, digamos - o processo P1.

3) Não permitir que o relacionamento de dependentes com pais de entidades diferentes ocorra fora do mesmo objeto. Por exemplo: se o dado D1 for componente do fluxo F1 e do depósito de dados DD1, então F1 e DD1 devem pertencer ao mesmo processo, digamos P1.

As restrições 2 e 3 poderiam ser ainda modificadas, por exemplo, levando-se em conta o fato da composição recursiva da relação raiz. Os dependentes diretos de um sub-objeto recursivo podem ser considerados, ou não, como sub-objetos pertencentes ao objeto raiz. Por exemplo: na restrição 3 acima, pode ser permitido compartilhar o dado D1, se F1 for componente direto de P1, e DD1 componente direto do processo P1.1, sendo P1.1 um componente recursivo de P1.

Com um mecanismo generalizado de restricões de integridade, é possível implementar semânticas diferentes no relacionamento de composição de objetos complexos (vide 3.1).

VI.5) Indicações para Implementação.

As relações que compõem um objeto complexo no COPPEREL (dependentes ou raiz) são armazenadas como relações normais da base de dados. Não ocorre grupamento de registros ou qualquer outro mecanismo que as diferencie. A razão disto é

que os comandos LOPEREL anteriormente existentes (com exceção de INSERIR e RETIRAR) podem ser aplicados às relações do objeto complexo, diminuindo o esforço de implementação.

Para materializar as ligações entre registros pais e seus dependentes, a idéia é utilizarmos índices de junção (VALDURIEZ 1987). Para funcionamento eficiente destes, cada relação do objeto complexo tem um atributo adicional, totalmente invisível para o usuário, que funciona como um "surrogate", isto é, um identificador único, atribuído pelo sistema, e não reutilizável. O usuário deve continuar a utilizar o conceito de chave primária para acessar, univocamente, um registro de uma relação qualquer de um objeto complexo. O usuário deve evitar o uso de chaves concatenadas, pois o sistema não tem como garantir a integridade referencial.

Índices de Junção são usados para materializar relacionamentos de composição normais (1:N) ou compartilhados (N:M). Normalmente um relacionamento compartilhado seria representado por uma relação adicional, que, como tal, poderia ter atributos adicionais. Isto não é possível na proposta de objetos complexos no COPPEREL, isto é, o usuário não pode criar atributos para relacionamentos do tipo compartilhado. A razão disto é obter maior eficiência, pois um índice de junção pode ser usado diretamente, sem o "overhead" do uso de uma relação normal.

Índices de Junção, que materializam junção por chave estrangeira (o operador do predicado é igualdade), apresentam boa seletividade e, segundo (VALDURIEZ 1987), apresentam bons resultados de desempenho em relação a algoritmos usuais de junção.

Uma nova relação do esquema conceitual, de nome TAB_COMPOSIÇÃO, deve ser criada para representar a composição do objeto complexo. Cada registro, nesta relação, representa um relacionamento de composição entre uma relação dependente e seu pai. Os atributos desta relação são:

- Nome do arquivo pai;
- Nome do arquivo dependente;
- Indicação de Compartilhamento (Compartilhado ou Não).

A figura 6.9 representa as instâncias desta tabela para o objeto DFD.

TAB_COMPOSICAO

pai	dependente	comp.
PROCESSOS	PROCESSOS	NC
PROCESSOS	FLUXOS	NC
PROCESSOS	DEP_DADOS	C
PROCESSOS	ENT_EXT	NC
FLUXOS	DADOS	C
DEP_DADOS	DADOS	NC

Figura 6.9 - Relação TAB_COMPOSIÇÃO.

CAPÍTULO VII

CONCLUSÃO.

A pesquisa de modelos de dados (2.2) e controles operacionais (2.3), necessários para definir e construir Sistemas de Gerência de Banco de Dados que suportem satisfatoriamente as novas classes de aplicação, está ainda em suas fases iniciais. Provavelmente, um consenso só começará a ser esboçado à medida em que as propostas sejam transformadas em implementações reais testadas no uso prático. No entanto, algumas das funcionalidades desejáveis já começam a serem delineadas em sistemas experimentais (capítulo II) e outras foram por nós apontadas no capítulo III.

Três extensões foram propostas como o trabalho desta tese: i) interface para linguagem complexas; ii) campos longos e iii) objetos complexos.

A interface para linguagens hospedeiras, por nós implementada (capítulo IV), está permitindo, atualmente, o uso do COPPEREL em diversos trabalhos: i) uma ferramenta gráfica para projeto de sistemas está em fase final de acabamento (TRAVASSOS 1988); ii) uma interface para o usuário final, baseada em janelas, está sendo construída e iii) uma proposta para a simulação de um SGBD Orientado a Objetos em cima do COPPEREL está sendo estudada.

Uma das controvérsias mais acirradas é quanto adaptação do Modelo Relacional - enfoque seguido neste trabalho - que é um modelo simples, embasado teoricamente, e conta com alguns anos de utilização, versus a criação de SGDBs inteiramente novos, estes últimos inspirados nas propostas que tiveram divulgação mais recentemente, mais notadamente, aquelas que aplicam o paradigma de Orientação a Objetos. Porém, muitas das lições aprendidas em uma área podem ser aplicadas a outra. Desta forma, o desenvolvimento

e implementação (5.3) dos algoritmos de campos longos (capítulo V) visou a aplicação na criação de um novo tipo de dado que pode ser propriamente inserido no contexto do Modelo Relacional, como exemplificado nas extensões da LOPEREL (5.2); ou ainda, servir como o método de acesso para um Gerente Geral de Armazenamento para um SGBD Orientado a Objetos ou Extensível. O uso de rótulos (5.4), se eficientemente implementado, permitrá maior flexibilidade graças ao aumento do nível de abstração na manipulação da estrutura de dado.

A capacidade de lidar com Campos Longos, aliada a mecanismos que permitam a implementação de controles operacionais diretamente neste e ,também, a mecanismos que suportem Tipo Abstratos de Dados, permite obter, talvez, o que sejam as primitivas necessárias para a construção da nova geração de SGBDs, ou pelo menos, do núcleo de Sistemas de Gerência da Banco de Dados Extensíveis (2.1.4)

A proposta de Objetos Complexos (capítulo VI) no COPPEREL visou o estudo do enfoque estrutural, permitindo algum grau de flexibilidade e operações de alto nível (6.3), de acordo com os requisitos apontados em 3.1 e, também, com as limitações do sistema. Novamente, mais pesquisa é necessária no sentido de incorporar os enfoques operacional e comportamental, através de uma facilidade de Tipos Abstratos de Dados que permita reconhecer os Objetos Complexos como entidades reais do SGBD. Um ponto importante consiste em trabalhar no desenvolvimento de um Modelo Conceitual - adequado ao tratamento do problema de Objetos Complexos - que seja independente de novas propostas de implementação ou adaptação de sistemas já existentes.

Referências:

ABITEBOUL 1987 - Abiteboul S., Hull R. "IFO: A Formal Semantic Data Model" pp 525-565 ACM Trans. Database Syst. vol 12, no 4, december 1987.

ABRIAL 1974 - Abrial, J. R. "Data Semantics" Data Base Management, J. W. Klimbie and K. L. Koffemen, Eds North-Holland, Amsterdam, pp. 1-59, 1974.

AHLSEN 1985 - Ahlsen M., Bjornerstedt A., Hulten C. "OPAL: An Object-Based System for Application Development" pp 31-40 IEEE Database Engineering Vol 8 no 4 december 1985.

ANDREWS 1987 - Andrews T., Harris C., "Combining Language and Database Advances in an Object-Oriented Development Environment" pp 430- 440 Proc. of ACM OOPSLA october 1987.

BANCILHON 1987 - F. Bancilhon, T. Briggs, S. Kheshafian and P.Valduriez "FAD, A Powerful and Simple Database Language" Proceeding of the 1987 VLDB Conference.

BANERJEE 1987a - Banerjee J., Chou, H., Garza, J. F., Kim, W., Woelk, D., Balou, N. , Kim H. "Data Model Issues for Object-Oriented Applications" ACM Trans. Office Inf. Syst. 5,1 pp 3-26 january 1987.

BANERJEE 1987b - Banerjee J., Chou, H., Garza, J. F., Woelk, D. "Composite Object Support in an Object-Oriented Database System" pp 118- 125 Proc. of ACM OOPSLA october 1987.

BANERJEE 1987c - Banerjee J., Kim, W., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases" Proc. ACM SIGMOD Annual Conference, december 1987.

BERNSTEIN 1987 - Bernstein P. A. "Database System Support for Software Engineering - An Extended Abstract -" pp 166-178 9th Inter. Conf. on Software Engineering, Monterey, april 1987.

BEVER 1988 - M. Bever, D. Ruland "Aggregation and Generalization Hierarchis in Office Automation" ACM Transaction Office Automation 1988.

BLOOM 1987 - Bloom T., Zdonick S.B. "Issues in the design of Object-Oriented Database Programming Languages" pp 441- 451 Proc. of ACM OOPSLA october 1987.

BRODIE 1982 - Brodie, M. L., Silva E. "Active and Passive Component Modelling: ACM/PCM" pp 41-91 Information

Systems Design Methodologies: A Comparative Review may 1982.

BRODIE 1984 - Brodie, M. L. "On the Development of Data Models" On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages, M. L. Brodie, J. Mylopoulos, and J. W, Schmidt, Eds. Springer-Verlag, New York, pp 19-48, 1984.

CAREY 1986a - Carey, M.J., De Witt, D.J., Richardson, J.E., and E. Shekita, "Object and File Management in EXODUS Extensible Database System" pp 91-100 Proceeding of the 1986 VLDB Conference, Kyoto, Japan.

CAREY 1986b - Carey, M.J., De Witt, D.J., Richardson, J.E. and E. Shekita, Frank .D, Graefe .G,Muralikrisha, "The Architecture of the EXODUS Extensible DBMS" Proceeding of the 1986 International Workshop on Object-Oriented Database System.

CHEN 1976 - Chen, P. "The Entity-Relationship Model: Toward a Unified View of Data" ACM Trans. Database Systems, pp 9-36, march 1976.

CODD 1979 - Codd, E. F. "Extending the database Relational Model to Capture More Meaning" ACM Trans. Database Systems pp 397-434, December 1979.

COX 1986 - Cox, B. J. "Object-Oriented programming" Addison-Wesley, 1986.

DATE 1986 - Date, C. J. "An Introduction to Database Systems" (4th edition- Addison-Wesley, 1986.

DITTRICH 1986a - Dittrich, K., Lorie, R.A. "Object-Oriented Database Concepts for Engineering Applications" Research Report, IBM Research Laboratory, San Jose, 1985.

DITTRICH 1986b - Dittrich, K. "Object-Oriented database systems - a workshop report" Proc. 5th ER Conference, Dijon North Holland Publ.Group, 1986.

FISHMAN 1987 - Fishman, D. H., Beech D., Cate H. P., Chow E. C., Connors T., Davis J. W., Derrett N., Hoch C. G., Kent W., Lyngbaek P., Mahbod B., Neimat M. A., Ryan T. A., Shan C. "Iris " An Object-Oriented Database Management System" ACM Trans. Office Inf. Syst. 5,1 pp 48-69 january 1987.

FOISSEAU 1982 - Foisseau J., Valette F.R. " A Computer Aided Design Data Model: FLOREAL" pp 316-334 File Structures and Data Bases for CAD, North-Holland Pub. Comp.,1982.

GALLO 1986 - Gallo F., Minot R., Thomas I. "The Object Management System of PCTE as a Software Engineering

Database Management System" SIGPLAN Notices vol 22 no 1, pp 12-15 january 1986.

GANE 1979 - Gane, C. e Sarson, T.,"Structured Systems Analysis", Prentice-Hall, 1979.

GOLDBERG 1983 - Goldberg A., Robson D. , "Smalltalk-80: The Language and its Implementation" Addison-Wesley, 1983.

GOMAA 1984 - Gomaa H. " A Software Design Method for Real Time Systems" pp 938-949 Communnications of ACM vol 27 no 9, september 1984.

GRAY 1978 - Gray J. "Notes on Data Base Operating Systems" IBM Research Report: RJ188, IBM Research, San Jose, Calif, 1978.

GRAY 1981 - Gray J. "The Transaction Concept: Virtues and Limitations" Proceedings of VLDB, 1981.

GUSTAFSSON 1982 - Gustafsson,M.R., Karlsson, T., Bubenko J.A. "A Declarative Approach to Conceptual Information Modeling" pp 93-142 Information Systems Design Methodologies: A Comparative Review, may 1982.

HAMMER 1981 - Hammer, M., Mc Leod, D. "Database Description with SDM:A semantic data model" ACM Trans. Database Syst. pp 351-386, september 1981.

HARDER 1987 - T. Harder, K. Meyer-Wegner, B. Mitschang and A. Sikeler "PRIMA - a DBMS Prototype Supporting Engineering" pp 433-442 Proceeding of the 1987 VLDB Conference.

HASKIN 1981 - Haskin R. L. , K., Lorie, R.A. "On extending the functions of a relational database system" Research Report, IBM Research Laboratory, San Jose, 1981.

HELD 1975 - Helg, G., Stonebreaker, M. R., Wong, E. "INGRES - A Relational Data Base System" pp 409-416 Proc. AFIPS NCC, 1975.

HITCHCOCK 1985 - Hitchcock P., Brown A. W., Weedon R., Earl A. N., Whittington R. P., Robinson D.S. "The Use of Database for Software Engineering"

HORNICK 1987 - Hornick F. M., Zdonik, S.B. " A Shared, Segmented Memory System for an Object-Oriented Database" ACM Trans. Office Inf. Syst. 5,1 pp 70-95 january 1987.

HULL 1987 - Hull, R., King, R. "Semantic Database Modeling: Survey, applications and research issues" ACM Comput. Surv. pp 201-260 , september 1987.

KASPER 1988 - Kasper O. "Abstract Data Types with Shared Operations" Sigplan Notices, vol 23 no. 6, 1988.

KATZ 1987 - Katz H. R., Chang E. "Managing Change in a Computer-Aided Design Database" pp 455-462 Proc of VLDB, England, september 1987.

KEMPER 1987 - Kemper, A., Wallrath, M. "An analysis of Geometric Modeling in Database Systems" ACM Comp. Surveys 19 pp 47-91, march 1987.

KERSCHBERG 1976 - Kerschberg, L., Pacheco, J.E.S. "A Functional Data Base Model" Tech. Rep. Pontifícia Univ. Católica.

KING 1984 - King, R., Mc Leod, D. "A unified model and methodology for conceptual database design" On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages, M. L. Brodie, J. Mylopoulos, and J. W, Schmidt, Eds. Springer-Verlag, New York, pp 19-48, 1984.

KING 1986 - King R. "A Database Management System Based on an Object-Oriented Model" pp 443-468 Expert Database Systems, Benjamin/Cummings Publishing Company, 1986.

KLAHOLD 1986 - Klahold P., Schlageter G., Wilkes W. "A General Model for Version Management in Databases" pp 319-327 Proceeding of the 1986 VLDB Conference, Kyoto, Japan.

LORIE 1982 - Lorie R. "Issues in Databases for Design Applications" pp 215-229 File Structures and Data Bases for CAD, North-Holland Pub. Comp.,1982.

LORIE 1984 - Lorie R., Kim W., McNabb D.,Plouffe W., Meier A. "User Interface and Access Techniques for Engineering Databases" Research Report, IBM Research Laboratory, San Jose, 1984.

MANOLA 1986 - F.Manola, U. Dayal, "PDM: An Object-Oriented Data Model" Proceeding of the 1986 International Workshop on Object-Oriented Database System.

MARYANSKI 1986 - F. Maryanski, J.Bedell, S. Hoelscher, S. Hong, L. McDonald, J. Peckhan, D. Stock, "The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems" Proceeding of the 1986 International Workshop on Object-Oriented Database System.

MATTOSO 1985 - Mattoso, M.L.Q., Zakimi, B.M., Prazeres, M.J., Souza, J.M., "LOPEREL: Uma Linguagem de quarta Geração e sua utilização em Ambiente de SGBD", Anais de CIL85, Barcelona, Espanha, maio 1985.

MATTOSO 1987 - Mattoso, M.L.Q., "A Incorporação de Ferramentas de Apoio aos Usuários do SGBD COPPEREL", dissertação de Mestrado, Progr. de Sistemas - COPPE/UFRJ, maio 1987.

MELO 1988a - Melo R. N. "Bancos de Dados Não Convencionais: A Tecnologia do BD e Suas Novas Áreas de Aplicação" VI Escola de Computação, Campinas, 1988.

MELO 1988b - Melo R. N., Lanzelotte R.S.G., Rezende M.A.M., Gualandi P. "OMS - EITIS: Um Sistema de Gerenciamento de Objetos para o projeto EITIS" Projeto ESTRA, IV Reunião de Trabalho, volume 1 abril-outubro, 1988.

MOSS 1987 - Moss J.E.B. "Log-Based Recovery for Nested Transactions" pp 427-432 Proc of VLDB, England, september 1987.

MYLOPOULOS 1980 - Mylopoulos J., Berstein P. A., Wong H. "A Language Facility for Designing Database-Intensive Applications" pp 185-207 ACM Trans. on Database Systems vol 5 no 2, june 1980.

NAVATHE 1986 - Navathe, S., Elmasri, R., Larson J. "Integrating User Views in Database Design" pp 50-62 Computer 19, january 1986.

NIERSTRASZ 1985 - Nierstrasz O.M., Tsichritzis D.C. "An Object-Oriented Environment for OIS Applications" pp 335-345 Proc. of VLDB 1985, Stockholm

OZSOYOGLU 1987 - Ozsoyoglu G., Ozsoyoglu Z. M., Matos V. "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions" ACM Trans. Database. Syst. 12,4 pp 566-592 december 1987.

PAUL 1987 - Paul H.B., Scheck H.J., Scholl M.H.,Weikum G., Deppisch U. "Architecture and Implementation of the Darmstadt Database Kernel System" Proc. ACM SIGMOD Annual Conference, december 1987.

PECKHAM 1988 - Peckham, J , Maryanski, F. "Semantic Data Models" ACM Computing Surveys, Vol 20, No 3, September 1988.

PENEDO 1986 - Penedo H.M. "Prototyping a Project Master Data Base for Software Engineering Environmennts" pp 1-11 SIGPLAN Notices, Vol 22 no 1, january 1986.

RICHTER 1982 - Richter,G., Durchholz R. "IML-Inscribed High-Level Petri Nets" pp 335-368 Information Systems Design Methodologies: A Comparative Review may 1982.

ROCHA 1988 - Rocha,A.R.C. e Souza, J.M, "TABA: Uma Estação de Trabalho para o Engenheiro de Software", Relatório Técnico COPPE/UFRJ ES-145/88.

SCHOLL 1987 - Scholl M., Paul H.B., Scheck H.J. "Supporting Flat Relations by a Nested Relational Kernel" pp 137-146 Proc of VLDB, England, september 1987.

SCHWARZ 1986 - P. Schwarz, W. Chang, J.C. Frugtag, G. Lohman, J. McPherson, C. Mohan, H. Pirahesh

"Extensibility in the Starburst Database System". Proceeding of the 1986 International Workshop on Object-Oriented Database System.

SHIPMAN 1981 - Shipman, D. W. "The Functional data model and the data language DAPLEX". ACM Trans. Database Syst. pp 140-173, march 1981.

SMITH 1977 - Smith, J. M., Smith, D.C.P. "Database Abstractions: Aggregation and Generalization" ACM Trans. Database Syst. pp 105-133 march, 1977.

SMITH 1987 - Smith K., Zdonik S.B. "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems" pp 452-465 Proc. of ACM OOPSLA october 1987.

SOUZA 1988a - Souza, J.M., Rocha, A.R.C., Aguiar, T.C. "TABA: Uma Estção de Trabalho Configurável para Desenvolvimento de Software" pp 12-28 Ambientes de Desenvolvimento de Software e o Projeto TABA, Seminário do Prog. Eng. Sist. e Comp. COPPE/UFRJ, eds Rocha, A.R.C., Souza, J.M., dezembro de 1988.

SOUZA 1988b - Souza, J.M. e Mattoso, M.L.Q., "COPPEREL-PC: A Versão do SGBD COPPEREL para Microcomputadores Tipo PC", Anais do 3o. Simpósio Brasileiro de Banco de Dados, março 1988.

STEFICK 1986 - Stefick, M., Bobrow, D. G. "Object-Oriented Programming: Themes and variations" AI Magazine pp 40-62, january 1986.

STONEBREAKER 1983 - Stonebreaker, M., H. Stettner, N.Lynn, J.Kalash and A.Guttman, "Document Processing in a Relational Database System", ACM Transactions on Office Information Systems. 1,2, April 83.

STONEBREAKER 1986 - Stonebreaker M. "Object Management in POSTRGRES Using Procedures" Proceeding of the 1986 International Workshop on Object-Oriented Database System.

STONEBREAKER 1987 - Stonebreaker M., Rowe L. "The POSTRGRES Data Model" pp 83-96 Proc of VLDB, England, september 1987.

SU 1983 - Su, S.Y.W. "SAM*: A semantic association model for corporate and scientific-statistical databases" Inf. Sci. 29, pp 151-199, 1983.

SU 1986 - Su, S.Y.W. "Modeling integrated manufacturing data with SAM*" Computer 19 pp 34-49, january 1986.

TAKAHASHI 1988 - Takahashi T. "Introdução a Programação Orientada a Objetos" III EBAI, Curitiba, janeiro 1988.

TEOREY 1986 - Teorey, T.J., Yang, D., Fry, J. P. "A logical design methodology for relational databases using the

extended entity relationship model" ACM Comput. Surv. 18 pp 197-222, june 1986.

TRAVASSOS 1988 - Travassos, G. H., "FEGRES: Uma Ferramenta Gráfica para Especificação de Sistemas", Relatório técnico Programa de Engenharia de Sistemas - COPPE/UFRJ, 1988.

TSICHRITZIS 1987 - Tsichritzis D., Fiume E., Gibbs S., Nierstrasz O. "KNOs: KNowledge Acquisition, Dissemination, and Manipulation Objects" ACM Trans. Office Inf. Syst. 5,1 pp 96-112 january 1987.

ULLMAN 1982 - Ullman J. "Principles of Database Systems" Computer Science Press 1982.

VALDURIEZ 1986 - Valduriez P., Khoshafian S., Copeland G. "Implementation Techniques of Complex Objects" pp 101-110 Proceeding of the 1986 VLDB Conference, Kyoto, Japan.

VALDURIEZ 1987 - Valduriez P., "Join Indices" pp 218-246 ACM Trans. Database Systems vol 12 no 2, june 1987

VERHEIJEN 1982 - Verheijen, G.M.A., Van Bekkum J. "NIAM: An Information Analysis Method" pp 537-589 Information Systems Design Methodologies: A Comparative Review may 1982.

ZANIOLO 1986 - Zaniolo C., Ait-Kaci H., Beech D., Cammarata S., Keshberg L., Maier D. "Object Oriented Database Systems and Knowledge Systems" pp 49-66 Expert Database Systems, Benjamin/Cummings Publishing Company, 1986.

ZAKIMI 1982a - Zakimi, B.M. et all., "COPPEREL: Sistema de Gerência de Base de Dados da COPPE Baseado em Algebra Relacional", Anais do IX SEMISH, 1982.

ZAKIMI 1982b - Zakimi, B.M. et all., "COPPEREL: Sistema de Gerência de Base de Dados da COPPE", Anais da IX Conferência Latino-Americana de Informática, Lima, 1982.

ZDONICK 1985 - Zdonick S. "Object Management Systems for Design Environments" pp 23-30 IEEE Database Engineering Vol 8 no 4 december 1985.

APÊNDICE A

SINTAXE COMPLETA DA LOPEREL COM EXTENSÕES

Palavras reservadas e abreviaturas:

ABANDONAR	ABAND
ABOLIR	ABOL
ABRIR	
ABSOLUTO	ABS
ADICIONANDO	ADIC
ALFANUMERICO	A
ARQUIVO	ARQ
ASSERCAO	ASS
ATRIBUTOS	ATRS
ATUALIZAR	AT
AUTORIZACAO	
AUTORIZAR	
BASE	
CALCULAR	CALC
CATALOGAR	CAT
CONECTAR	CONECT
CARD	
CASO	
CHAVE	CH
COM	
COMO	
COMPONENTE	COMP
COMPARTILHADO	COMPAR
COMPLETO	COMPL
CONCATENAR	CONC
CONC_CADEIA	CONC_C
CONTENHA	CONT
CONV_DATA	
COPIAR	COP
COSSENO	COS
CRIAR	
CRUZAR	CRUZ
DADOS	
DATA	
DE	
DEFINIR	DEF
DESALOCAR	DES
DESCONECTAR	DESC
DESVIAR	DESV
DESVIO_PADRAO	D_P
E	
EM	
ENCERRAR	ENC
ERRO	
ESCREVER	ESC
ESTREITAR	ESTR
EXECUTAR	EXEC

EXPONENCIAL	EXP
FALSO	F
FAZER	
FECHAR	
FIM	
FORA	
GRAVADOS	GRAVS
GRAVAR	GR
GRUPAR	GRUP
HORA	
IMPRIMIR	IMP
INDICE	IND
INTEGRIDADE	INTEG
INTEIRO	I
INT_REAL	I_R
INSERIR	INS
JUNTAR	JUNT
LER	
LER_ESCREVER	L_E
LOGARITMO	LOG
LONGO	
MAIOR	
MEDIA	MED
MENOR	
MODIFICAR	MOD
MODIFICAR-CAMPO	MOD-C
MOSTRAR	MOST
MOSTRAR-CAMPO	MOST-C
OBJETO	OBJ
ORDEM	
ORDENAR	ORD
OU	
PARA	
PARTES	PART
PERCORRENDO	PERC
POR	
PRIM_CADEIA	PRIM_C
PROCEDIMENTO	PROC
QUE	
RAIZ	
RAIZ_QUADRADA	R_Q
REAL	R
REAL_INT	R_I
RECONSTRUIR	REC
REGISTROS	REGS
REMOVER	REM
REM_CADEIA	REM_C
RENOMEAR	REN
RETIRAR	RET
REVOGAR	REV
SE	
SEGUINTES	SEGS
SELECIONAR	SEL
SEM	
SENO	SEN
SESSAO	
SOBRE	
SUBS_CADEIA	SUBS_C
SUBTRAIR	SUB

```
TAL
TIPO
TOTAL                          TOT
ULT_CADEIA                     ULT_C
UNIR
USUARIO                        US
VARIAVEL                       VAR
VERDADEIRO                     V
VERIFICAR                      VER
VISTA
```

Sintaxe completa da linguagem:

```
programa ::=

    [sessao;] ... ENCERRAR

sessao ::=

    abrir-sessao ; [transacao;] ... fechar-sessao

abrir-sessao ::=

    ABRIR SESSAO PARA USUARIO nome-usuario/senha-usuario

fechar-sessao ::=

    FECHAR SESSAO

transacao ::=

    {abrir-base-de-dados} ; [gerencia;] ...
      fechar-base-de-dados

    {criar-base-de-dados}   [comando; ]

criar-base-de-dados ::=

    CRIAR BASE DE DADOS  nome-da-base-de-dados EM
      dispositivo1 E disposotivo2
      PARA inteiro REGISTROS POR ARQUIVO

abrir-base-de-dados ::=

    ABRIR BASE DE DADOS nome-da-base-de-dados

fechar-base-de-dados ::=

    FECHAR BASE DE DADOS [SEM ATUALIZAR]
```

```
gerencia ::=

     {definir-vista              }
     {abolir-vista               }
     {definir-assercao           }
     {abolir-assercao            }
     {catalogar-procedimento     }
     {abolir procedimento        }
     {remover-arquivo            }
     {remover-variavel           }
     {renomear-arquivo           }
     {renomear-atributos         }
     {autorizar-usuario          }
     {revogar-autorizacao        }
     {reconstruir-base-de-dados}
     {gerencia-condicional       }
     {remover-indice             }


definir-vista ::=

     DEFINIR VISTA nome-da-vista cabecalho COMO :
           [instrucao;] ... FIM


abolir-vista :: =

     ABOLIR VISTA nome-da-vista


definir-assercao ::=

     DEFINIR ASSERCAO nome-da-assercao cabecalho
      COMO : [instrucao;] ... FIM


abolir-assercao ::=

     ABOLIR ASSERCAO nome-da-assercao


catalogar-procedimento ::=

     CATALOGAR PROCEDIMENTO nome-do-procedimento
       [ (nome-do-parametro ,,,) ] COMO :
                            ---

        corpo-do-procedimento FIM


corpo-do-procedimento ::=

     [ qualquer terminal da linguagem ] ... ##


abolir-procedimento ::=

     ABOLIR PROCEDIMENTO nome-do-procedimento
```

```
remover-arquivo ::=

    REMOVER ARQUIVO nome-do-arquivo


remover-variavel ::=

    REMOVER VARIAVEL nome-da-variavel


renomear-arquivo ::=

    RENOMEAR ARQUIVO nome-antigo PARA nome-novo


renomear-atributos ::=

    RENOMEAR ATRIBUTOS DE nome-do-arquivo :

    { atributo-velho PARA atributo-novo } ,,,
                                          ---


autorizar-usuario ::=

    AUTORIZAR USUARIO nome-do-usuario / senha-do-usuario

              {REMOVER      }
    PARA      {ESCREVER     } nome-do-arquivo
              {LER_ESCREVER}
              {LER          }


revogar-autorizacao ::=

    REVOGAR AUTORIZACAO DE
          nome-do-usuario / senha-do-usuario

    PARA nome-do-arquivo


reconstruir-base-de-dados ::

    RECONSTRUIR BASE DE DADOS nome-da-BD


gerencia-condicional ::=

    SE expressao-logica FAZER :  [gerencia;]

                                 [comando; ] ... FIM


remover-indice ::=

    REMOVER INDICE DE nome-do-arquivo
          SOBRE nome-do-atributo ,,,
                                 ---
```

```
instrucao ::=

                          {comando                }
    [ (rotulo) : ]        {desvio                 }
                          {instrucao-condicional  }

desvio ::=

    DESVIAR PARA rotulo

instrucao-condicional ::=

    SE expressao-logica FAZER : [instrucao;] ... FIM
```

```
comando ::=
     {criar-arquivo           }
     {criar-variavel          }
     {desalocar-arquivo       }
     {desalocar-vista         }
     {verificar-integridade   }
     {em-caso-de-erro         }
     {ordenar-registros       }
     {abandonar-ordem         }
     {copiar                  }
     {uniao                   }
     {intersecao              }
     {diferenca               }
     {concatenacao-cartesiana}
     {projecao                }
     {divisao                 }
     {juncao                  }
     {selecao                 }
     {modificar-registros     }
     {remover-registros       }
     {inserir-registros       }
     {inserir-em-variavel     }
     {recuperacao             }
     {calculo-expressao       }
     {procedimento            }
     {bloco                   }
     {criar-indice            }
     {conexao                 }
     {desconexao              }
     {mostrar-campo           }
     {modificar-campo         }
     {criar-rotulo            }
     {eliminar-rotulo         }

criar-arquivo ::=

     CRIAR ARQUIVO [$]nome-do-arquivo cabecalho


criar-variavel ::=

     CRIAR VARIAVEL [$]nome-da-variavel tipificacao


desalocar-arquivo ::=

     DESALOCAR ARQUIVO nome-do-arquivo


desalocar-vista ::=

     DESALOCAR VISTA nome-da-vista


verificar-integridade ::=

     VERIFICAR INTEGRIDADE COM nome-da-assercao
```

```
em-caso-de-erro ::=

    EM CASO DE ERRO EXECUTAR nome-do-procedimento


ordenar-registros ::=

    ORDENAR [REGISTROS DE] nome-do-arquivo
          POR nome-do-atributo ,,,
                               ---

abandonar-ordem ::=

    ABANDONAR ORDEM DE nome-do-arquivo


copiar ::=

    COPIAR
    [ : {nome-do-atributo1 PARA nome-do-atributo2} ,,, DE ]
                                                   ---
         nome-do-arquivo1 EM [$]nome-do-arquivo2


uniao ::=

    UNIR nome-do-arquivo1
     COM nome-do-arquivo2 EM [$]nome-do-arquivo3


intersecao ::=

    CRUZAR nome-do-arquivo1
      COM nome-do-arquivo2 EM [$]nome-do-arquivo3


diferenca ::=

    SUBTRAIR nome-do-arquivo1 DE nome-do-arquivo2
          EM [$]nome-do-arquivo3

concatenacao-cartesiana ::=

    CONCATENAR nome-do-arquivo1 COM nome-do-arquivo2
         EM [$]nome-do-arquivo3
```

```
projecao ::=

    ESTREITAR nome-do-arquivo1 PARA {nome-do-atributo} ,,,
                                                       ---
        EM [$]nome-do-arquivo2

divisao ::=

    GRUPAR nome-do-arquivo-dividendo

    POR {nome-do-atributo-dividendo} ,,,
                                     ---
    TAL QUE {nome-do-atributo-dividendo} ,,,
                                         ---
    CONTENHA {nome-do-atributo-divisor} ,,,
                                        ---
    DE nome-do-arquivo-divisor

    EM [$]nome-do-arquivo-quociente

juntar ::=

    JUNTAR nome-do-arquivo1 COM nome-do-arquivo2

    TAL QUE {nome-do-atributo1 op-rel nome-do-atributo2},,,
                                                        ---
    EM [$]nome-do-arquivo3

op-rel ::=

    { =  }
    { <> }
    { <  }
    { <= }
    { >  }
    { >= }
    { ]  }
    { [  }

selecao ::=

    {selecao-simples}
    {selecao-complexa}

selecao-simples ::=

    SELECIONAR [REGISTROS DE] nome-do-arquivo1
      TAL QUE condicao EM [$]nome-do-arquivo2

selecao-complexa    ::=

    SELECIONAR OBJETO [*] nome-arq1
      TAL QUE condicao EM [$]prefixo
      [{COMPLETO                                }]
      [{ [ADICIONANDO] PARTES {nome-arq2},,,  }]
```

```
modificar-registros ::=

    MODIFICAR [REGISTROS DE] nome-do-arquivo
     TAL QUE :
     {condicao ( {nome-do-atributo PARA expressao},,, )} ,,,
                                                  ---     ---

remover-registros ::=

    {remover-simples}
    {remover-complexo}

remover-simples   ::=

    RETIRAR [REGISTROS DE] nome-do-arquivo TAL QUE condicao

remover-complexo  ::=

     RETIRAR PARTES nome-arq-comp TAL QUE condicao

inserir-registros ::=

    {inserir-simples}
    {inserir-complexo}

inserir-simples   ::=

                          {DE nome-arq-externo}
    INSERIR [REGISTROS] {SEGUINTES           }
                          {GRAVADOS          }
      EM nome-arq-interno


inserir-complexo   ::=

                    {DE nome-arq-externo}
    INSERIR PARTES  {SEGUINTES           } EM nome-arq-comp
                    {GRAVADOS            }

     [COMO COMPONENTE DE {condicao EM nome-arq-pai},,,]


inserir-em-variavel ::=

    INSERIR EM nome-da-variavel


recuperacao ::=

    {recuperacao-simples}
    {recuperacao-complexa}

recuperacao-simples ::=

    {IMPRIMIR}
    {MOSTRAR }
    {GRAVAR  }
```

```
[nome-do-arquivo                                    ]
[:lista-de-saida2,,,                                ]
                 ---
[nome-do-arquivo : lista-de-saida1,,,               ]
                                  ---
[nome-do-arquivo COM lista-de-saida3,,,             ]
                                    ---
[nome-do-arquivo POR nome-do-atributo,cadeia]


lista-de-saida1 ::=

    [@                                                  ]
    [/                                                  ]
    [[cadeia                                ]           ]
    [[nome-do-atributo                      ] (coluna)  ]
    [[nome-da-variavel                      ]           ]
    [[nome-do-cursor.nome-do-atributo]                  ]


lista-de-saida2 ::=

    [@                                                  ]
    [/                                                  ]
    [[cadeia                                ]           ]
    [[nome-da-variavel                      ] (coluna)  ]
    [[nome-do-cursor.nome-do-atributo]                  ]


lista-de-saida3 ::=

    [@                                  ]
    [/                                  ]
    [cadeia                             ]
    [nome-do-atributo                   ]
    [nome-da-variavel                   ]
    [nome-do-cursor.nome-do-atributo]


recuperacao-complexa ::=

    MOSTRAR OBJETO nome-arq


calculo-expressao ::=


    CALCULAR expressao EM [nome-da-variavel                 ]
                          [nome-do-cursor.nome-do-atributo]


procedimento ::=

    nome-do-procedimento[([parametro],,,)]
                                    ---

parametro ::=

    [qualquer terminal da linguagem] ... #
```

```
bloco ::=

     [PARA nome-do-cursor PERCORRENDO nome-do-arquivo]
       FAZER : [instrucao;] ... FIM


criar-indice ::=

     CRIAR INDICE EM nome-do-arquivo
           SOBRE {nome-do-atributo} ,,,
                                    ---


expressao-logica ::=

     fator-logico [OU fator-logico] ...


fator-logico ::=

     {FALSO                        }
     {VERDADEIRO                   }
     {expressao op-rel expressao}

             [    {FALSO                        }]
             [ E  {VERDADEIRO                   }]
             [    {expressao op-rel expresao}] ...


op-rel ::=

     [ =   ]
     [ <>  ]
     [ <   ]
     [ <=  ]
     [ >   ]
     [ >=  ]
     [ [   ]
     [ ]   ]

conexao   ::=

     CONECTAR [REGISTROS DE] nome_arq_componente
      TAL QUE condicao1
      [COMO]  COMPONENTE  [DE]
      {condicao2 EM  nome_arq_pai},,,


desconexao ::=

      DESCONECTAR [REGISTROS DE] nome_arq_componente
       TAL QUE condicao1
       FORA [DE] {[condicao2] [EM] nome_arq_pai},,,


mostrar-campo ::=

      MOSTRAR-CAMPO nome-arq, nome-atr (inicio:fim)
```

```
modificar-campo ::=

      MODIFICAR-CAMPO   nome-arq,   nome-atr   (inicio:fim)
        PARA  expressao-alfanumerica


criar-rotulo ::=

      CRIAR ROTULO nome-rotulo EM nome-arq,
        nome-atr (posicao)
        [TAL QUE condicao]


eliminar-rotulo ::=

       ELIMINAR ROTULO nome-rotulo EM nome-arq, nome-atr
         [TAL QUE condicao]


expressao ::=

     {expressao-aritmetica  }
     {expressao-alfanumerica}


expressao-aritmetica ::=

     [-]termo[{+}     ]
             [{-}termo]...


termo ::=

     primario[{*}        ]
             [{/}primario]...
```

```
primario ::=

     {inteiro                                                          }
     {real                                                             }
     {nome-de-variavel                                                 }
     {nome-de-cursor.nome-de-atributo                                  }
     {{ABSOLUTO      }                                                 }
     {{EXPONENCIAL   }                                                 }
     {{LOGARITMO     }                                                 }
     {{RAIZ_QUADRADA} (expressao-aritmetica)                           }
     {{SENO          }                                                 }
     {{COSSENO       }                                                 }
     {{INT_REAL      }                                                 }
     {{REAL_INT      }                                                 }
     {CONV_DATA (expressao-alfanumerica)                               }
     {{MAIOR             }                                             }
     {{MENOR             }                                             }
     {{TOTAL             } (nome-do-arquivo,nome-do-atributo)          }
     {{DESVIO_PADRAO     }                                             }
     {{MEDIA             }                                             }
     {CARD (nome-do-arquivo)                                           }


expressao-alfanumerica ::=

     {nome-de-variavel                 }
     {nome-de-cursor.nome-de-atributo  }
     {constante-alfanumerica           }


constante-alfanumerica ::=

     {cadeia                                                  }
     {DATA                                                    }
     {HORA                                                    }
     {{CONC_CADEIA}                                           }
     {{REM_CADEIA } (exp-alfa1, exp-alfa2)                    }
     {SUBS_CADEIA (exp-alfa1, exp-alfa2, exp-alfa3)           }
     {{PRIM_CADEIA}                                           }
     {{ULT_CADEIA } (nome-do-arquivo, nome-do-atributo)       }


cabecalho ::=

     COM inteiro REGISTROS DE inteiro ATRIBUTOS :

     {nome-do-atributo tipificacao} ,,,
                                    ---

     COM CHAVE : {nome-do-atributo-chave} ,,,
                                          ---

     [E COM INDICE SOBRE : {nome-do-atributo-indice} ,,,]
                                                     ---

     [[ RAIZ                                          ]]

     [[ RAIZ RECURSIVA [COMPARTILHADA]                ]]

     [[ {COMPONENTE [COMPARTILHADO]
                    DE {nome-arq-pai},,,},,,]]
```

```
tipificacao ::=

     {DE TIPO} {REAL (inteiro:inteiro) }
     {  :      } {INTEIRO (inteiro)      }
               {ALFANUMERICO (inteiro) }
               {LONGO}

condicao ::=

     fator-condicional [OU fator-condicional] ...


fator-condicional ::=

     {FALSO                                                    }
     {VERDADEIRO                                               }
     {nome-var  op-rel  {cadeia                              }}
     {                  {[-]{inteiro                       }}}
     {                  {[-]{numero                        }}} EEE
     {                  {   {nome-da-variavel              }}} ---
     {                  {   {nome-do-cursor.nome-do-
                                              atributo}}}


componentes-da-linguagem ::=

     {simbolo-basico    }
     {identificador     }
     {numero            }
     {comentario        }
     {palavra-reservada }
     {cadeia            }


simbolo-basico ::=

     {letra      }
     {digito     }
     {delimitador}
     {sublinha   }

letra ::=

     A --> Z

digito ::=

     0 --> 9

sublinha ::=

     _
```

```
delimitador ::=

     {espaco}
     {  ,    }
     {  ;    }
     {  "    }
     {  :    }
     {  (    }
     {  )    }
     {  #    }
     {  ##   }
     {  .    }
     {  $    }
     {  @    }
     {  +    }
     {  -    }
     {  *    }
     {  /    }
     {  =    }
     {  <>   }
     {  <    }
     {  <=   }
     {  >    }
     {  >=   }
     {  [    }
     {  ]    }


espaco ::=

     b/ ...


identificador ::=

           {letra   }
     letra {digito  } ...
           {sublinha}


numero ::=

     {inteiro}
     {real}


inteiro ::=

     digito ...


real ::=

     inteiro . [digito] ...
```

```
comentario ::=

    (* [ qualquer caracter EBCDIC ] ... *)

cadeia ::=

    " { qualquer caracter EBCDIC } "
```

APÊNDICE B

Este apêndice descreve, brevemente, os módulos do COPPEREL que foram alterados para suportar a interface CPREL (vide capítulo III).

Módulo CARACT

Este módulo lê os caracteres de entrada, de acordo com um pedido do léxico, devolvendo uma classificação. Uma nova classe (de valor 27) foi incluída para reconhecer o fim do buffer de entrada. A rotina lê os caracteres do buffer enviado pela interface CPREL.

Módulo CARATT

Lê os caracteres pedidos pelo módulo LEXCOT. Sua alteração foi similar a do módulo CARACT.

Módulo CRRENT

Este módulo é responsável por criar novas entidades. Foi alterado para ajustar os valores da estrutura de dados usada pelo módulo IMPRGS, em caso de re-utilização do identificador da tabela.

Módulo ERREXC

Este módulo trata dos erros de execução. Foi modificado para retornar um código de erro ao invés de enviar uma mensagem para a tela.

Módulo IMPRGS

Este módulo tem como objetivo, no COPPEREL interativo, exibir todos os registros de uma relação, sendo acionado pelo comando MOSTRAR. No COPPEREL para linguagem hospedeira, seu código foi alterado para servir ao comando PROXIMO_REGISTRO.

Foi incluída uma estrutura de dados que mantém a identificação das tabelas já acessadas durante a transação acompanhadas do número do último registro acessado. Esta estrutura permite que sejam enviados comandos PROXIMO_REGISTRO intercaladamente para diversas relações, sem perder o contexto. Foram incluídas novas mensagens para indicar que a relação está vazia ou que todos os registros já foram retornados.

Se dois ou mais comandos PROXIMO_REGISTRO são emitidos para a mesma relação, sem a intercalação de um comando PROXIMO_REGISTRO para outra relação, um teste evita a consulta aos meta-dados que seria necessária para recuperar informações de formato, aumentando o desempenho.

Módulo INSRGS

Este é o responsável por ler do teclado um registro, e inserí-lo em uma relação. As mensagens que eram exibidas na tela foram suprimidas, criando-se novos códigos de retorno. Em caso de erro léxico no buffer que contém o registro a ser inserido, o controle é retornado a linguagem hospedeira, ao invés de continuar esperando uma entrada correta. Os outros módulos responsáveis pela análise são: CARATT e LEXCOT, cujas alterações foram similares às dos módulos CARACT e LEXICO.

Módulo LEXCOT

Este módulo é o analisador léxico para entrada de registros a serem inseridos na base de dados. Foi alterado de maneira similar ao módulo LEXICO.

Módulo LEXICO

Este módulo é o analisador léxico da LOPEREL. Foi alterado para reconhecer a nova classe 27, que representa o fim do buffer de comandos passado pela interface CPREL. Ao invés de continuar esperando novos caracteres, retorna à interface.

Módulo MSGERR

Esta rotina tem como objetivo emitir uma mensagem de erro, a partir de um código de erro. No COPPEREL interativo, ela só é invocada em caso de erro sintático. Sua ação é enviar uma mensagem para a tela, fechar o banco de dados e abortar a execução. Esta atitude foi tomada pois não tem sentido tratar erros sintáticos na linguagem hospedeira. Erros semânticos e de execução são, porém, sinalizados de volta ao programa hospedeiro.

Módulo SMNTC2

Este módulo é parte do analisador semântico da LOPEREL. Foi alterado para ajustar os valores iniciais da estrutura de dados usada pelo módulo IMPRGS quando a transação termina.

Módulo SNTTCO

Este módulo é o analisador sintático, que é o programa principal na versão interativa. Na versão para linguagem hospedeira, passou a ser uma sub-rotina que recebe como

parâmetro de entrada o comando LOPEREL e, opcionalmente, um registro para ser incluído no banco. Como parâmetro de saída é retornado um código de execução e, opcionalmente, um registro recuperado.

Diversos pontos desta rotina sofreram alterações similares. Nos pontos após a chamada ao semântico, em caso de erro, ao invés da rotina MSGERR ser invocada, um codigo de retorno é calculado sendo enviado de volta para a interface CPREL. Se houver mais de um erro no comando passado, seja este de natureza léxica, sintática, semântica ou de execução, apenas o primeiro erro é detectado, voltando o código deste.

APÊNDICE C

Exemplo de uma sessão com Campos Longos

```
$ CAMPO_LONGO
Base de Campos Longos
Tecle AJUDA para maiores informacoes
>AJUDA


Comandos Disponiveis :

    CRIAR

    MOSTRAR objid (inicio:fim)

    MODIFICAR objid (inicio:fim) PARA "literal"

    NO objid

    FOLHA folhaid

    AJUDA

    FIM

 Entre com o Comando para ajuda especifica
 ou <return> para voltar
Ajuda?...CRIAR

    CRIAR

       Cria um novo objeto no Banco de Objetos
         retornando sua identificacao (objid)

Ajuda?...MODIFICAR

    MODIFICAR objid (inicio:fim) PARA "literal"

     Troca o pedaco compreendido entre inicio
       e fim do objeto identificado por objid
       pelo literal fornecido entre aspas.

     Para eliminar um intervalo,
       troque pelo literal vazio: ""

     Para inserir um intervalo, omita
       o parametro fim. O intervalo e
       inserido apos o parametro inicio.

     Para inserir no inicio do objeto use o
       parametro inicio igual a zero.

Ajuda?...
>
```

```
>CRIAR
foi criado o objeto cuja raiz e          213
>
>MOSTRAR 213(1:10)
Conteudo Objeto          213
objeto tem tamanho            0
>
>AJUDA


Comandos Disponiveis :

    CRIAR

    MOSTRAR objid (inicio:fim)

    MODIFICAR objid (inicio:fim) PARA "literal"

    NO objid

    FOLHA folhaid

    AJUDA

    FIM

 Entre com o Comando para ajuda especifica
 ou <return> para voltar
Ajuda?...MOSTRAR

    MOSTRAR objid (inicio:fim)

     Exibe na tela o pedaco compreendido entre
       inicio e fim do objeto identificado
       por objid.

Ajuda?...
>
>MODIFICAR 213(0:) PARA
"Ha pontos do nosso literal que sao mantidos"
>
>MOSTRAR 213(1:100)
Conteudo Objeto          213
Ha pontos do nosso literal que sao mantidos
objeto tem tamanho           43
>
```

```
>AJUDA

Comandos Disponiveis :

    CRIAR

    MOSTRAR objid (inicio:fim)

    MODIFICAR objid (inicio:fim) PARA "literal"

    NO objid

    FOLHA folhaid

    AJUDA

    FIM

 Entre com o Comando para ajuda especifica
 ou <return> para voltar
Ajuda?...NO

    NO objid

      Exibe um dump contextual do no
        identificado por objid

      As informacoes incluem o tipo do no,
        o numero de entradas existentes e o
        conteudo destas
      Um no de tipo diferente de zero aponta
        para folhas

Ajuda?...FOLHA

    FOLHA folhaid

      Exibe o conteudo da folha identificada
        por folhaid.
      Todo o conteudo e exibido,
        aparecendo tambem bytes que foram
        logicamente eliminados.

Ajuda?...
>
>NO 213
 no      =          213
 tipo    =           -1
----------|---------c(i)--------p(i)-------
          1          11         644
          2          22         647
          3          33         646
          4          43         645
-----------------------------------------
>
>
```

```
FOLHA 644
conteudo =
Ha pontos d
>
>FOLHA 647
conteudo =
o nosso lit
>
>FOLHA 646
conteudo =
eral que sa
>
>FOLHA 645
conteudo =
o mantidos
>
>MODIFICAR 213(43:) PARA
" em segredo por seus visitantes. So ficam c"
>
>MOSTRAR 213(1:100)
Conteudo Objeto           213
Ha pontos do nosso literal que sao mantidos em segredo por
seus visitantes. So ficam c
objeto tem tamanho            86
>
>NO 213
 no       =           213
 tipo     =             0
----------|---------c(i)--------p(i)-------
           1           44          215
           2           86          214
-----------------------------------------
>
>NO 215
 no       =           215
 tipo     =           214
----------|---------c(i)--------p(i)-------
           1           11          644
           2           22          647
           3           33          646
           4           44          645
-----------------------------------------
>
>NO 214
 no       =           214
 tipo     =            -1
----------|---------c(i)--------p(i)-------
           1           11          651
           2           22          650
           3           32          649
           4           42          648
-----------------------------------------
>
>FOLHA 651
conteudo =
em segredo
>
```

```
>FOLHA 650
conteudo =
por seus vi
>
>FOLHA 649
conteudo =
sitantes.
>
>FOLHA 648
conteudo =
So ficam c
>
>MODIFICAR 213(86:) PARA
"onhecidos por pequenos grupos, pessoas que "
>
>MOSTRAR 213(1:200)
Conteudo Objeto          213
Ha pontos do nosso literal que sao mantidos em segredo por
seus visitantes. So ficam conhecidos por pequenos grupos,
pessoas que
objeto tem tamanho          129
>
>NO 213
 no      =          213
 tipo    =            0
----------|---------c(i)--------p(i)-------
          1           44          215
          2           87          214
          3          129          216
-----------------------------------------
>
>NO 215
 no      =          215
 tipo    =          214
----------|---------c(i)--------p(i)-------
          1           11          644
          2           22          647
          3           33          646
          4           44          645
-----------------------------------------
>
>NO 214
 no      =          214
 tipo    =          216
----------|---------c(i)--------p(i)-------
          1           11          651
          2           22          650
          3           32          649
          4           43          648
-----------------------------------------
>
>FOLHA 651
conteudo =
em segredo
>
>FOLHA 650
conteudo =
por seus vi
>
```

```
>FOLHA 649
conteudo =
sitantes.
>
>FOLHA 648
conteudo =
So ficam co
>
>NO 216
 no      =          216
 tipo    =           -1
----------|---------c(i)--------p(i)-------
          1          11          655
          2          22          654
          3          32          653
          4          42          652
-----------------------------------------
>
>FOLHA 655
conteudo =
nhecidos po
>
>FOLHA 654
conteudo =
r pequenos
>
>FOLHA 653
conteudo =
grupos, pe
>
>FOLHA 652
conteudo =
ssoas que
>
>MODIFICAR 213(129:) PARA
"gostariam de manter tranquilos seus paraiso"
>
>MOSTRAR 213(1:300)
Conteudo Objeto          213
Ha pontos do nosso literal que sao mantidos em segredo por
seus visitantes. So ficam conhecidos por pequenos grupos,
pessoas que gostariam de manter tranquilos seus paraiso
objeto tem tamanho          172
>
>NO 213
 no      =          213
 tipo    =            0
----------|---------c(i)--------p(i)-------
          1          44          215
          2          87          214
          3         130          216
          4         172          217
-----------------------------------------
>
```

```
>NO 217
 no       =          217
 tipo     =           -1
----------|---------c(i)--------p(i)-------
          1           11          659
          2           22          658
          3           32          657
          4           42          656
-----------------------------------------
>
>FOLHA 659
conteudo =
ostariam de
>
>FOLHA 658
conteudo =
 manter tra
>
>FOLHA 657
conteudo =
nquilos se
>
>FOLHA 656
conteudo =
us paraiso
>
>MODIFICAR 213(172:) PARA
"s praieiros. e uma destas cidades."
>
>MOSTRAR 213(1:300)
Conteudo Objeto          213
Ha pontos do nosso literal que sao mantidos em segredo por
seus visitantes. So ficam conhecidos por pequenos grupos,
pessoas que gostariam de manter tranquilos seus paraisos p
raieiros. e uma destas cidades.
objeto tem tamanho          206
>
>NO 213
 no       =          213
 tipo     =            0
----------|---------c(i)--------p(i)-------
          1           44          215
          2           87          214
          3          130          216
          4          173          217
          5          206          218
-----------------------------------------
>
>NO 218
 no       =          218
 tipo     =           -1
----------|---------c(i)--------p(i)-------
          1           11          662
          2           22          661
          3           33          660
-----------------------------------------
>
```

```
>FOLHA 662
conteudo =
 praieiros.
>
>FOLHA 661
conteudo =
 e uma dest
>
>FOLHA 660
conteudo =
as cidades.
>
>MOSTRAR 213(180:200)
Conteudo Objeto          213
iros. e uma destas ci
>
>MODIFICAR 213(184:) PARA " SAO FRANCISCO DO SUL"
>
>MOSTRAR 213(1:300)
Conteudo Objeto          213
Ha pontos do nosso literal que sao mantidos em segredo por
seus visitantes. So ficam conhecidos por pequenos grupos,
pessoas que gostariam de manter tranquilos seus paraisos p
raieiros. SAO FRANCISCO DO SUL e uma destas cidades.
objeto tem tamanho          227
>
>NO 213
 no      =          213
 tipo    =            0
----------|---------c(i)--------p(i)-------
          1           44          215
          2           87          214
          3          130          216
          4          173          217
          5          227          218
-----------------------------------------
>
>NO 218
 no      =          218
 tipo    =           -1
----------|---------c(i)--------p(i)-------
          1           11          662
          2           22          664
          3           32          663
          4           43          661
          5           54          660
-----------------------------------------
>
>FOLHA 663
conteudo =
SCO DO SUL
>
>FOLHA 662
conteudo =
 praieiros.
>
```

```
>FOLHA 664
conteudo =
 SAO FRANCI
>
>FOLHA 663
conteudo =
SCO DO SUL
>
>FOLHA 661
conteudo =
 e uma dest
>
>FOLHA 660
conteudo =
as cidades.
>
>MODIFICAR 213(227:) PARA " Fundada em 1504, foi colonizada"
>
>MOSTRAR 213(1:300)
Conteudo Objeto          213
Ha pontos do nosso literal que sao mantidos em segredo por
seus visitantes. So ficam conhecidos por pequenos grupos,
pessoas que gostariam de manter tranquilos seus paraisos p
raieiros. SAO FRANCISCO DO SUL e uma destas cidades. Funda
da em 1504, foi colonizada
objeto tem tamanho          259
>
>NO 213
 no      =          213
 tipo    =            0
----------|---------c(i)--------p(i)-------
          1          130          221
          2          259          220
-----------------------------------------
>
>NO 221
 no      =          221
 tipo    =            0
----------|---------c(i)--------p(i)-------
          1           44          215
          2           87          214
          3          130          216
-----------------------------------------
>
>NO 215
 no      =          215
 tipo    =          214
----------|---------c(i)--------p(i)-------
          1           11          644
          2           22          647
          3           33          646
          4           44          645
-----------------------------------------
>
>FOLHA 644
conteudo =
Ha pontos d
>
```

```
>FOLHA 647
conteudo =
o nosso lit
>
>FOLHA 646
conteudo =
eral que sa
>
>FOLHA 645
conteudo =
o mantidos
>
>NO 214
 no      =          214
 tipo    =          216
----------|---------c(i)--------p(i)-------
          1          11          651
          2          22          650
          3          32          649
          4          43          648
-----------------------------------------
>
>FOLHA 651
conteudo =
em segredo
>
>FOLHA 650
conteudo =
por seus vi
>
>FOLHA 649
conteudo =
sitantes.
>
>FOLHA 648
conteudo =
So ficam co
>
>NO 216
 no      =          216
 tipo    =          217
----------|---------c(i)--------p(i)-------
          1          11          655
          2          22          654
          3          32          653
          4          43          652
-----------------------------------------
>
>FOLHA 655
conteudo =
nhecidos po
>
>FOLHA 654
conteudo =
r pequenos
>
```

```
>FOLHA 653
conteudo =
grupos, pe
>
>FOLHA 652
conteudo =
ssoas que g
>
>NO 220
 no       =          220
 tipo     =            0
----------|---------c(i)--------p(i)-------
          1           43          217
          2           86          218
          3          129          219
-----------------------------------------
>
>NO 217
 no       =          217
 tipo     =          218
----------|---------c(i)--------p(i)-------
          1           11          659
          2           22          658
          3           32          657
          4           43          656
-----------------------------------------
>
>FOLHA 659
conteudo =
ostariam de
>
>FOLHA 658
conteudo =
 manter tra
>
>FOLHA 657
conteudo =
nquilos se
>
>FOLHA 656
conteudo =
us paraisos
>
>NO 218
 no       =          218
 tipo     =          219
----------|---------c(i)--------p(i)-------
          1           11          662
          2           22          664
          3           32          663
          4           43          661
-----------------------------------------
>
>FOLHA 662
conteudo =
 praieiros.
>
```

```
>FOLHA 664
conteudo =
 SAO FRANCI
>
>FOLHA 663
conteudo =
SCO DO SUL
>
>FOLHA 661
conteudo =
 e uma dest
>
>NO 219
 no       =          219
 tipo     =           -1
----------|---------c(i)--------p(i)-------
          1           11          660
          2           22          667
          3           33          666
          4           43          665
-----------------------------------------
>
>FOLHA 660
conteudo =
as cidades.
>
>FOLHA 667
conteudo =
 Fundada em
>
>FOLHA 666
conteudo =
 1504, foi
>
>FOLHA 665
conteudo =
colonizada
>
>MOSTRAR 213 (20:30)
Conteudo Objeto          213
litoral que
>
>MODIFICAR 213(23:23) PARA "o"
>
>MOSTRAR 213 (20:30)
Conteudo Objeto          213
litoral que
>
>MOSTRAR 213(76:184)
Conteudo Objeto          213
 So ficam conhecidos por pequenos grupos, pessoas que g
ostariam de manter tranquilos seus paraisos praieiros.
>
```

```
>MODIFICAR 213(76:184) PARA ""
folha desalocada:           655
folha desalocada:           654
folha desalocada:           653
folha desalocada:           652
no desalocado:          216
folha desalocada:           648
folha desalocada:           659
folha desalocada:           658
folha desalocada:           657
folha desalocada:           656
no desalocado:          217
folha desalocada:           664
>
>MOSTRAR 213(1:300)
Conteudo Objeto          213
Ha pontos do nosso litoral que sao mantidos em segredo por
seus visitantes. SAO FRANCISCO DO SUL e uma destas cidades.
Fundada em 1504, foi colonizada
objeto tem tamanho           150
>
>NO 213
 no      =           213
 tipo    =             0
----------|---------c(i)--------p(i)-------
          1           75          221
          2          150          220
-----------------------------------------
>
>NO 221
 no      =           221
 tipo    =             0
----------|---------c(i)--------p(i)-------
          1           44          215
          2           75          214
-----------------------------------------
>
>NO 215
 no      =           215
 tipo    =           214
----------|---------c(i)--------p(i)-------
          1           11          644
          2           22          647
          3           33          646
          4           44          645
-----------------------------------------
>
>FOLHA 644
conteudo =
Ha pontos d
>
>FOLHA 647
conteudo =
o nosso lit
>
>FOLHA 646
conteudo =
oral que sa
>
```

```
>FOLHA 645
conteudo =
o mantidos
>
>NO 214
 no      =          214
 tipo    =          218
----------|---------c(i)--------p(i)-------
          1          11          651
          2          22          650
          3          31          649
-----------------------------------------
>
>FOLHA 651
conteudo =
em segredo
>
>FOLHA 650
conteudo =
por seus vi
>
>FOLHA 649
conteudo =
sitantes.
>
>NO 220
 no      =          220
 tipo    =            0
----------|---------c(i)--------p(i)-------
          1          32          218
          2          75          219
-----------------------------------------
>
>NO 218
 no      =          218
 tipo    =          219
----------|---------c(i)--------p(i)-------
          1          11          662
          2          21          663
          3          32          661
-----------------------------------------
>
>FOLHA 662
conteudo =
 SAO FRANCI
>
>FOLHA 663
conteudo =
SCO DO SUL
>
>FOLHA 661
conteudo =
 e uma dest
>
```

```
>NO 219
 no       =          219
 tipo     =           -1
----------|---------c(i)--------p(i)-------
          1           11          660
          2           22          667
          3           33          666
          4           43          665
-----------------------------------------
>
>FOLHA 660
conteudo =
as cidades.
>
>FOLHA 667
conteudo =
 Fundada em
>
>FOLHA 666
conteudo =
 1504, foi
>
>FOLHA 665
conteudo =
colonizada
>
>MOSTRAR 213(1:9)
Conteudo Objeto          213
Ha pontos
>
>MODIFICAR 213(1:9) PARA "Existem diversos recantos"
>
>MOSTRAR 213(1:300)
Conteudo Objeto          213
Existem diversos recantos do nosso litoral que sao mantidos
em segredo por seus visitantes. SAO FRANCISCO DO SUL e uma
destas cidades. Fundada em 1504, foi colonizada
objeto tem tamanho          166
>
>NO 213
 no       =          213
 tipo     =            0
----------|---------c(i)--------p(i)-------
          1           91          221
          2          166          220
-----------------------------------------
>
>NO 221
 no       =          221
 tipo     =            0
----------|---------c(i)--------p(i)-------
          1           27          215
          2           60          222
          3           91          214
-----------------------------------------
>
```

```
>NO 222
 no       =           222
 tipo     =           214
----------|---------c(i)--------p(i)-------
           1           11          647
           2           22          646
           3           33          645
-----------------------------------------
>
>FOLHA 647
conteudo =
o nosso lit
>
>FOLHA 646
conteudo =
oral que sa
>
>FOLHA 645
conteudo =
o mantidos
>
>NO 221
 no       =           221
 tipo     =             0
----------|---------c(i)--------p(i)-------
           1           27          215
           2           60          222
           3           91          214
-----------------------------------------
>
>
>NO 215
 no       =           215
 tipo     =           222
----------|---------c(i)--------p(i)-------
           1            9          644
           2           18          669
           3           27          668
-----------------------------------------
>
>
>FOLHA 644
conteudo =
Existem d
>
>FOLHA 669
conteudo =
iversos r
>
>FOLHA 668
conteudo =
ecantos d
>
>MOSTRAR 213(52:90)
Conteudo Objeto          213
mantidos em segredo por seus visitantes
>
```

```
>MODIFICAR 213(52:90) PARA "escondidos"
folha desalocada:           650
no desalocado:          214
folha desalocada:           649
>
>MOSTRAR 213(1:300)
Conteudo Objeto          213
Existem diversos recantos do nosso litoral que sao escondi
dos. SAO FRANCISCO DO SUL e uma destas cidades. Fundada em
1504, foi colonizada
objeto tem tamanho           137
>
>
>NO 213
 no       =           213
 tipo     =             0
----------|---------c(i)--------p(i)-------
           1           62          221
           2          137          220
-----------------------------------------
>
>
>NO 221
 no       =           221
 tipo     =             0
----------|---------c(i)--------p(i)-------
           1           27          215
           2           62          222
-----------------------------------------
>
>
>NO 220
 no       =           220
 tipo     =             0
----------|---------c(i)--------p(i)-------
           1           32          218
           2           75          219
-----------------------------------------
>
>
>NO 218
 no       =           218
 tipo     =           219
----------|---------c(i)--------p(i)-------
           1           11          662
           2           21          663
           3           32          661
-----------------------------------------
>
>
 >
>FOLHA 651
conteudo =
s. segredo
>
>FOLHA 649
conteudo =
.itantes.
>
```

```
>MOSTRAR 213(1:300)
Conteudo Objeto           213
Existem diversos recantos do nosso litoral que sao escondi
dos. SAO FRANCISCO DO SUL e uma destas cidades. Fundada em
1504, foi colonizada
objeto tem tamanho           137
>
>
>MOSTRAR 213(1:62)
Conteudo Objeto           213
Existem diversos recantos do nosso litoral que sao escondi
dos.
>
>MODIFICAR 213 (1:62) PARA ""
folha desalocada:            669
folha desalocada:            668
folha desalocada:            647
folha desalocada:            646
folha desalocada:            645
no desalocado:             220
no desalocado:             222
no desalocado:             218
folha desalocada:            651
>
>MOSTRAR 213(1:300)
Conteudo Objeto           213
 SAO FRANCISCO DO SUL e uma destas cidades. Fundada em 1504,
foi colonizada
objeto tem tamanho            75
>
>
>NO 213
 no       =           213
 tipo     =             0
----------|---------c(i)--------p(i)-------
           1           21          215
           2           75          219
-----------------------------------------
>
>
>
>
>MOSTRAR 213 (30:75)
Conteudo Objeto           213
estas cidades. Fundada em 1504, foi colonizada
>
>MOSTRAR 213 (22:75)
Conteudo Objeto           213
 e uma destas cidades. Fundada em 1504, foi colonizada
>
>MODIFICAR 213(22:75) PARA ""
folha desalocada:            660
folha desalocada:            667
folha desalocada:            666
folha desalocada:            665
>
```

```
>MOSTRAR 213(1:100)
Conteudo Objeto          213
 SAO FRANCISCO DO SUL
objeto tem tamanho           21
>
>
>
>MODIFICAR 213(1:21) PARA ""
folha desalocada:          644
folha desalocada:          651
no desalocado:          215
folha desalocada:          662
folha desalocada:          663
folha desalocada:          661
no desalocado:          219
no desalocado:          213
>
>
>FIM
$
```