GUILHERME FERREIRA BONFIOLI

BANCO DE DADOS RELACIONAL E OBJETO-RELACIONAL: UMA COMPARAÇÃO USANDO POSTGRESQL

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para a obtenção do título de Bacharel em Ciência da Computação

LAVRAS
MINAS GERAIS - BRASIL
2006

GUILHERME FERREIRA BONFIOLI

BANCO DE DADOS RELACIONAL E OBJETO-RELACIONAL: UMA COMPARAÇÃO USANDO POSTGRESQL

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para a obtenção do título de Bacharel em Ciência da Computação

Área de Concentração:
Banco de Dados

Orientadora:
Profª. Olinda Nogueira Paes Cardoso

LAVRAS
MINAS GERAIS - BRASIL
2006

Ficha Catalográfica preparada pela Divisão de Processos Técnico da Biblioteca Central da UFLA

Bonfioli, Guilherme Ferreira

Banco de Dados Relacional e Bancos de Dados Objeto-relacional: Uma comparação usando PostgreSQL / Guilherme Ferreira Bonfioli. Lavras – Minas Gerais, 2006. 50p : il.

Monografia de Graduação – Universidade Federal de Lavras Departamento de Ciência da Computação

1. Informática 2. Banco de Dados 3. Orientação a Objeto

GUILHERME FERREIRA BONFIOLI

BANCO DE DADOS RELACIONAL E OBJETO-RELACIONAL: UMA COMPARAÇÃO USANDO POSTGRESQL

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para a obtenção do título de Bacharel em Ciência da Computação

Aprovada em 26 de abril de 2006.

_____
Prof. Rêmulo Maia Alves

_____
Prof. Guilherme Bastos Alvarenga

_____
Profa. Olinda Nogueira Paes Cardoso
(Orientadora)

LAVRAS
MINAS GERAIS - BRASIL

AGRADECIMENTOS

Agradeço a Deus, a meus pais e meus irmãos que de tudo fizeram para que eu pudesse estar aqui hoje e a todos que estiveram presentes comigo e que de alguma forma me apoiaram nesta caminhada. Deixo aos meus companheiros de república, colegas de sala, colegas de trabalho, professores e amigos um grande abraço. Sou grato a todos.

RESUMO

A maioria dos sistemas de bancos de dados desenvolvidos atualmente é baseada no modelo relacional. No entanto, novos modelos de Sistemas Gerenciadores de Banco de Dados (SGBD) têm surgido devido à demanda de novas aplicações e ao crescimento do paradigma de orientação a objetos, no cenário de desenvolvimento destas. Assim, surge a necessidade de se utilizar um SGBD que suporte estes conceitos e atenda a estas demandas das novas aplicações. O presente trabalho apresenta um estudo sobre banco de dados relacional, orientação a objetos, SGBD objeto-relacional e o padrão *Object Data Management Group* (ODMG). Com base nessas tecnologias foi elaborado um estudo comparativo de duas implementações distintas de uma mesma aplicação, utilizando diferentes modelagens no SGBD objeto-relacional, PostgreSQL.

Palavras-chaves: banco de dados, orientação a objetos, objeto-relacional, SGBD PostgreSQL.

ABSTRACT

Most of the database systems developed actually is based on the relational model. However, new models of Database Management Systems (DBMS) have been appearing due to the demand of new applications and to the growth of the object orientated paradigm. Like this, the need appears of using a DBMS that supports these concepts and assist these new applications demands. The present work presents a study on relational database model, object oriented concepts, object-relational DBMS and the pattern ODMG. Based on those technologies a comparative study of two different implementations from the same application was elaborated, using different models in the object-relational DBMS, PostgreSQL.

Keywords: database, object oriented, object-relational, DBMS PostgreSQL.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

BLOB - Binary Large Object

CAD - Computer Aided Design

CAM - Computer Aided Manufacturing

ODMG - Object Data Management Group

OID - Identificador Único de Objeto

OO - Orientado a Objeto

SGBD - Sistemas de Gerenciamento de Banco de Dados

SGBDOO - Sistemas de Gerenciamento de Banco de Dados Orientado a Objeto

SGBDOR - Sistemas de Gerenciamento de Banco de Dados Objeto-relacional

SQL - Structured Query Language

1. INTRODUÇÃO

O papel de um Sistema Gerenciador de Banco de Dados (SGBD) é fundamental no desenvolvimento de sistemas de informações modernos, de qualquer porte ou segmento. Atualmente, a maior parte dos bancos de dados desenvolvidos nos SGBDs baseiam-se no modelo relacional, que vem sendo muito bem sucedido no que se diz respeito ao desenvolvimento da tecnologia de banco de dados necessária para muitas aplicações convencionais. Entretanto o modelo relacional apresenta algumas limitações quando aplicações de banco de dados mais complexas precisam ser projetadas e implementadas.

Aplicações como banco de dados para engenharia e arquitetura, experiências científicas, telecomunicações, sistemas de informações geográficas e multimídia, dentre outras, possuem requisitos e características que diferem de aplicações empresariais convencionais, demandam estruturas de dados mais complexas que as tabelas relacionais e melhores técnicas de acesso a estes dados.

A programação orientada a objetos vem ganhando cada vez mais espaço no cenário de desenvolvimento de aplicações. Assim, surge a necessidade de se utilizar um sistema de banco de dados que também comporte os conceitos desta tecnologia.

Neste contexto novos modelos de banco de dados baseados no paradigma de orientação a objetos estão em ascendência visando atender à demanda dessas novas aplicações (Rodrigues Júnior, 2005). Dentre eles pode-se citar os Sistemas Gerenciadores de Banco de Dados Orientado a Objetos (SGBDOO) e os Sistemas Gerenciadores de Banco de Dados Objeto-Relacional (SGBDOR). Os SGBDORs combinam as características da programação orientada a objetos com as estruturas utilizadas no modelo relacional.

Num passado recente a escolha de qual SGBD utilizar em uma aplicação geralmente dependia de questões técnicas e do custo das licenças destes sistemas. Atualmente isso não é mais uma realidade devido ao surgimento de diversos SGBDs de qualidade e que são gratuitos e *open source*.

Software livre é uma realidade cada vez mais constante no mundo da informática. O mesmo fato ocorreu com banco de dados. Com um mercado de sistemas proprietários bastante consolidado, incluindo as grandes empresas como IBM, Microsoft e Oracle, os bancos de dados livres, tais como o MySQL e o PostgreSQL, começam a se destacar

solucionando assim o problema dos altos custos com a minimização destes (Oliveira, 2005).

O PostgreSQL foi escolhido para o desenvolvimento deste trabalho por ser um Sistema Gerenciador de Banco de Dados Objeto-Relacional (SGBDOR), ou seja, dá suporte tanto ao modelo de banco de dados relacional, quanto ao modelo objeto-relacional e por ser um sistema livre.

Atualmente a presença dos SGBDORs é freqüente e a sua escolha como opção para desenvolvimento de aplicações mais avançadas é natural. Porém surge o problema em decidir qual é o modelo da tecnologia de banco de dados a ser adotado no caso da implementação de aplicações convencionais.

1.1. Objetivos

Este trabalho tem como objetivo geral fazer uma comparação entre duas implementações distintas de um mesmo sistema de banco de dados, utilizando dois modelos bancos de dados: o relacional e o objeto-relacional. Para tanto, foi modelado um sistema de banco de dados de uma empresa de prestação de serviços de transporte de mudanças e de transporte de cargas.

Os objetivos específicos são: construir o modelo relacional do banco de dados e implementá-lo; construir o modelo objeto-relacional do banco de dados e implementá-lo; construir funções de manipulação de dados que explorem características orientadas a objeto no banco de dados e testá-las; comparar as duas implementações e analisar a partir das comparações se a adoção do modelo objeto-relacional para implementação de aplicações convencionais apresenta vantagens sobre o modelo relacional.

Dois bancos de dados foram construídos, um usando os conceitos do modelo relacional e outro usando os conceitos de modelo objeto-relacional. Ambos os bancos de dados foram implementados no SGBD PostgresSQL, que é um SGBDOR e portanto suporta os dois modelos, para posteriormente serem comparados e analisados.

Para fins de estudos os módulos de Cadastro de Cliente e de Cadastro de Pedido foram suficientes para apresentar os conceitos dos dois modelos.

No presente trabalho procura-se contribuir com uma pesquisa abrangente sobre a recente introdução da tecnologia de orientação a objetos na área de bancos de dados, em especial quando aplicada juntamente com a consolidada tecnologia de banco de dados relacional. Sendo assim, este trabalho apresenta a base teórica fundamental para a compreensão desta tecnologia de banco de dados e sua viabilidade no desenvolvimento de determinadas aplicações.

Após o estudo teórico, é realizada uma avaliação detalhada e cuidadosa das formas de implementação de cada um dos modelos. Com isto, obtém-se informações necessárias para orientar quem procura implantar a tecnologia de banco de dados objeto-relacional, iniciando um novo projeto ou migrando de um modelo relacional já existente.

1.2. Estrutura do Trabalho

Este trabalho está organizado da seguinte forma: o Capítulo 2 apresenta o referencial teórico, que está divido em seções que tratam do modelo relacional, de conceitos de orientação a objetos, do padrão ODMG (*Object Data Management Group*), de SGBDs objeto-relacionais e do SGBD PostgreSQL; o Capítulo 3 apresenta a metodologia utilizada para o desenvolvimento deste trabalho; o Capítulo 4 apresenta os resultados e as discussões que foram obtidos; e, por fim, o Capítulo 5 apresenta a conclusão .

2. REFERENCIAL TEÓRICO

O Sistema Gerenciador de Banco de Dados (SGBD), por essência, permite que bancos de dados sejam concorrentemente compartilhados por muitos usuários e aplicações utilizando manuseio de armazenamento e estratégias de otimização. Segundo Neves (2002) precisa-se manter os dados para executar as tarefas de leitura a partir de banco de dados, de atualização e de inserção dos dados no banco de dados, preservando a integridade dos mesmos. São exemplos de SGBDs contemporâneos: Oracle, Interbase, SQL Server, Firebird, Ingress, Sybase, PostgreSQL, MySQL, dentre muitos outros.

2.1. Modelo Relacional

Neste capítulo o SGBD é sustentado pelo modelo relacional, que é matematicamente conciso, completo, anti-redundante e consistente internamente. Ainda de acordo com Neves (2002), o banco de dados relacional é muito mais utilizado comercialmente e pode ser usado para resolver muitos problemas cotidianos.

O modelo relacional surgiu devido a seguintes necessidades: aumentar a independência de dados nos sistemas gerenciadores de banco de dados; prover um conjunto de funções apoiadas em álgebra relacional para armazenamento e recuperação de dados; permitir processamento *ad hoc*. Este modelo foi resultado de um estudo teórico realizado por Codd (1970), tendo por base a teoria dos conjuntos e álgebra relacional. O modelo foi apresentado num artigo publicado em 1970, mas que só nos anos 80, foi implementado.

O Modelo relacional revelou-se ser o mais flexível e adequado ao solucionar os vários problemas que se colocam ao nível da concepção e implementação da base de dados. A estrutura fundamental do modelo relacional é a relação. Uma relação é constituída por um ou mais atributos (campos), que traduzem o tipo de dados a armazenar. Cada instância do esquema (linha) designa-se por uma tupla (registro). O modelo relacional implementa estruturas de dados organizadas em relações (tabelas). Porém para trabalhar com essas tabelas algumas restrições tiveram que ser impostas para evitar aspectos indesejáveis no modelo relacional, tais como, repetição de informação,

incapacidade de representar parte da informação e perda de informação. Essas restrições são: integridade referencial, chaves, integridade de junções de relações.

Segundo Elmasri e Navathe (2005), o modelo relacional representa o banco de dados como uma coleção de relações. Informalmente cada relação se assemelha a uma tabela de valores, ou, até certo ponto, a um arquivo de registros "plano" (*flat file*).

Um modelo de dados basicamente é uma combinação de um conjunto de estruturas de dados, a tabela (ou relação); um conjunto de operadores que formam a linguagem de manipulação do banco de dados; e regras de integridade que são definidas no esquema do banco de dados, e são aplicadas nas instâncias do banco de dados.

Quando uma relação é imaginada na forma de uma tabela de valores, cada linha na tabela representa uma coleção de valores de dados relacionados. No modelo relacional, cada linha de uma tabela representa um fato que corresponde geralmente a uma entidade ou a um relacionamento do mundo real. O nome da tabela o os nomes das colunas são utilizados para auxiliar na interpretação do significado dos valores em cada linha.

Na terminologia formal do modelo relacional, uma linha é chamada de tupla, o título da coluna é denominado de atributo e a tabela é chamada de relação. O tipo de dado que descreve os tipos de valores que podem aparecer em cada coluna é denominado de domínio. A Figura 1 representa a terminologia citada anteriormente.

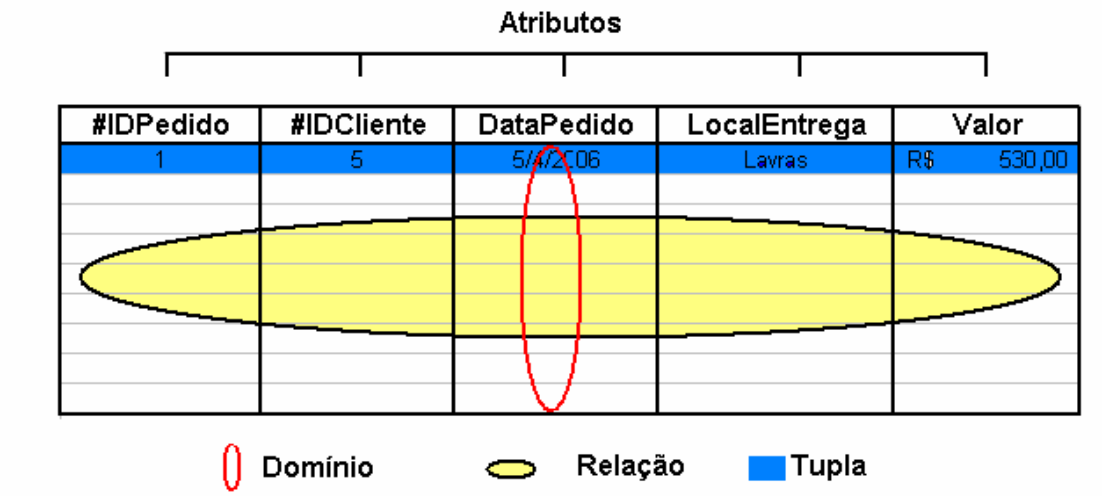

Figura 1: Representação dos conceitos do modelo relacional.
Fonte: Elaborado pelo autor.

2.1.1. Chaves Primárias e Chaves Estrangeiras

Um banco de dados relacional é composto por diversas tabelas que armazenam os dados operacionais dos sistemas em funcionamento no dia-a-dia da empresa ou organização. Dependendo do tipo e da quantidade de dados que se pretende armazenar, são criados diversos bancos de dados, um para cada domínio de aplicação.

As restrições de integridade do tipo identidade e referencial são mantidas basicamente pela utilização dos vários tipos de chaves.

Para estabelecer as restrições de integridade e estabelecer os relacionamentos entre as tabelas, utiliza-se um campo identificado como chave. Uma chave primária corresponde a uma ou várias colunas que não possuem valores duplicados dentro de uma tabela. Uma chave estrangeira corresponde a uma ou várias colunas em que os valores estejam identificados necessariamente como chave primária de outra tabela. A chave estrangeira é o mecanismo que define os relacionamentos em um banco de dados relacional (Date, 2000).

A chave primária poderá ser escolhida a partir do conjunto de chaves candidatas possíveis para aquela entidade, da mesma forma que as chaves candidatas representam a identificação exclusiva das tuplas daquela entidade. É bom lembrar que uma chave primária representa um valor único e não nulo (Date, 2000)

Uma chave estrangeira é um conjunto de atributos de uma relação R1 cujos valores devem corresponder a valores de alguma chave candidata de outra relação R2. Por exemplo, clientes e pedidos quando o pedido está associado a um cliente, o atributo IDCliente da relação pedidos identifica a associação com o cliente. Este atributo é então considerado uma chave estrangeira para este relacionamento. As chaves estrangeiras são ferramentas poderosas para a manutenção da integridade referencial do banco de dados. A Figura 2 apresenta um exemplo de Chave estrangeira.

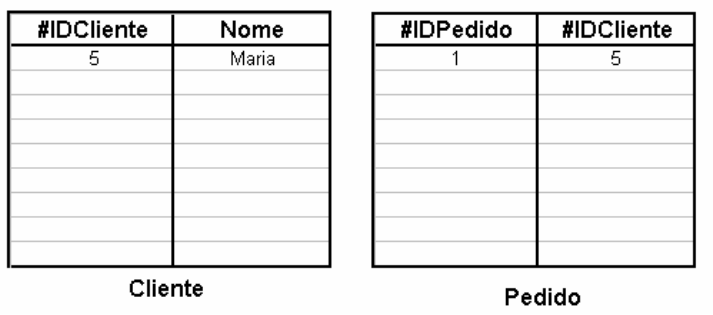

Figura 2: Exemplo de Chave Estrangeira.
Fonte: Elaborado pelo autor.

2.1.2. Integridade de Entidade e Integridade Referencial

Segundo Date (2000), termo integridade refere-se à precisão ou correção de dados no banco de dados. Nesse contexto "integridade" significa semântica e são as restrições de integridade que representam o significado dos dados.

A restrição de integridade de entidade é especificada em relações individuais e declara que nenhum valor da chave primária pode ser nulo. Isso se justifica porque o valor da chave primária sendo nula implica que não podermos identificar algumas tuplas (Elmasri e Navathe 2005). Por exemplo, no caso de duas ou mais tuplas tivessem valores nulos para suas chaves primárias, não seria possível distinguir entre elas.

A restrição de integridade referencial é especificada entre duas relações e é utilizada para manter a consistência entre tuplas destas relações. Informalmente, a restrição de integridade referencial declara que uma tupla em uma relação que se refere a uma outra relação deve se referir a uma tupla existente naquela relação (Elmasri e Navathe, 2005).

2.1.3. Limitações do modelo relacional

O avanço das aplicações impulsionadas pelo mercado de desenvolvimento de sistemas fez surgir novas características que foram adicionadas a elas. Surgiram as necessidades de se representar tipos de dados longos e complexos, expansão dos modelos de dados e linguagens de consulta para acomodar novos tipos de dados, suporte a transações longas, suporte ao processamento de conhecimento com tratamento de

restrições e gatilhos, contando também o aumento no grau de inter-relacionamentos entre os dados (Cardoso, 2003).

Alguns exemplos destas aplicações são aplicações para Automação de Escritório, Aplicações Médicas e Científicas, Aplicações de Geoprocessamento, Manufatura Industrial (CAM), Desenvolvimento de Software (CASE), Projetos de Circuitos Eletrônicos (CAD), Sistemas de Multimídia, Sistemas baseados em conhecimento e outras mais.

Os bancos de dados relacionais funcionam bem com tipos simples de dados, envolvendo poucas associações entre relações, entretanto apresentam algumas deficiências quando aplicações de banco de dados mais complexas precisam ser projetadas e implementadas.

Vale destacar as limitações quanto à ausência de tipos definidos pelo usuário: dificuldade em representar objetos do mundo real nas aplicações; incompatibilidade com as linguagens de programação; divergências entre os objetos da linguagem e as estruturas das entidades do modelo relacional; suporte a grandes bancos de dados; suporte a transações longas; e suporte a processamento de conhecimento. Os bancos de dados orientados a objetos foram propostos para fazer face às necessidades de aplicações mais complexas. O método de orientação a objetos oferece flexibilidade para lidar com algumas dessas exigências sem ser limitado pelos tipos de dados e linguagens de consulta disponíveis em sistemas de banco de dados convencionais (Elmasri e Navathe, 2005).

Para solucionar problemas gerados a partir de tais limitações surgiram os SGBBOO.

2.2. Conceitos de Orientação a Objetos

O Banco de Dados Orientado a Objetos integra o conceito de orientação a objetos com aptidões de banco de dados, oferecendo modelos diretos, atuais e intuitivos para o desenvolvimento de aplicações. Conforme Gualberto (2004, p. 2), o desenvolvimento dos Sistemas de Gerenciamento de Banco de Dados Orientado a Objetos (SGBDOO) teve origem na combinação de idéias dos modelos de dados tradicionais e de linguagens de programação orientada a objetos.

Segundo Larman (2000), a Orientação a Objeto (OO) é um paradigma para o desenvolvimento de aplicações. Ela aproxima a linguagem de computadores ao mundo

real, visando realizar a modelagem e codificação de uma forma mais natural. Com ela, o sistema fica organizado como uma coleção de objetos que interagem entre si.

A técnica de OO propõe uma maior reusabilidade, através da eliminação de redundância de código, com conseqüente aumento na produtividade; melhorias na manutenabilidade, pois as modificações no código são bem localizadas e não provoca descontrole em outras partes do software; e também maior confiabilidade, devido ao encapsulamento que impede o acesso direto aos dados (atributos) dos objetos (Larman, 2000).

Para Larman (2000), o objeto é o conceito central da abordagem orientada a objeto (OO). Ele pode ser entendido como algo do mundo real com limites bem definidos e identidade própria. Um objeto possui atributos (dados) e operações. Quando se define um objeto conceitual a partir de um objeto do mundo real, diz-se estar fazendo uma abstração.

Segundo Elmasri e Navathe (2005), objetos geralmente possuem dois componentes: estado (valor) e comportamento (operações). Portanto, ele é um tanto semelhante a uma variável de programa numa linguagem de programação, exceto pelo fato de que geralmente terá uma estrutura de dados complexa, bem como operações específicas definidas pelo programador. Objetos numa linguagem de programação OO existem somente durante a execução de um programa e são, por isso, chamados de objetos transientes.

De acordo com Larman (2000), classe é um conjunto de objetos que possuem características e comportamento em comum. Por exemplo, para os objetos “Maria”, “José”, e “Pedro” pode-se definir a classe “Pessoa”. Cada objeto deve ser criado antes de ser utilizado. O ato de criar um objeto é denominado instanciação. Todo objeto é instancia de uma classe. A instanciação de um objeto é feita através de um método especial denominado construtor. Todos os objetos instanciados terão a mesma estrutura de métodos e atributos de sua classe (Larman, 2000).

Ainda segundo Larman (2000), os atributos de um objeto são como variáveis que armazenam dados. Por exemplo, um objeto de uma classe “Pessoa” pode conter os atributos “nome” e “pedidos”, e os valores “Maria”, “Lista de pedidos”.

As operações são o que os objetos podem realizar. Para este mesmo objeto da classe “Pessoa”, pode-se ter, por exemplo, as operações “faz_pedido” e “lista_devedores”. Quando uma operação é solicitada ao objeto, diz-se que este objeto recebeu uma mensagem. Esta mensagem pode trazer consigo parâmetros, que são valores passados ao

objeto que a recebe. Por exemplo, traduzindo a mensagem “pessoa.faz_pedido(pedido)”, significa-se estar solicitando ao objeto “pessoa” para executar a operação “faz_pedido”, passando como parâmetro o “pedido”.

Conforme Larman (2000), o fato de uma classe reunir tanto as características quanto o comportamento dos objetos é chamado de encapsulamento. Assim, outros objetos só terão acesso às operações ou atributos que estiverem declarados como públicos.

Dá-se o nome de polimorfismo o fato de objetos de classes diferentes reagirem de forma particular a uma mesma operação (Larman, 2000). Por exemplo, a operação “faz_pedido” do objeto da classe “PessoaFisica” tem uma codificação diferente do objeto de uma classe “PessoaJuridica”, embora os objetivos da operação sejam semelhantes.

Um outro conceito muito importante da OO é a herança. Com ela, uma classe pode herdar todas as operações e atributos de uma outra classe, acrescentando os seus próprios atributos e operações. A classe que herdou a estrutura é chamada de subclasse, enquanto a que cedeu a estrutura chama-se superclasse. Por exemplo, a classe “Pessoa” pode ser herdada pela classe “PessoFisica”, acrescentando atributos como “CPF”, “DataNascimento”. Uma classe pode ter um atributo que se refere a uma outra classe. A essa referência, dá-se o nome de associação. Por exemplo, o atributo “End” de uma classe “Pessoa” pode referir a classe “Endereço”, que tenha mais atributos como “rua”, “numero”, “cidade”, “cep”, “estado”.

As seções a seguir apresentam uma explicação mais detalhada dos conceitos de OO utilizados nos SGBDs.

2.2.1. Identidade de Objeto

Segundo Elmasri e Navathe (2005), um sistema de banco de dados OO fornece uma identidade única pra cada objeto independente armazenado no banco de dados. Essa identidade única é geralmente implementada através de um identificador de objeto, ou OID único, gerado pelo sistema. O valor de um OID não é visível para o usuário externo, mas é utilizado pelo sistema para identificar cada objeto univocamente e para criar e gerenciar referências interobjeto.

A principal propriedade exigida por um OID é que ele seja imutável, isto é, o valor do OID para um objeto não deve se alterar, com isso preserva-se a identidade do objeto do

mundo real, que está sendo representado. Para se garantir a imutabilidade do OID, sistemas de gerenciamento de banco de dados OO devem possuir algum mecanismo para gerar OIDs. Continuando com a descrição de Elmasri e Navathe (2005), é desejável que cada OID seja utilizada apenas uma vez, ou seja, mesmo que um objeto seja removido do banco de dados, seu OID não deve ser atribuído a outro objeto. Essas duas propriedades implicam que o OID não deve depender de qualquer valore de atributo do objeto e nem deve ser baseado no endereço físico do objeto armazenado uma vez que esses valores, atributo e endereço físico podem ser modificados ou corrigidos. No entanto, alguns sistemas de fato utilizam o endereço físico como OID, com a finalidade de aumentar a eficiência da recuperação do objeto. Normalmente utiliza-se números inteiros grandes como OIDs e mapeia-se o valor do OID para o endereço físico do objeto utilizando alguma forma de tabela *hash*.

Conforme Elmasri e Navathe (2005), alguns modelos antigos de dados OO exigiam que tudo, desde um valor simples até um objeto complexo fosse representado como um objeto; sendo assim todo valor básico, tal como um inteiro (*integer*) ou uma *string* teria um OID. Isso permite que dois valores básicos possuam OIDs diferentes, o que pode ser útil em alguns casos. Por exemplo, o valor inteiro 50 pode ser utilizado para representar a idade de uma pessoa ou o peso de uma pessoa. Portanto dois objetos básicos com OIDs diferentes poderiam ser criados, porem ambos os objetos criados representariam o valor inteiro 50. Embora seja útil como modelo teórico, não é muito prático, uma vez que pode levar a geração de um número muito grande de OIDs. Dessa forma, a maioria dos sistemas de banco de dados OO admite a representação tanto de objetos como de valores. Todo o objeto deve possuir um OID imutável, enquanto um valor não possui nenhum OID e se mantém por si mesmo. O valor é geralmente armazenado dentro de um objeto e não pode ser referenciado a partir de outros objetos.

2.2.2. Estrutura do Objeto

Conforme Elmasri e Navathe (2005), em banco de dados OO, o estado, valor corrente de um objeto complexo, pode ser construído a partir de outros objetos ou outros valores, utilizando certos construtores de tipo. Os objetos podem ser visualizados como um trio (i,c,v), onde "i" é um identificador de objeto único (OID), "c" é o construtor de tipo (ou seja, indicação de como o estado do objeto é construído) e "v" é o estado do objeto (ou valor corrente).

Os construtores geralmente utilizados são *atom* (átomo), tupla, *set* (conjunto), mas outros construtores como *list, bag e array* também podem ser utilizados. Os construtores do tipo set (conjunto), *list*, *bag* e *array* são chamados de tipos de coleção para distingui-los dos tipos básicos e das tuplas. A principal característica de um tipo coleção é que o estado do objeto será uma coleção de objetos que podem ser não ordenados (como *set* ou *bag*) ou ordenados (como *list* ou *array*). O construtor do tipo *tupla* é geralmente chamado de tipo estruturado (*struct type*), uma vez que corresponde ao construtor de *struct* nas linguagens de programação C e C++.

O construtor de átomos é utilizado para representar todos os valores atômicos básicos, como números inteiros, números reais, *strings* de caracteres, booleanos e quaisquer outros tipos de dados que o sistema diretamente suporta (Elmasri e Navathe, 2005).

O estado do objeto "v" de um objeto (i, c, v), é interpretado com base o construtor "c".

- Para c=*atom*, o estado (valor) "v" é um valor atômico do domínio de valores suportados pelo sistema.
- Para c=*set*, o estado "v" é um conjunto de identificadores de objeto $\{i_1, i_2, i_3..., i_n\}$ que são os OIDs para um conjunto de objetos que são tipicamente do mesmo tipo.
- Para c=tupla, o estado "v" é uma tupla da forma $<a_1:i_1, a_2:i_2 ... a_n:i_n>$ onde cada aj é um nome de atributo e cada ij é um OID.
- Para c=*list*(lista), o valor "v" é uma lista ordenada $\{i_1, i_2,..., i_n\}$ de OIDs de objetos do mesmo tipo e estes estão ordenados.

- Para c=*array*, o estado do objeto é uma disposição (*array*) unidimensional de identificadores de objeto. A principal diferença entre uma *array* e uma lista é que uma lista pode possuir um número arbitrário de elementos, enquanto que um *array* geralmente possui um tamanho máximo e a diferença entre set e o *bag* é que todos os elementos num conjunto devem ser diferentes, enquanto que em uma *bag* pode possuir elementos duplicados.

2.2.3. Objetos Complexos

Segundo Elmasri e Navathe (2005), a capacidade de lidar com objetos complexos foi um dos principais motivos para a criação do modelo de banco de dados orientado a objetos, pois na época não havia nenhum sistema que oferecesse tal recurso.

Existem dois tipos de objetos complexos: os não-estruturados e os estruturados.

Um objeto complexo não-estruturado é um objeto que utiliza um tipo que não é padrão no banco de dados e podem ocupar uma grande área de armazenamento.

Estes objetos são do tipo BLOB, que significa *Binary Large Object*, e podem armazenar, por exemplo, os dados referentes a uma imagem *bitmap*, um texto formatado, uma música entre outros.

Elmasri e Navathe (2005), afirmam que, por padrão, os SGBDOOs não interpretam os dados contidos nestes objetos, fornecendo apenas a funcionalidade de recuperar as informações parciais ou totais deste objeto para a aplicação. No entanto, o SGBDOO permite a criação de um tipo abstrato para estes tipos de objetos, e implementar as operações necessárias para definir o comportamento deste objeto. Assim, conclui-se que um SGBDOO é um sistema de tipo extensível, ou seja, pode-se criar bibliotecas de tipos de objetos abstratos, e reutilizar em outras aplicações.

Para exemplificar, supondo-se que se deseja armazenar dados referentes às impressões digitais em um banco de dados de investigações policiais. É necessário utilizar um objeto complexo não-estruturado. Assim, implementam-se as operações, por exemplo, a que faz o processamento de como localizar uma digital dentre as armazenadas no banco. Já um objeto complexo estruturado difere do não-estruturado pelo fato do SGBDOO conhecer a estrutura interna deste objeto. Ele é constituído da aplicação dos diversos construtores, de forma repetida. Por exemplo, um objeto complexo da classe CEP pode

conter os objetos locais numero_cep e logradouro, objetos que referenciam a outros objetos, como cidade e estado, e objetos que são conjunto de objetos, como bairros.

2.2.4. Encapsulamento, Nomeação e Acessibilidade

Para Elmasri e Navathe (2005), o conceito de ocultação e encapsulamento de informação podem ser aplicados em objetos do banco de dados. A idéia principal é definir o comportamento de um tipo de objeto baseado nas operações que podem ser extremamente aplicadas aos objetos daquele tipo. A estrutura interna do objeto fica oculta e o objeto somente se torna acessível através de uma série de operações predefinidas. Algumas dessas operações podem ser utilizadas para criar ou destruir objetos, outras operações podem atualizar o estado do objeto, recuperar parte de um objeto ou aplicar cálculos sobre o objeto. A combinação entre as operações de recuperação, cálculo e atualização também são possíveis.

Os usuários externos do objeto se tornam cientes apenas da interface de tipo de objeto, que define os nomes e argumentos (parâmetros) de cada operação. A implementação fica oculta aos usuários externos; ela inclui a definição das estruturas de dados internas do objeto e a implementação das operações que acessam essas estruturas. Na terminologia OO, a parte de interface de cada operação é chamada de assinatura, a implementação da operação é chamada de método. Geralmente, um método é invocado enviando-se uma mensagem para o objeto, para que seja executado o método correspondente.

De acordo com Elmasri e Navathe (2005), o requisito de que todos os objetos de uma aplicação de banco de dados sejam completamente encapsulados é bastante rígido. Uma maneira de relaxar este requisito é dividir a estrutura de um objeto entre atributos (variáveis de instância) visíveis e ocultos. Atributos visíveis podem ser diretamente acessados para a leitura por parte de operadores externos ou por uma linguagem de consulta de alto nível. Os atributos ocultos de um objeto são completamente encapsulados e somente podem ser acessados através de operações predefinidas.

Continuando com as definições de Elmasri e Navathe (2005), o termo classe é geralmente utilizado para fazer referência a uma definição de tipo de objeto, juntamente com as definições das operações para aquele tipo. Um número de operações é declarado para cada classe de objeto e a assinatura (interface) de cada operação é incluída na

definição da classe. Um método (implementação) para cada operação dever ser definido em outro lugar, utilizando uma linguagem de programação. Operações típicas incluem a operação *object constructor* (construtora de objeto), que é utilizada para criar um novo objeto e a operação *destructor* (destruidora), que é utilizada para destruir um objeto. Outras operações também podem ser declaradas, como as operações modificadoras de objeto que modificam atributos de um objeto e operações de recuperação de informação sobre o objeto.

Uma operação geralmente é aplicada a um objeto, através da utilização da notação de ponto (.). Se "d" é uma referencia a um objeto qualquer, podemos invocar uma operação "op1" como "d.op1". Essa notação também é utilizada para fazer referência a atributos de um objeto.

Geralmente objetos são criados por um programa de aplicação em execução, mas nem todos os objetos devem ser armazenados no banco de dados. Objetos transientes existem durante a execução de um programa e desaparecem uma vez terminada a execução do programa. Objetos persistentes são armazenados no banco de dados e persistem mesmo após a execução do programa.

Segundo Elmasri e Navathe (2005), os SGBDOO possuem dois mecanismos para persistir um objeto: nomeação e persistência.

O mecanismo de nomeação envolve dar a um objeto um nome persistente único, através do qual ele possa ser recuperado por aquele e outros programas. Todos esses nomes dados a objetos devem ser únicos dentro de um determinado banco de dados. Dessa forma os objetos persistentes nomeados são utilizados como ponto de entrada para o banco de dados, através dos quais usuários e aplicações possam acessá-los no banco de dados.

O mecanismo de acessibilidade opera tornando o objeto acessível a partir de algum objeto persistente. Um objeto "B" é dito acessível a partir de um objeto "A" se uma seqüência de referências no gráfico do objeto conduzirem do objeto "A" ao objeto "B". Por exemplo, suponha numa aplicação de controle de pedidos de serviço, um banco de dados que mantém os serviços oferecidos por esta empresa. Assim, têm-se a classe Serviços, com os atributos nome, origem, destino e preço de viagem. Pode-se então criar uma nomeação chamada TodosServiços, que é um conjunto (*set*) de objetos da classe Serviços. Desta forma, todos os produtos podem ser adicionados a este conjunto, e tornam-se automaticamente persistentes devido à acessibilidade.

2.2.5. Hierarquias de Tipo e Herança

Na maioria das aplicações de banco de dados, existem inúmeros objetos do mesmo tipo ou classe. Dessa forma, banco de dados orientados a objetos deve oferecer a capacidade de classificar objetos com base em seus tipos.

Segundo Elmasri e Navathe (2005), sistemas de banco de dados orientados a objeto devem permitir a definição de novos tipos baseados em outros tipos predefinidos, conduzindo a uma hierarquia de tipo.

Quando um projetista ou usuário precisam criar um novo tipo que seja semelhante, mas não idêntico a um tipo já definido temos o conceito de subtipo.

Segundo Elmasri e Navathe (2005), um subtipo herda todas as funções de um tipo predefinido, que é denominado supertipo. Por exemplo sejam os tipos:
PESSOA FÍSICA: Nome, Endereço, Telefone, DataCadastro, DataDeNasc, Idade, CPF.
PESSOA JURIDICA: Nome, Endereço, Telefone, DataCadastro, CGC, InscricaoEstadual.

Note que os atributos Nome, Endereço, Telefone e DataCadastro existem tanto na classe PESSOA FÍSICA quanto PESSOA JURIDICA. Assim, deve-se utilizar uma hierarquia de tipo, fazendo com que PESSOA JURIDICA e PESSOA JURIDICA sejam subtipos da classe PESSOA, por exemplo:
PESSOA: Nome, Endereço, Telefone, DataCadastro
PESSOA FÍSICA: subtype-of PESSOA: DataDeNasc, Idade, CPF.
PESSOA JURIDICA subtype-of PESSOA: CGC, InscricaoEstadual.

Sistemas de Banco de Dados OO que seguem o padrão ODMG (*Object Data Management Group*), a ser apresentado na Seção 2.3, permitem criar extensões para os tipos. Uma extensão é um objeto persistente que possui um conjunto de todos os objetos da classe. O Banco de Dados impõe uma restrição à extensão, onde os objetos da extensão de uma classe devem ser um subconjunto da extensão de sua superclasse (Elmasri e Navathe, 2005).

2.2.6 Polimorfismo

Segundo Larman (2000), os bancos de dados OO também estão preparados para o conceito de polimorfismo, ou sobrecarga de operador. Este conceito permite que um mesmo nome de operação, aplicado a objetos de classes diferentes e que tenham uma superclasse em comum, possa ter implementações diferentes. Quando uma mensagem é passada ao objeto, o SGBDOO seleciona a implementação adequada de acordo com o tipo do objeto.

2.3. O Padrão ODMG

A estrutura padrão para os bancos de dados objeto foi feita pelo Grupo de gerenciamento dados objetos (ODMG - *Object Data Management Group*). Esse grupo é formado por representações da maioria dos fabricantes no mercado de banco de dados objeto. Membros do grupo estão comprometidos a incorporar o padrão em seus produtos. O termo Modelo Orientado Objetos é usado para o documento padrão que contém a descrição geral das facilidades de um conjunto de linguagens de programação orientadas a objetos e a biblioteca de classes que pode formar a base para o Sistema de banco de Dados.

Com a criação dos SGBDOOs de diversos fabricantes, surgiu a necessidade de estabelecer um padrão para esta nova tecnologia. Assim, os sistemas passariam a ser mais portáveis, pois uma aplicação desenvolvida sobre a base de um BDOO poderia facilmente ser adaptada para um outro sistema de banco de dados. Além disto, é interessante haver a interoperabilidade, ou seja, acessar diversos sistemas de banco de dados orientado a objeto utilizando uma mesma linguagem, (Elmasri e Navathe, 2005).

Para tanto, os fabricantes de sistemas de banco de dados orientados a objeto se reuniram e criaram o padrão ODMG. Este padrão foi criado em 1993 com a versão 1.0, e foi revisto novamente para estabelecer a versão 2.0, que é a mais atual (Silberschatz, 1999).

Objetos e literais são blocos de construção básicos do modelo de objetos. A principal diferença entre eles é que um objeto possui um identificador do objeto e um valor atual e o literal possui apenas um valor, (Elmasri e Navathe, 2005).

Para Silberschatz (1999), um literal pode ser de três tipos: atômico, estruturado ou de coleção. Os literais atômicos são valores de tipos predefinidos no banco, tais como inteiro, caracter, ponto flutuante, cadeia de caracteres (*string*), enumeração, etc. Os literais estruturados armazenam uma estrutura de dados, tais como hora (por exemplo, que é estruturado em hora, minuto, segundo e milisegundo), data e *timestamp*. Já os literais de coleção permitem armazenar uma coleção de literais ou objetos. São eles o Set, Bag, Array, que já foram explanados na Seção 2.2.2., e o *Dictionary*, que mantém uma coleção de associações do tipo chave - valor. Este literal de coleção geralmente é utilizado para criar um índice em uma coleção de valores.

O objeto na ODMG possui quatro características: identificador, estrutura, tempo de vida e estado. O identificador é o OID, ou um nome único no banco que referencia esta OID. A estrutura compreende os atributos e operações do qual o objeto é formado. O tempo de vida diz respeito se o objeto é transiente ou persistente. E o estado é o valor contido neste objeto (Silberschatz, 1999).

Segundo o mesmo autor, a ODMG permite a utilização de interfaces. Elas não são instanciáveis, ou seja, não é possível criar objetos a partir delas. Já as classes possuem construtores de objetos, que podem ser persistidos no banco de dados. Existem dois tipos de heranças na ODMG. A primeira ocorre quando uma classe ou interface herda de uma outra interface. Esta herança é declarada pela palavra-chave *extents*, e neste caso permite que seja feita herança múltipla (Silberschatz, 1999). Note que a herança de uma interface herda somente as assinaturas de operações e atributos finais (constantes). Portanto, a implementação deve ser feita na sub-classe da herança. Outro tipo de herança ocorre somente entre classes, e é explicitada pela palavra *extends*. Neste caso, não é permitida herança múltipla.

No modelo ODMG, todos os objetos herdam a interface *Object*. Esta possui 3 operações: "*same_as*", que compara a identidade do objeto com outro; "*copy*", que efetua a copia do objeto em outro, e "*delete*", que elimina o objeto. Segundo Silberschatz (1999), para objetos de coleção, a ODMG possui também uma hierarquia de interfaces. Todos os objetos de coleção herdam da interface "*Collection*", que disponibiliza as operações padrões para todas as demais interfaces de coleção. São elas "*Set*", "*Bag*", "*List*", "*Array*" e "*Dictionary*". Outro recurso que a ODMG especifica é a declaração de relacionamentos entre objetos. Estes só podem ser utilizados em relacionamentos binários (Silberschatz,

1999). Para exemplificar, suponha duas classes: Produto e Categoria. Na classe Produto, tem-se um atributo chamado categoria, que irá armazenar o OID de um objeto da classe categoria. Já este objeto da classe categoria terá um atributo denominado possui_produtos, que irá armazenar uma coleção (*Set*) dos produtos desta categoria. A integridade deste relacionamento é gerenciada automaticamente pelo banco de dados. Ainda na declaração das classes de objeto, a ODMG apresenta mais três conceitos: Extensões, Chaves e Objetos de Fábrica.

Para Silberschatz (1999), extensões são declaradas através da palavra chave *extent*. Ela faz com que o sistema crie um objeto com o nome especificado, que será uma coleção de todos os objetos desta classe. Uma chave é declarada através da palavra-chave *key*. Ela é utilizada para especificar que um ou mais atributos na classe serão exclusivos dos demais objetos criados por esta classe. Esta restrição será controlada automaticamente pelo sistema de banco de dados (Silberschatz, 1999). Por fim, o conceito de Objetos de Fábrica diz respeito à criação de construtores de objetos. A ODMG disponibiliza uma interface padrão chamada *ObjectFactory*, que possui a operação new. Assim, o usuário pode implementar a operação *new* diferentemente para cada tipo de objeto.

2.4. SGBDs Objeto-Relacionais

Quando os bancos de dados orientados a objetos foram introduzidos, algumas das falhas perceptíveis do modelo relacional parecem ter sido solucionadas com esta tecnologia e acreditava-se que tais bancos de dados ganhariam grande parcela do mercado. Porém hoje acredita-se que os Bancos de Dados Orientados a Objetos serão usados em aplicações especializadas, enquanto os sistemas relacionais continuarão a sustentar os negócios tradicionais nos quais as estruturas de dados baseadas em relações são suficientes.

A área de atuação dos SGBDs Objeto-Relacional tenta suprir a dificuldade dos sistemas relacionais convencionais, que é o de representar e manipular dados complexos. A solução proposta é a adição de facilidades para manusear tais dados utilizando-se das facilidades SQL existentes. Para isso foi necessário adicionar: extensões dos tipos básicos no contexto SQL; representações para objetos complexos no contexto SQL; herança no contexto SQL; sistema para produção de regras.

2.5 PostgreSQL

Toda esta seção foi baseada em documentos extraídos do *The Postgresql Global Development Group* (Postgresql, 2006)**.**

O PostgreSQL é um Sistema Gerenciador de Banco de Dados Objeto-Relacional (SGBDOR) desenvolvido no Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley. O PostgreSQL descende do POSTGRESS, código original de Berkeley, possuindo o código fonte aberto. Fornece suporte às linguagens SQL92/SQL99, além de outras funcionalidades avançadas. O POSTGRESS foi pioneiro em muitos conceitos orientado a objeto que agora estão se tornando disponíveis em alguns bancos de dados comerciais.

Os SGBDs tradicionais suportam um modelo de dados composto por uma coleção de relações com nome, contendo atributos de um tipo específico. Nos sistemas comerciais em uso, os tipos possíveis incluem número de ponto flutuante, inteiro, cadeia de caracteres, valores monetários e data. É amplamente reconhecido que este modelo não é adequado para aplicações futuras de processamento de dados. O modelo relacional substituiu com sucesso os modelos anteriores principalmente pela sua simplicidade. Entretanto, esta simplicidade tornou a implementação de certas aplicações muito difícil. O PostgreSQL oferece recursos adicionais pela incorporação de vários conceitos e funcionalidades. Dentre elas, podemos destacar:

- herança (especialização e generalização)
- avançado sistema de tipos de dado
- funções
- restrições
- gatilhos
- integridade da transação

Estas funcionalidades colocam o PostgreSQL dentro da categoria de banco de dados referida como objeto-relacional. Repare que é diferente daqueles referidos como orientados a objetos que, em geral, não são muito adequados para apoiar as linguagens tradicionais de banco de dados relacional. Portanto, embora o PostgreSQL possua algumas funcionalidades de orientação a objetos, está firmemente ligado ao mundo dos bancos de

dados relacionais. Na verdade, alguns bancos de dados comerciais incorporaram recentemente funcionalidades nas quais o PostgreSQL foi pioneiro.

2.5.1. Breve Histórico

O sistema de gerenciamento de banco de dados objeto-relacional hoje em dia conhecido por PostgreSQL, e por um breve período de tempo chamado Postgres95, é derivado do pacote POSTGRESS, que utiliza um SGBDOO. O PostgreSQL é considerado o mais avançado banco de dados de código aberto disponível atualmente, oferecendo controle de concorrência multiversão, suportando praticamente todas as construções do SQL, incluindo subconsultas, transações, tipos definidos pelo usuário e funções, e dispondo de um amplo conjunto de ligações com linguagens procedurais, incluindo C, C++, Java, Perl, Tcl e Python.

O POSTGRES passou por várias versões desde então. A primeira versão de demo do sistema ficou operacional em 1987, e foi exibida em 1988 na Conferência ACM-SIGMOD. A versão 1, descrita em Stonebraker, Lawrence e Hirohama (1990), foi liberada para alguns poucos usuários externos em junho de 1989. A versão 2 foi liberada em junho de 1990, contendo um novo sistema de regras. A versão 3 surgiu em 1991 adicionando suporte a múltiplos gerenciadores de armazenamento, um executor de consultas melhorado, e um sistema de regras reescrito. Para a maior parte, as versões seguintes até o Postgres95 focaram a portabilidade e a confiabilidade.

O POSTGRES foi usado para implementar muitas aplicações diferentes de pesquisa e produção, incluindo: sistema de análise de dados financeiros, pacote de monitoramento de desempenho de turbina a jato, banco de dados de acompanhamento de asteróides, banco de dados de informações médicas, além de vários sistemas de informações geográficas. O POSTGRES também foi usado como ferramenta educacional em diversas universidades. Por fim, a Illustra Information Technologies (posteriormente incorporada pela Informix, que agora pertence à IBM) pegou o código e comercializou. O POSTGRES se tornou o principal gerenciador de dados do projeto de computação científica Sequoia 2000 no final de 1992.

O tamanho da comunidade de usuários externos praticamente dobrou durante o ano de 1993. Começou a ficar cada vez mais óbvio que a manutenção do código do protótipo e seu suporte estavam consumindo grande parte do tempo que deveria ser dedicado às

pesquisas sobre banco de dados. Em um esforço para reduzir esta sobrecarga de suporte, o projeto do POSTGRES terminou oficialmente com a versão 4.2.

2.5.2. O Postgres95

Andrew Yu e Jolly Chen em 1994 adicionaram um interpretador da linguagem SQL ao POSTGRES, tornando-se um SGBDOR. O Postgres95 foi em seguida liberado para a Web para encontrar seu caminho no mundo como descendente de código aberto do código original do POSTGRES.

O código do Postgres95 era totalmente escrito em ANSI C e reduzido em tamanho em 25%. Muitas mudanças internas melhoraram o desempenho e a facilidade de manutenção. O Postgres95 versão 1.0.x era 30-50% mais rápido que o POSTGRES versão 4.2, utilizando o Wisconsin Benchmark. Além da correção de erros, as principais melhorias foram a linguagem de consultas PostQUEL foi substituída pelo SQL (implementado no servidor); as subconsultas não foram permitidas até o PostgreSQL, mas podiam ser simuladas no Postgres95 por meio de funções SQL definidas pelo usuário; as agregações foram re-implementadas; o suporte para a cláusula *GROUP BY* das consultas também foi adicionado.

Além do programa monitor, um novo programa, psql, foi disponibilizado para consultas SQL interativas utilizando o *Readline* do GNU.

Um breve tutorial introduzindo as funcionalidades regulares da linguagem SQL, assim como as do Postgres95, foi distribuído junto com o código fonte. O utilitário *make* do GNU (no lugar do *make* do BSD) foi utilizado para a geração. Além disso, o Postgres95 podia ser compilado com o GCC sem correções (o alinhamento de dados para a precisão dupla foi corrigido).

2.5.3. O PostgreSQL

Em 1996, por questões de conveniência, foi escolhido um novo nome para substituir o nome Postgre95. O nome PostgreSQL foi escolhido para refletir o relacionamento entre o POSTGRES original e as versões mais recentes com funcionalidade SQL. Ao mesmo tempo, foi mudado o número da versão para começar em 6.0, colocando os números na

seqüência original começada pelo projeto POSTGRES.

A ênfase durante o desenvolvimento do Postgres95 era identificar e compreender os problemas existentes no código do servidor. Com o PostgreSQL, a ênfase foi mudada para a melhoria das funcionalidades e dos recursos, embora o trabalho continuasse em todas as áreas.

As principais melhorias no PostgreSQL incluem: o bloqueio no nível de tabela foi substituído por um sistema de concorrência multi-versão, permitindo os que estão lendo continuarem a ler dados consistentes durante a atividade de escrita, possibilitando efetuar cópias de segurança utilizando o *pg_dump* enquanto o banco de dados se mantém disponível para consultas; a implementação de funcionalidades importantes no servidor, incluindo subconsultas, padrões, restrições e gatilhos, *triggers*. A incorporação de funcionalidades adicionais compatíveis com a linguagem SQL92, incluindo chaves primárias, chaves estrangeiras, identificadores entre aspas, conversão implícita de tipo de cadeias de caracteres literais, conversão explícita de tipos e inteiros binário e hexadecimal. Os tipos nativos foram aperfeiçoados, incluindo vários tipos para data, hora, vetores e matrizes, e suporte adicional para tipos geométricos.

Transação é um conceito fundamental de todo sistema de banco de dados. O ponto essencial da transação é englobar vários passos em uma única operação de tudo ou nada. Os estados intermediários entre os passos não são vistos pelas demais transações simultâneas e, se ocorrer alguma falha que impeça a transação chegar até o fim, então nenhum dos passos intermediários irá afetar o banco de dados de forma alguma.

O gatilho, *trigger*, pode ser definido para executar antes ou depois de uma operação de *insert*, *update* ou *delete*, tanto uma vez para cada linha modificada quanto uma vez por instrução SQL. Quando ocorre o evento do gatilho, a função de gatilho é chamada no momento apropriado para tratar o evento. A função *trigger* deve ser criada antes do *trigger*.

A linguagem SQL é a que o PostgreSQL, e a maioria dos bancos de dados relacionais, utiliza como linguagem de comandos. Entretanto, todas as declarações SQL devem ser executadas individualmente pelo servidor de banco de dados. Isto significa que o aplicativo cliente deve enviar o comando para o servidor de banco de dados, aguardar que seja processado, receber os resultados, realizar algum processamento, e enviar o próximo comando para o servidor. Tudo isto envolve comunicação entre processos e pode,

também, envolver tráfego na rede se o cliente não estiver na mesma máquina onde se encontra o servidor de banco de dados.

Usando a linguagem PL/pgSQL pode ser agrupado um bloco de processamento e uma série de comandos dentro do servidor de banco de dados, juntando o poder da linguagem procedural com a facilidade de uso da linguagem SQL, e economizando muito tempo, porque não há necessidade da sobrecarga de comunicação entre o cliente e o servidor. Isto pode aumentar o desempenho consideravelmente. Surge daí o conceito de procedimento armazenado, *stored precedure.*

Também podem ser utilizados na linguagem PL/pgSQL todos os tipos de dados, operadores e funções da linguagem SQL.

3. METODOLOGIA

3.1. Tipo de Pesquisa

A classificação dos tipos de pesquisas varia de acordo com o enfoque dado, segundo interesses, condições, campos e objetivos. Cabe ao pesquisador a escolha do método que melhor se aplique à sua investigação.

Nesse sentido, quanto a sua natureza a presente pesquisa é de caráter tecnológica, uma vez que utiliza conhecimentos básicos e tecnologias existentes sobre bancos de dados e tem como objeto um novo processo de utilização desta tecnologia para o desenvolvimento de aplicações.

Quanto aos objetivos a pesquisa pode ser classificada como exploratória com procedimento experimental de laboratório com finalidade apresentar novos conceitos de implementação de banco de dados utilizando a tecnologia objeto-relacional.

A pesquisa contou com apoio de base bibliográfica e documental baseadas em livros , revistas, teses e dissertações.

3.2. Procedimentos Metodológicos

Para a realização deste trabalho, primeiramente foi realizado um levantamento teórico sobre a área de bancos de dados e a introdução da tecnologia de orientação a objeto nesta. Em seguida um experimento foi realizado numa aplicação de banco de dados. Para isto, dois modelos de dados foram construídos, uma tendo por base o modelo relacional e outra o modelo objeto-relacional, afim de posteriormente estes modelos serem comparados.

O PostgreSQL foi escolhido para o desenvolvimento deste trabalho por ser um Sistema Gerenciador de Banco de Dados Objeto-Relacional (SGBDOR) livre, ou seja, dá suporte aos modelos de banco de dados relacional e objeto-relacional, sem custo para sua utilização.

A modelagem de dados foi realizada no programa DBDesigner Versão 4 uma ferramenta gratuita e apresentar flexibilidade. Alguns tipos de dados existentes no SGBD PostgreSQL não estão contidos no DBDesigner, porém esta ferranenta dá condições para o usuário criar os tipos que precisar usar em seu modelo.

As tabelas e as funções foram implementadas no PostgresSQL, por ser um SGBD com as duas características de modelagem, por um sistema livre e também pioneiro neste conceito de banco de dados objeto-relacional.

Os passos de construção para cada um dos modelos foram comparados, desde as facilidades e dificuldades na construção das tabelas, das funções da execução de operações básicas de inserção, atualização, remoção e manutenção no banco de dados.

Ao fim das comparações uma avaliação foi feita para decidir sobre a viabilidade de se utilizar a modelagem objeto-relacional em um novo projeto de banco de dados, ou ainda em migrar da tecnologia relacional para esta tecnologia híbrida.

4. RESULTADOS E DISCUSSÕES

4.1 Descrição da Aplicação

O sistema utilizado como modelo neste trabalho é uma aplicação de uma Empresa Transportadora. O sistema é de gerenciamento de clientes e de pedidos de serviços de transporte e possui as seguintes funcionalidades: Cadastrar Clientes, Cadastrar Pedido de Serviço e alguns tipos de consultas. O processo de cadastro constitui das operações de inserção, modificação e exclusão.

O Cliente pode ser uma Pessoa Física ou uma Pessoa Jurídica, sendo que alguns atributos são comuns aos dois como outros são específicos para cada. Os atributos comuns entre eles são o Nome, Data de Cadastro, Rua, Número, Complemento, Bairro, Cidade, Estado, Telefones, e-mail e Referência. O Cliente Pessoa Física possui CPF e Data de nascimento como atributos específicos já os atributos específicos para Pessoa Jurídica são CGC e Inscrição Estadual.

Os pedidos são classificados como Pedido de Mudança e Pedidos de Transporte de Carga. Os atributos comuns aos pedidos são: Data de Apanha, Local de Apanha, Responsável, Data de Entrega, Local de Entrega, Valor do Transporte, Valor do Assegurado, Forma de Pagamento, Situação do Pagamento e Observação. O atributo específico para Pedido de Mudança é Lista de Nome dos Móveis que serão transportados e para Pedido de Transporte de Carga é a Lista de Produtos contendo nome do produto, quantidade e peso.

Para processar o cadastro de Clientes e Pedidos no banco de dados da aplicação foram construídas funções específicas para o modelo relacional e para o modelo objeto-relacional, afim de serem posteriormente comparadas as duas implementações.

As seções a seguir apresentam os modelos de dados relacional e objeto-relacional da aplicação em questão.

4.2. Modelo de Dados Relacional

A Figura 3 apresenta um desenho do Modelo Relacional referente ao banco de dados da aplicação proposta.

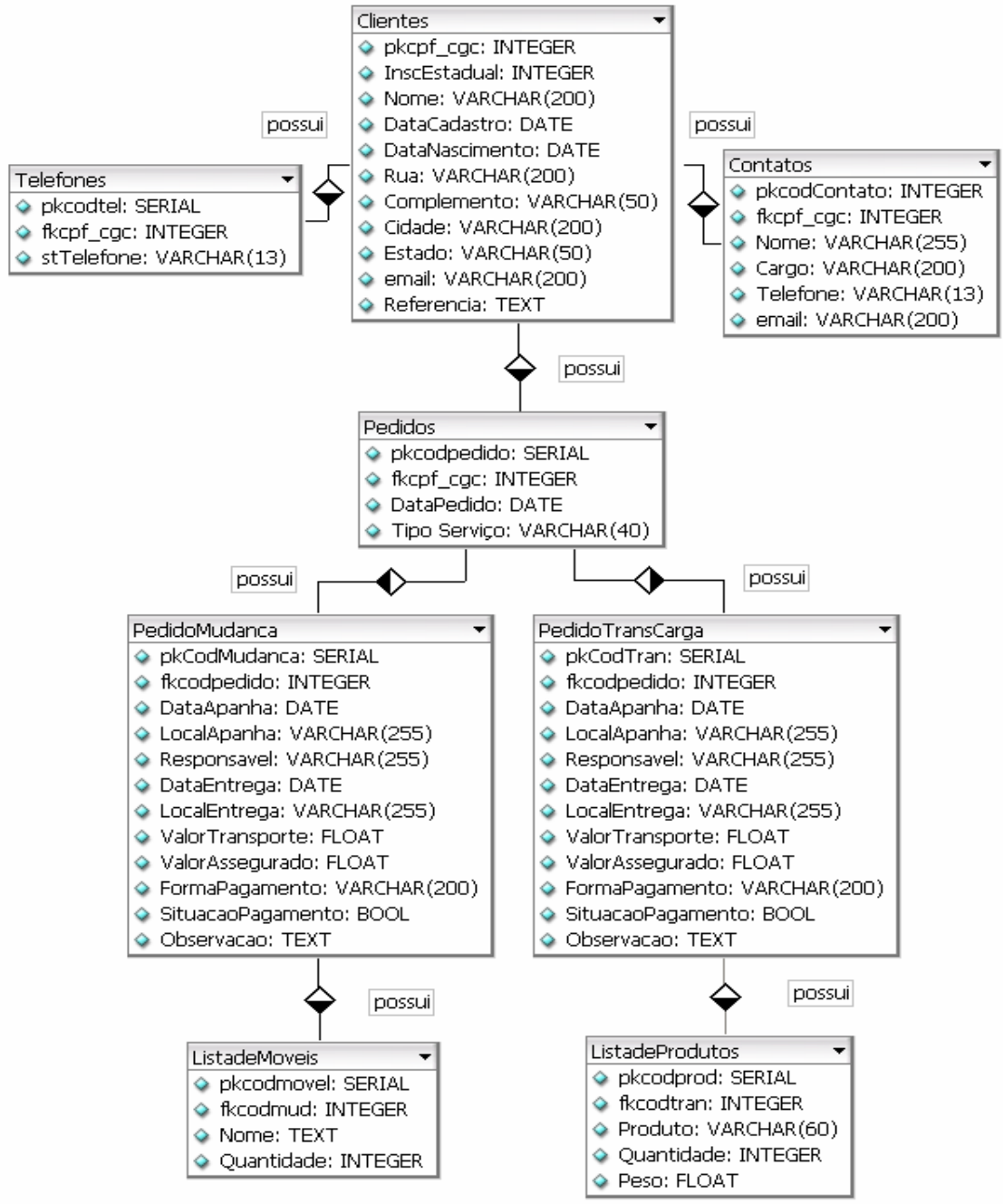

Figura 3: Modelo de Dados Relacional.
Fonte: Elaborada pelo autor.

O sistema baseado no modelo relacional possui oito entidades sendo elas denominadas Clientes, Telefones, Contatos, Pedidos (representa o relacionamento), Pedido de Mudança, Pedido de Transporte de Carga, Lista de Móveis e Lista de Produtos.

A tabela Clientes é utilizada tanto para armazenar Clientes Pessoa Física quanto Clientes Pessoa Jurídica. Como um Cliente pode possuir mais de um telefone e a quantidade de telefones pode variar de cliente para cliente foi criada uma tabela chamada Telefones somente para armazenar os telefones referentes ao cliente. A tabela denominada Contatos tem a mesma funcionalidade da tabela de Telefones já que para um Cliente podemos ter vários contatos referentes a ele.

A tabela Pedidos faz o relacionamento entre os Clientes e os Pedidos de Mudança ou Pedidos de Transporte de Carga. Houve a necessidade de criar tabela Lista de Móveis para associar ao pedido de mudança os móveis que serão transportados. Uma tabela de Lista de Produtos também foi criada para armazenar os produtos de um Pedido de Transporte de Carga.

Os *scripts* de criação das tabelas estão disponíveis no Anexo A e os *scripts* de criação das funções estão disponíveis no Anexo B.

4.3. Modelo de Dados Objeto-relacional

A Figura 4 apresenta um desenho do Modelo Objeto-relacional referente ao banco de dados da aplicação proposta.

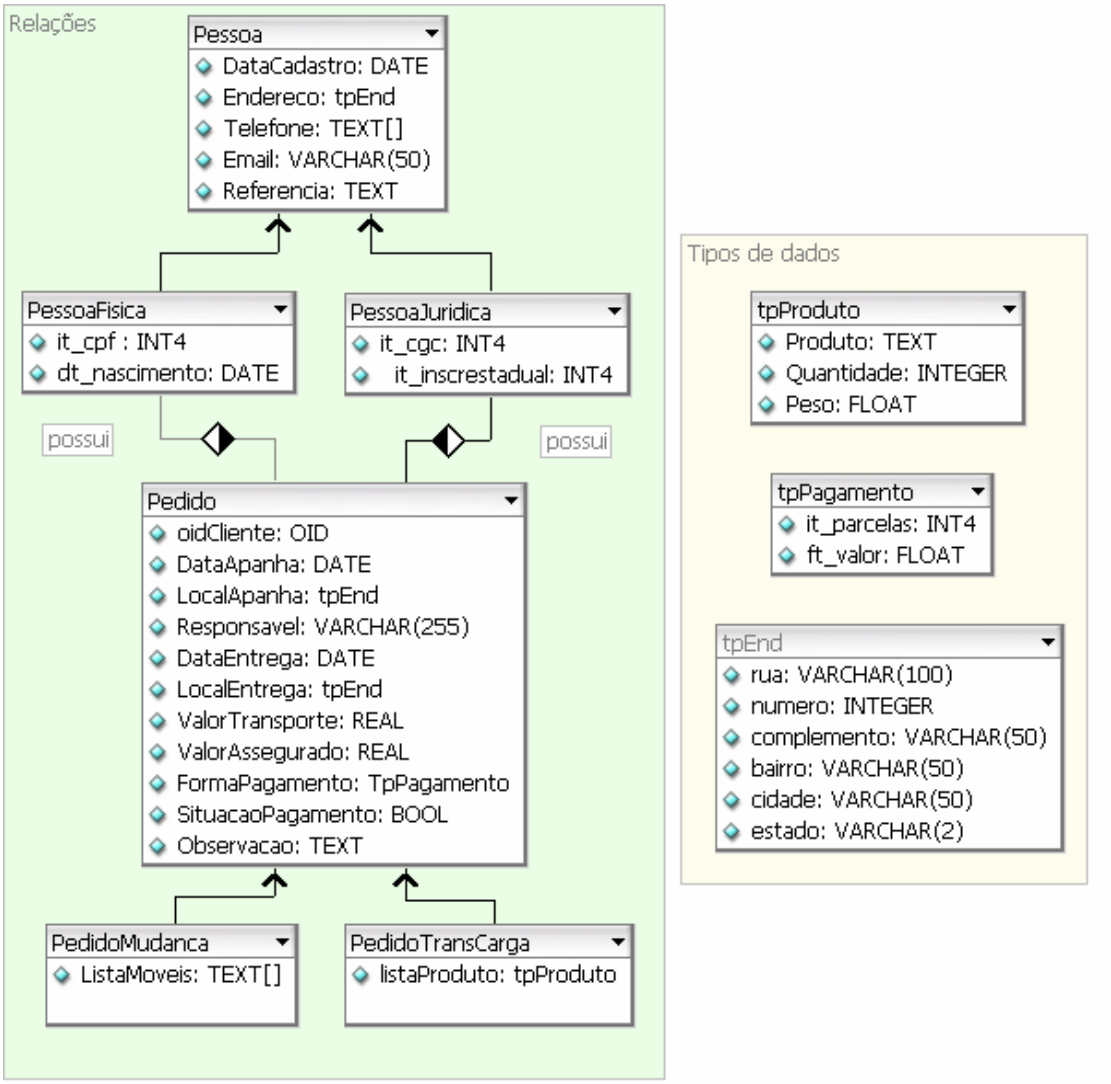

Figura 4: Modelo de Dados Objeto-relacional.
Fonte: Elaborado pelo autor.

Para a construção do modelo de dados da aplicação baseado no modelo objeto-relacional foram criadas seis entidades. São elas: Pessoa, Pessoa Física, Pessoa Jurídica, Pedidos, PedidoMudanca, PedidoTransCarga. As entidades PessoaFisica e PessoaJuridica herdam atributos da entidade pai denominada Pessoa, portanto sendo chamadas de entidades filhas, representando os conceitos de herança.

As entidades PedidoMudanca e PedidoTransCarga herdam os atributos da entidade Pedido utilizando também dos conceitos de herança oriundos do modelo orientado a objeto.

Vale destacar o tipo de dado do atributo Telefone oferecido pelo PostgresSQL que é um vetor do tipo texto declarado como text[], no qual é usado para armazenar atributo multivalorado. O endereço considerado um atributo composto pôde ser representado por um tipo de dado definido pelo usuário, este tipo foi denominado como tpEnd e possui os atributos rua, número, complemento, cidade, bairro e estado. Para o atributo forma de pagamento, denominado um atributo composto, foi criado o tipo de dado tpPagamento, com os atributos it_parcelas e it_valor representando respectivamente o número de parcelas e o valor da parcela. O tipo tpProduto armazena os atributos do atributo composto chamado produto da tabela PedidoTransCarga.

Os scripts de criação das tabelas estão disponíveis no Anexo C e os scripts de criação das funções estão disponíveis no Anexo D.

4.4. Análise Comparativa

Esta seção faz uma avaliação detalhada e cuidadosa sobre as formas de implementação de cada um dos modelos.

Na construção do banco de dados utilizando o modelo objeto-relacional, foi construído um número menor de tabelas. Isto foi possível devido a utilização de vetores para representar atributos multivalorados, que eliminou a necessidade da existência de uma nova tabela para este fim. Por outro lado, no modelo relacional os telefones de um cliente precisaram ser armazenados em uma tabela específica para isto.

A definição de tipos de dados pelo usuário possibilitou que o atributo composto endereço fosse representado diretamente na entidade, como um único atributo. Diminuindo o número de colunas das tabelas e aumentando a familiaridade com as linguagens de programação OO.

As Figuras 5 e 6 apresentam os *scripts* de criação das tabelas nos dois modelos distintos, que comparam as diferenças e exemplificam estes casos.

```
CREATE TABLE bdrel clientes
(
  pkcpf_cgc int4 NOT NULL,
  inscestadual int4,
  nome varchar(200),
  datacadastro date,
  datanascimento date,
  rua varchar(200),
  cidade varchar(60),
  estado varchar(2),
  email varchar(200),
  referencia text,
  tipo varchar(20),
  CONSTRAINT clientes_pkey PRIMARY KEY (pkcpf_cgc)
)

CREATE TABLE bdrel.telefones
(
  clientes_pkcpf_cgc int4 NOT NULL,
  sttelefone varchar(13),
  CONSTRAINT telefones_pkey PRIMARY KEY (clientes_pkcpf_cgc),
  CONSTRAINT telefones_clientes_pkcpf_cgc_fkey FOREIGN KEY (clientes_pkcpf_cgc)
REFERENCES bdrel.clientes (pkcpf_cgc) ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Figura 5: *Script* de criação das tabelas Cliente e Telefones no modelo relacional.
Fonte: Elaborado pelo autor.

```
CREATE TYPE bdobjrel.tpend AS
   (rua varchar(100),
    num int4,
    bairro varchar(50),
    complemento varchar(50),
    cidade varchar(50),
    estado varchar(2));

CREATE TABLE bdobjrel.pessoa
(
  nome varchar(100),
  datacadastro date,
  endereco bdobjrel.tpend,
  telefone text[],
  email varchar(50),
  referencia text
)
```

Figura 6: *Script* de criação da tabela Cliente e do tipo Endereço no modelo OR.
Fonte: Elaborado pelo autor.

O conceito de herança presente no modelo objeto-relacional eliminou o processo de mapeamento dos objetos das linguagens de programação para as relações existentes no banco de dados, evitou a construção de duas relações com atributos iguais entre si e otimizou o processo de consulta, inserção, atualização e remoção.

Para o modelo relacional três relações foram criadas e há a necessidade de chaves primárias e estrangeiras para estabelecer o relacionamento entre elas. As operações de *select*, *insert*, *update* e *delete* ficaram prejudicadas, pois deve ser feito acesso a pelo menos duas relações do banco de dados para a execução das mesmas. Por exemplo, para inserir um pedido é preciso especificar o tipo do pedido na aplicação e mapeá-lo nas relações Pedidos e PedMudanca ou PedTspCarga do modelo relacional.

No modelo objeto-relacional as operações de *select, insert*, *update* e *delete* foram otimizadas, pois precisaram ser executadas em apenas uma das relações, PedMudanca ou PedTspCarga. Além disso, no caso do comando *select*, se for preciso fazer uma consulta que retorne todos os pedidos existentes no banco de dados basta fazer uma consulta em uma relação Pedido, enquanto que, no caso do modelo relacional, seria preciso fazer uma união de duas relações. Exemplo de uma consulta que retorna todos os pedidos existentes no banco de dados objeto-relacional: *select* * *from* bdobjrel.pedido

O conceito de OID, onde cada objeto possui um identificador único dentro do banco de dados, diminuiu a necessidade da utilização das restrições de entidade e de integridade referencial (ou uso de chave primária e de chave estrangeira). Os identificadores de objetos permitem fazer referências entre objetos.

Por exemplo, no modelo objeto-relacional, o relacionamento entre o cliente pessoaFisica ou o cliente pessoaJuridica com a relação pedidos é feito usando o OID do cliente como referencia na relação pedido. A Figura 7 apresenta o código referente a este exemplo.

```
CREATE TABLE bdobjrel.pedido
(
  oid_cliente oid,
  datapedido date
  dataapanha date,
  localapanha bdobjrel.tpend,
  dataentrega date,
  localentrega bdobjrel.tpend,
  valortransporte float4,
  valorassegurado float4,
  formapagamento bdobjrel.tppagamento,
  situacaopagamento bool DEFAULT false,
  observacao text,
)
WITH OIDS;
```

Figura 7: Script de criação da tabela Pedido com uso de OID.
Fonte: Elaborado pelo autor.

O procedimento armazenado, *stored procedure*, permitiu o agrupamento de diversas funções SQL dentro de um bloco, tirando da aplicação a responsabilidade sobre as funções já que estas estão armazenadas no servidor, além do mais todo o processamento é feito de uma vez só, diminuindo o fluxo de dados entre o cliente e o servidor de banco de dados. Isso pode acarretar uma economia no tempo de desenvolvimento da aplicação, no tempo de processamento, e um possível aumento no desempenho do sistema.

A Figura 8 apresenta um exemplo de uma função que insere um pedido, mas antes é preciso fazer uma busca para saber qual é o OID do cliente que está fazendo o pedido para depois inserir na tabela do pedido especificado.

```
CREATE OR REPLACE FUNCTION bdobjrel.f_inserepedidomudanca(text, date, date,
bdobjrel.tpend, date, bdobjrel.tpend, _text, float8,  float8, bdobjrel.tppagamento, bool, text)
  RETURNS bool AS
$BODY$
declare
  oidcl oid;

begin
  select into oidcl oid from bdobjrel.pessoa where nome = $1;
  IF oidcl IS null THEN
    return false;
    ELSE
     insert into bdobjrel.ped_mudanca (oid_cliente, datapedido, dataapanha, localapanha,
dataentrega, localentrega,
      listainstalacao, valortransporte, valorassegurado, formapagamento, situacaopagamento,
observacao)
        values (oidcl, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12);
     return true;
  END IF ;
end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Figura 8: Exemplo de *stored procedure*.
Fonte: Elaborado pelo autor.

Os gatilhos, *triggers*, são recursos que auxiliam no reforço da integridade referencial de um banco de dados. Por exemplo, supondo que só é permitido excluir alguma pessoa se esta não possui nenhum pedido recente, considerando que um pedido recente é aquele feito a menos de 60 dias, portanto não deve-se apagar esse cliente, muito menos os pedidos referentes a ele. Quando um registro da relação pessoa vai ser excluído, um *trigger* associado a essa relação dispara e antes de excluir o registro ele verifica se essa pessoa possui algum pedido recente. Se existir algum pedido recente nada é feito, caso

contrário apaga-se a pessoa e todos os pedidos antigos referentes a esta pessoa. A Figura 9 apresenta o *trigger* referente a este exemplo.

```
CREATE TRIGGER tg_excluircliente
  BEFORE DELETE
  ON bdobjrel.pessoa
  FOR EACH ROW
  EXECUTE PROCEDURE bdobjrel.f_excluir_cliente();

CREATE OR REPLACE FUNCTION bdobjrel.f_excluir_cliente()
  RETURNS "trigger" AS
$BODY$
Declare
  ultimoPedido date;  data  date;  n_dias float;  oidCli oid;
BEGIN
  select into n_dias date(now()) - (SELECT max(datapedido) from bdobjrel.ped_mudanca
where oid_cliente = old.oid) AS dias;
  oidCli:= old.oid;
  if n_dias >= 60 or n_dias is null then
     DELETE FROM bdobjrel.ped_mudanca WHERE oid_cliente = oidCli ;
     RAISE NOTICE 'Cliente excluído com sucesso!';
  else
     RAISE EXCEPTION 'Cliente com pedido recente!';
  end if;
  return old;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Figura 9: Exemplo de trigger.
Fonte: Elaborado pelo autor.

O PostgreSQL dá suporte ao polimorfismo, que contribui para a criação de funções de mesmo nome e que desempenham o mesmo papel para diferentes objetos dentro do banco de dados. Um exemplo pode ser visto com o caso da função insereCliente que pode inserir um cliente pessoa física ou um cliente pessoa jurídica, ficando por parte do SGBD definir qual seu comportamento de acordo com os atributos passados pela aplicação, tirando a responsabilidade do programador de ter que decidir onde o cliente deverá ser inserido. A Figura 10 apresenta a criação de funções que representam este exemplo.

```
CREATE OR REPLACE FUNCTION bdobjrel.f_insereCliente(int4, int4, text, date, bdobjrel.tpend,
_text, text, text)
  RETURNS bool AS
$BODY$
declare
  cgc alias for $1;  insc_est alias for $2;  nom alias for $3;  dt_cad alias for $4;  ender alias for $5;
tel alias for $6;  e_mail alias for $7;  referen alias for $8;
  begin
  insert into bdobjrel.pessoajuridica
    (it_cgc, it_inscrestadual, nome, datacadastro, endereco, telefone, email, referencia )
      values (cgc, insc_est, nom, dt_cad, ender, tel, e_mail, referen);

    RAISE NOTICE 'Cliente PF inserido com sucesso!';
  return true;
end;  $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
CREATE OR REPLACE FUNCTION bdobjrel.f_insereclIente(int4, date, text, date, bdobjrel.tpend,
_text, text, text)
  RETURNS bool AS
$BODY$
declare
  cpf alias for $1;  dt_nasc alias for $2;  nom alias for $3;  dt_cad alias for $4;  ender alias for $5;
tel alias for $6;  e_mail alias for $7;  referen alias for $8;
  begin
  insert into bdobjrel.pessoafisica
    (it_cpf, dt_nascimento, nome, datacadastro, endereco, telefone, email, referencia )
      values (cpf, dt_nasc, nom, dt_cad, ender, tel, e_mail, referen);

    RAISE NOTICE 'Cliente PF inserido com sucesso!';
  return true;

end;  $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Figura 10: Exemplos de criação de funções – polimorfismo.
Fonte: Elaborado pelo autor.

Analisando a implementação do banco de dados objeto-relacional no PostgreSQL pode-se afirmar que este SGBD é bastante flexível, pois é compatível com o modelo de bancos de dados relacional e compatível com a teoria de programação orientada a objetos.

Devido ao fato da possibilidade da utilização de procedimentos armazenados e gatilhos, juntamente com a definição dos dados, a utilização deste tipo de SGBD minimiza o tempo gasto no desenvolvimento de aplicações, uma vez que o seu código será reduzido.

Todas as funções de manipulação de dados estando armazenadas no próprio SGBD, consequentemente reduz o gasto com processamento de comunicação entre o cliente e o servidor de banco de dados, principalmente quando estes estão em máquinas diferentes. Diante disto, a administração e a manutenção do banco de dados é facilitada.

5. CONCLUSÃO

Os objetivos propostos neste trabalho de fazer uma comparação entre duas implementações distintas de um mesmo sistema de banco de dados, utilizando dois modelos bancos de dados: o relacional e o objeto-relacional e analisar, a partir das comparações, se a adoção do modelo objeto-relacional para implementação de aplicações convencionais apresenta vantagens sobre o modelo relacional foram alcançados.

Pode-se observar que o PostgreSQL permite a implementação dos principias conceitos da tecnologia de orientação a objetos, tais como, identidade de objeto, herança, polimorfismo, construção de novos tipos de dados complexos, gatilhos, funções e procedimentos armazenados.

Os estudos realizados permitem concluir que, embora a ODMG defina um padrão banco de dados orientado a objetos, ainda há carência em relação à sua padronização, pois muitos SGBDs possuem suas próprias linguagens de programação ou implementam apenas parcialmente este padrão.

Os resultados das implementações e das análises feitas mostraram que o PostgreSQL é flexível, compatível com os bancos de dados relacionais, compatível com as linguagens de programação orientadas a objetos, a sua utilização minimiza o tempo gasto no desenvolvimento de aplicações, reduz o gasto com processamento de comunicação, facilitando assim a administração e a manutenção do banco de dados.

O trabalho realizado contribuiu com a pesquisa sobre a recente introdução da tecnologia de orientação a objetos na área de bancos de dados, em especial quando aplicada juntamente com a consolidada tecnologia de banco de dados relacional. Sendo assim, este trabalho apresenta a base teórica fundamental para a compreensão destas tecnologias e sua viabilidade no desenvolvimento de determinadas aplicações, dando suporte e orientação para quem a desconhece e deseja adotá-la ou para quem deseja migrar de um sistema relacional existente para um sistema objeto-relacional.

Como sugestão de trabalhos futuros sugere-se o desenvolvimento de uma aplicação que utilize o banco de dados objeto-relacional implementado, aproveitando todas as suas funcionalidades criadas.

Devido ao fato deste trabalho estar comparando características de programação referentes ao uso de diferentes modelos de bancos de dados, não foi desenvolvida uma aplicação para o sistema, impossibilitando que testes mais consistentes sobre desempenho

fossem realizados no mesmo. Sendo assim, este trabalho não teve como objetivo fazer uma comparação de desempenho entre as duas formas distintas de implementação. Porém sugere-se que isto seja feito num trabalho futuro.

6. REFERÊNCIAS BIBLIOGRÁFICAS

CARDOSO, Olinda N. P. Banco de Dados. Lavras: UFLA/FAEPE, 2003.

CODD, E. F. A Relational Model of Data for Large Shared Data Banks. Revista CACM.

DATE, J. C. Introdução a Sistemas de Bancos de Dados, tradução da 7a. Edição Americana, Rio de Janeiro. Campus, 2000.

ELMASRI, R. & NAVATHE, S. B. Sistemas de Banco de Dados 4ª Edição, São Paulo: Addison Wesley, 2005.

LARMAN, C. Utilizando UML e padrões. Porto Alegre: Bookman, 2000.

NEVES, Denise Lemes F. PostgreSQL: Conceitos e Aplicações. São Paulo: Érica, 2002.

OLIVEIRA, C. H. P. Bancos de Dados Livre x Pago. Disponível em: http://www.sqlmagazine.com.br/Artigos/Outros/01_Banco_FreeXPago.asp. Acessado em: 08.dez.2005.

POSTGRESQL, The Postgresql Global Development Group. Documentação do PostgreSQL 8.0.0 Disponível em: http://pgdocptbr.sourceforge.net/pg80/. Acessado em: 27.mar.06.

RODRIGUES JÚNIOR, M. C. A Implementação Objeto-relacional no Oracle, SQL Magazine, Edição 15, Ano 2, página 8, 2005.

SILBERSCHATZ, Abranham; KORTH, Henry F. e SUDARSHAN, S. Sistema de Banco de Dados. 3ª ed. – São Paulo: MAKRON Books, 1999.

STONEBRAKER, M. ROWE, LAWRENCE A. e HIROHAMA, M. The implementation of POSTGRES. Transactions on Knowledge and Data Engineering, Vol. 2, NO. 1, March 1990.

ANEXO A

Script de criação das tabelas do modelo de dados relacional.

```
CREATE TABLE bdrel.clientes
(
  pkcpf_cgc int4 NOT NULL,
  inscestadual int4,
  nome varchar(200),
  datacadastro date,
  datanascimento date,
  rua varchar(200),
  cidade varchar(60),
  estado varchar(2),
  email varchar(200),
  referencia text,
  tipo varchar(20),
  CONSTRAINT clientes_pkey PRIMARY KEY (pkcpf_cgc)
);

CREATE TABLE bdrel.contatos
(
  idcontato serial NOT NULL,
  cliente_cpf_cgc int4 NOT NULL,
  nome varchar(100),
  cpf int4,
  cargo varchar(50),
  telefone varchar[],
  email varchar(50),
  CONSTRAINT contatos_pkey PRIMARY KEY (idcontato),
  CONSTRAINT contatos_cliente_cpf_cgc_fkey FOREIGN KEY (cliente_cpf_cgc)
REFERENCES bdrel.clientes (pkcpf_cgc) ON UPDATE CASCADE ON DELETE
CASCADE
);

CREATE TABLE bdrel.telefones
(
  clientes_pkcpf_cgc int4 NOT NULL,
  sttelefone varchar(13),
  CONSTRAINT telefones_pkey PRIMARY KEY (clientes_pkcpf_cgc),
  CONSTRAINT telefones_clientes_pkcpf_cgc_fkey FOREIGN KEY
(clientes_pkcpf_cgc) REFERENCES bdrel.clientes (pkcpf_cgc) ON UPDATE NO
ACTION ON DELETE NO ACTION
);
```

CREATE TABLE bdrel.pedidos
(
  pkcodpedido serial NOT NULL,
  fkcliente_cpf_cgc int4 NOT NULL,
  tiposervico varchar(40),
  datapedido date,
  CONSTRAINT pedidos_pkey PRIMARY KEY (pkcodpedido),
  CONSTRAINT pedidos_fkcliente_cpf_cgc_fkey FOREIGN KEY (fkcliente_cpf_cgc)
REFERENCES bdrel.clientes (pkcpf_cgc) ON UPDATE CASCADE ON DELETE
CASCADE
);

CREATE TABLE bdrel.pedidomudancas
(
  pkcodmud int4 NOT NULL DEFAULT nextval('bdrel.mudancas_pkcodmud_seq'::text),
  fkcodpedido int4 NOT NULL,
  dataapanha date,
  localapanha varchar(255),
  dataentrega date,
  localentrega varchar(255),
  valortransporte float4,
  valorassegurado float4,
  formapagamento varchar(200),
  situacaopagamento varchar(200),
  observacao text,
  CONSTRAINT mudancas_pkey PRIMARY KEY (pkcodmud),
  CONSTRAINT mudancas_fkcodpedido_fkey FOREIGN KEY (fkcodpedido)
REFERENCES bdrel.pedidos (pkcodpedido) ON UPDATE CASCADE ON DELETE
CASCADE
);

CREATE TABLE bdrel.listademoveis
(
  pkcodmovel serial NOT NULL,
  "fkCodmud" int4 NOT NULL,
  movel varchar(60) NOT NULL,
  quantidade int4 NOT NULL,
  CONSTRAINT listademoveis_pkey PRIMARY KEY (pkcodmovel),
  CONSTRAINT "listademoveis_fkCodmud_fkey" FOREIGN KEY ("fkCodmud")
REFERENCES bdrel.pedidomudancas (pkcodmud) ON UPDATE RESTRICT ON
DELETE RESTRICT
);

CREATE TABLE bdrel.pedidotranscarga

```
(
  pkcodtran int4 NOT NULL DEFAULT nextval('bdrel.transcarga_pkcodtran_seq'::text),
  fkcodpedido numeric NOT NULL,
  dataapanha date,
  localapanha varchar(255),
  dataentrega date,
  localentrega varchar(255),
  valortransporte float4,
  valorassegurado float4,
  formapagamento varchar(200),
  situacaopagamento varchar(200),
  observacao text,
  CONSTRAINT transcarga_pkey PRIMARY KEY (pkcodtran),
  CONSTRAINT transcarga_fkcodpedido_fkey FOREIGN KEY (fkcodpedido)
REFERENCES bdrel.pedidos (pkcodpedido) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

CREATE TABLE bdrel.listadeprodutos

```
(
  produto varchar(60),
  quantidade int4,
  peso float8,
  fkcodtran int4 NOT NULL,
  "pkCodlistaprod" int4 NOT NULL DEFAULT
nextval('bdrel."listadeprodutos_pkCodlistaprod_seq"'::text),
  CONSTRAINT listadeprodutos_pkey PRIMARY KEY ("pkCodlistaprod"),
  CONSTRAINT listadeprodutos_fkcodtran_fkey FOREIGN KEY (fkcodtran)
REFERENCES bdrel.pedidotranscarga (pkcodtran) ON UPDATE RESTRICT ON
DELETE RESTRICT
) ;
```

ANEXO B

Script de criação das funções do modelo de dados relacional.

CREATE OR REPLACE FUNCTION bdrel.inserepessoafisica(int4, text, date, date, text, text, text, text, text, text)

```
  RETURNS bool AS
$BODY$
declare
  itcpf alias for $1;  stnome alias for $2;  dtnasc alias for $3;  dtcad alias for $4;  strua alias for $5;
  stcidade alias for $6;  stestado alias for $7;  stemail alias for $8;  streferencia alias for $9;
sttipo alias for $10;
begin

    insert into  bdrel.clientes (pkcpf_cgc,nome,datanascimento,
dataCadastro,rua,cidade,estado,email,referencia)
                            values(itcpf, stnome, dtnasc, dtcad, strua, stcidade, stestado,
stemail, streferencia);
    return true;

end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

CREATE OR REPLACE FUNCTION bdrel.inserepessoajuridica(int4, text, int4, date, text, text, text, text, text, text)

```
  RETURNS bool AS
$BODY$
declare
  itcgc alias for $1;  stnom alias for $2;  itinsc alias for $3;  dtcad alias for $4;  strua alias for $5;
  stcidade alias for $6;  stestado alias for $7;  stemail alias for $8;  streferencia alias for $9;
sttipo alias for $10;
begin

    insert into  bdrel.clientes (pkcpf_cgc, nome, inscestadual, dataCadastro, rua, cidade,
estado, email,
      referencia, tipo)
   values(itcgc, stnom, itinsc, dtcad, strua, stcidade, stestado, stemail, streferencia,sttipo);
    return true;
 end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

```
CREATE OR REPLACE FUNCTION bdrel.inseremudanca(text, text, text, text, text)
  RETURNS bool AS
$BODY$
declare
  cpf_cgc  int;   codPedido int;  stnome alias for $1;  dtpedido alias for $2;  stservico alias
for $3;  dtapanha alias for $4;  stlocalapanha alias for $5;
Begin
  /*recupero cpf_cgc do cliente */
  cpf_cgc =  bdrel.recupera_cpfcgc(stnome);
  /*insere o pedido para o cliente */
  insert  into bdrel.Pedidos (fkcliente_cpf_cgc, datapedido, tiposervico) values (cpf_cgc,
dtpedido::date, stservico);  /*dtpedido::date (converte a string passada no tipo data)*/

  /* recupera o codigo pedido do cliente referente a data */
  codPedido  = bdrel.recuperacodpedido(cpf_cgc, dtpedido::date);
  insert into bdrel.pedidomudancas (fkcodpedido, dataapanha, localapanha) values
(codPedido,
      dtapanha::date, stlocalapanha);
  return true;
end;$BODY$
  LANGUAGE 'plpgsql' VOLATILE;


CREATE OR REPLACE FUNCTION bdrel.inseretranspcarga(text, text, text, text,
text)
  RETURNS bool AS
$BODY$
declare
  cpf_cgc  int;    codPedido int;  stnome alias for $1;  dtpedido alias for $2;  stservico alias
for $3;  dtapanha alias for $4;  stlocalapanha alias for $5;
Begin
  /*recupero cpf_cgc do cliente */
  cpf_cgc =  bdrel.recupera_cpfcgc($1);
  /*Insere o pedido*/
  insert  into bdrel.Pedidos (fkcliente_cpf_cgc, datapedido, tiposervico) values (cpf_cgc,
dtpedido::date, stservico);  /*dtpedido::date (converte a string passada no tipo data)*/

  /* recupera o codigo pedido do cliente referente a data*/
  codPedido  = bdrel.recuperacodpedido(cpf_cgc, dtpedido::date);
  /*insere  o serviço de transporte de materiais*/
  insert into bdrel.pedidotransCarga (fkcodpedido, dataapanha, localapanha) values
(codPedido, dtapanha::date, stlocalapanha);

  return true;

end;$BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

```
CREATE OR REPLACE FUNCTION bdrel.recupera_cpfcgc(text)
  RETURNS SETOF int4 AS
$BODY$
   select pkcpf_cgc  from bdrel.clientes where nome = $1;

$BODY$
  LANGUAGE 'sql' VOLATILE;


CREATE OR REPLACE FUNCTION bdrel.recuperacodpedido(int4, text)
  RETURNS SETOF int4 AS
$BODY$
     select pkcodpedido  from bdrel.pedidos where fkcliente_cpf_cgc = $1 and datapedido =
$2::date;
$BODY$
  LANGUAGE 'sql' VOLATILE;
```

ANEXO C

Script de criação de tipos de dados do modelo de dados objeto-relacional.

```
CREATE TYPE bdobjrel.tpend AS
(
   rua varchar(100),
   num int4,
   bairro varchar(50),
   complemento varchar(50),
   cidade varchar(50),
   estado varchar(2)
);

create type bdobjrel.tpPagamento as(
   it_parcelas int,
   ft_valor float
)

CREATE TYPE bdobjrel.tpproduto AS
(
        produto text,
        quantidade integer,
        peso float
)
```

Script de criação de tabelas do modelo de dados relacional.

```
CREATE TABLE bdobjrel.pessoa(
  nome varchar(100),
  datacadastro date,
  endereco bdobjrel.tpend,
  telefone text[],
  email varchar(50),
  referencia text
) WITH OIDS;

create table bdobjrel.pessoafisica(
   it_cpf int unique,
   dt_nascimento date
)inherits(bdobjrel.pessoa)
WITH OIDS;
```

```
create table bdobjrel.pessoajuridica
(
   it_cgc int unique,
   it_inscrestadual int
)inherits(bdobjrel.pessoa)
WITH OIDS;

CREATE TABLE bdobjrel.pedido
(
  dataapanha date,
  localapanha bdobjrel.tpend,
  dataentrega date,
  localentrega bdobjrel.tpend,
  valortransporte float4,
  valorassegurado float4,
  formapagamento bdobjrel.tppagamento,
  situacaopagamento bool DEFAULT false,
  observacao text,
  oid_cliente oid,
  datapedido date
)
WITH OIDS;

create table bdobjrel.ped_mudanca
(
    listainstalacao text[2]
)
inherits(bdobjrel.pedido)


create table bdobjrel.ped_tspcarga
(
    produto bdobjrel.tpproduto
)
inherits(bdobjrel.pedido)
```

ANEXO D

Script de criação das funções do modelo de dados objeto-relacional.

CREATE OR REPLACE FUNCTION bdobjrel.f_inserecliente(int4, date, text, date, bdobjrel.tpend, _text, text, text)

```
  RETURNS bool AS
$BODY$
declare
  cpf alias for $1;  dt_nasc alias for $2;  nom alias for $3;  dt_cad alias for $4;  ender alias for $5;  tel alias for $6;  e_mail alias for $7;  referen alias for $8;
begin
     insert into bdobjrel.pessoafisica
    (it_cpf, dt_nascimento, nome, datacadastro, endereco, telefone, email, referencia )
      values (cpf, dt_nasc, nom, dt_cad, ender, tel, e_mail, referen);

    RAISE NOTICE 'Cliente PF inserido com sucesso!';
  return true;

end;  $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

CREATE OR REPLACE FUNCTION bdobjrel.f_inserecliente(int4, int4, text, date, bdobjrel.tpend, _text, text, text)

```
  RETURNS bool AS
$BODY$
declare
  cgc alias for $1;  insc_est alias for $2;  nom alias for $3;  dt_cad alias for $4;  ender alias for $5;  tel alias for $6;  e_mail alias for $7;  referen alias for $8;
  begin
  insert into bdobjrel.pessoajuridica
    (it_cgc, it_inscrestadual, nome, datacadastro, endereco, telefone, email, referencia )
      values (cgc, insc_est, nom, dt_cad, ender, tel, e_mail, referen);

    RAISE NOTICE 'Cliente PF inserido com sucesso!';
  return true;

end;  $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

CREATE OR REPLACE FUNCTION bdobjrel.f_alterarpessoafisica(int4, date, text, date, bdobjrel.tpend, _text, text, text)

```
 RETURNS bool AS
$BODY$
declare
 cpf alias for $1;  dt_nasc alias for $2;  nom alias for $3;  dt_cad alias for $4;  ender alias
for $5;  tel alias for $6;  e_mail alias for $7;  referen alias for $8;
 begin
 UPDATE bdobjrel.pessoafisica
    SET
      it_cpf = cpf,  dt_nascimento = dt_nasc, nome = nom,  datacadastro = dt_cad,
endereco = ender,       telefone = tel,  email = e_mail,          referencia = referen
      WHERE it_cpf = cpf ;

 IF NOT FOUND THEN
  RAISE EXCEPTION 'não foi encontrado o cliente !';
  return false;
 ELSE
  return true;
 END IF;

end; $BODY$
 LANGUAGE 'plpgsql' VOLATILE;
```

CREATE OR REPLACE FUNCTION bdobjrel.f_alterarpessoajuridica(int4, int4, text, date, bdobjrel.tpend, _text, text, text) RETURNS bool AS

```
$BODY$
declare
 cgc alias for $1;   insc_est alias for $2;  nom alias for $3;  dt_cad alias for $4;  ender alias
for $5;  tel alias for $6;  e_mail alias for $7;  referen alias for $8;
 begin
 UPDATE bdobjrel.pessoajuridica
    SET
      it_cgc = cgc,  it_inscrestadual = insc_est,   nome = nom,        datacadastro = dt_cad,
endereco = ender,       telefone = tel,  email = e_mail,          referencia = referen
      WHERE it_cgc = cgc ;

 IF NOT FOUND THEN
  RAISE EXCEPTION 'não foi encontrado o cliente !';
  return false;
 ELSE
  return true;
 END IF;
end; $BODY$
 LANGUAGE 'plpgsql' VOLATILE;
```

```
CREATE OR REPLACE FUNCTION bdobjrel.f_inserepedidomudanca(text, date,
date, bdobjrel.tpend, date, bdobjrel.tpend, _text, float8, float8, dobjrel.tppagamento,
bool, text)  RETURNS bool AS $BODY$
declare
  oidcl oid;

begin
  select into oidcl oid from bdobjrel.pessoa where nome = $1;
  IF oidcl IS null THEN
    return false;
     ELSE
     insert into bdobjrel.ped_mudanca (oid_cliente, datapedido, dataapanha, localapanha,
dataentrega, localentrega,
       listainstalacao, valortransporte, valorassegurado, formapagamento,
situacaopagamento, observacao)
         values (oidcl, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12);
     return true;
  END IF ;
end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE;


CREATE OR REPLACE FUNCTION bdobjrel.f_inserepedidotspcarga(text, date,
date, bdobjrel.tpend, date, bdobjrel.tpend, text, float8, float8, float8,
bdobjrel.tppagamento, text, text)
  RETURNS bool AS
$BODY$
declare
  oidcl oid;

begin
  select into oidcl oid from bdobjrel.pessoa where nome = $1;
  IF oidcl IS null THEN
    return false;
    ELSE
     insert into bdobjrel.ped_tspcarga (oid_cliente, datapedido, dataapanha, localapanha,
dataentrega, localentrega,
       produto, peso, valortransporte, valorassegurado, formapagamento, situacaopagamento,
observacao)
         values (oidcl, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13);
     return true;
  END IF ;
end;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

```
CREATE OR REPLACE FUNCTION bdobjrel.f_listarclientesdevedores()
  RETURNS SETOF bdobjrel.pessoa AS
$BODY$
  select nome, datacadastro, endereco, telefone, email, referencia
    from bdobjrel.pessoa, bdobjrel.ped_mudanca
    where bdobjrel.pessoa.OID = bdobjrel.ped_mudanca.OID_cliente and
         bdobjrel.ped_mudanca.situacaopagamento = false;
$BODY$

  LANGUAGE 'sql' VOLATILE;
```